Web Security and Malware Analysis

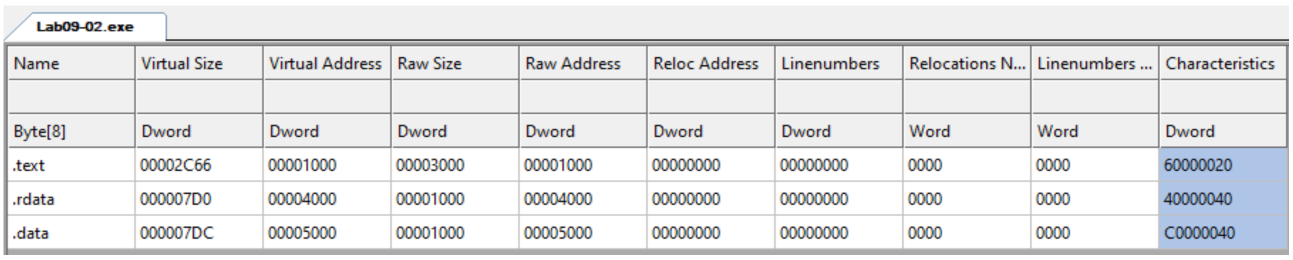
Answers for Assignment 9

SILVIA LUCIA SANNA – 70/90/00053

Task 1 – Full Malware Analysis

In this task I have to analyse a malware and retrieve as much info as possible.

I start analysing the PE structure from which I understand is a Windows executable as its magic number is 5A4D (or better as is in little endian: 4D5A) and can be executed in a 32 bit machine with Intel 386 and a specific OS version that can be found in majorOS and minorOS in the optional header (respectively with value 4 and 0 that I did not find to what corresponds). Analysing the optional header, I know different sizes of the executable: i.e. size of code 3000, size of initialized data 2000, no uninitialized data, image base 400000 (in fact all addresses start with 40xxxx), base of code 1000 (the first address of code area is in fact 401000), base of data 4000 (the first address of data is in fact 404000). These can also be confirmed in the section header. Last, in the imported directories all imported dll are listed: kernel32.dll, ws2_32.dll.

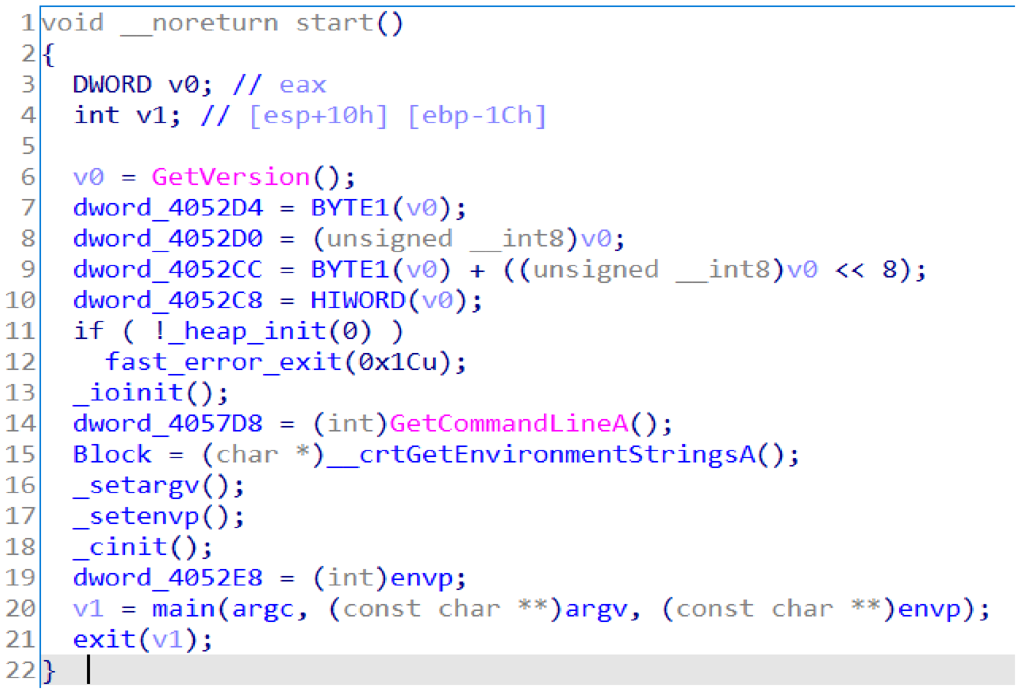

Lab09-02.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00002C66	00001000	00003000	00001000	00000000	00000000	0000	0000	60000020
.rdata	000007D0	00004000	00001000	00004000	00000000	00000000	0000	0000	40000040
.data	000007DC	00005000	00001000	00005000	00000000	00000000	0000	0000	C0000040

Figure 1: section header

After this, I immediately analyse the strings in the String window of IDA. In this section there is no string whose address starts with .data, even if in the section header there is a .data section and also even if in IDA View-A there are some strings in .data and two of them are particularly interesting: “cmd” and “unk_405034”, respectively at .data:00405030 and .data:00405034. In the String window, between all .rdata strings, the most interesting ones are: “Runtime Error”, “<program name unknown>”, “CreateProcessA”, “Sleep”, “GetModuleFileNameA”, “WSASocketA”, “GetCommandLineA”.

Analysing the code statically, I found two interesting “first” functions from which the code should be start: main and start.



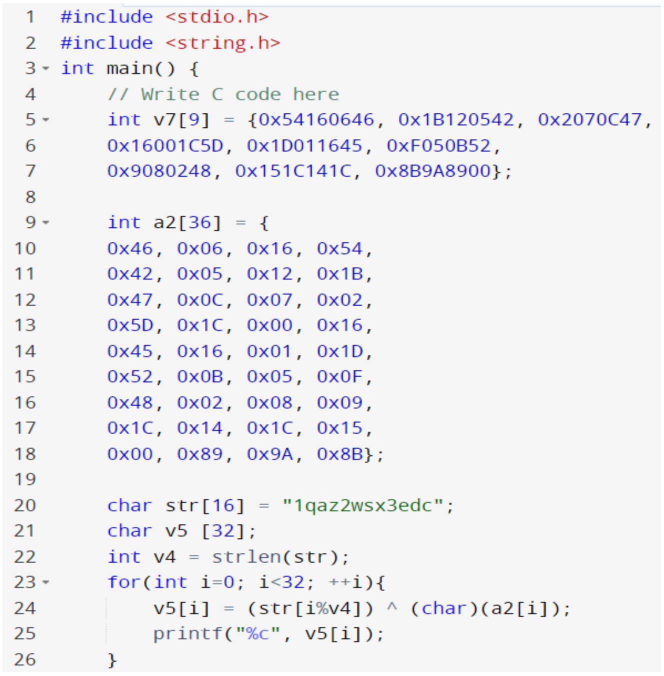
```
void __noreturn start()
{
  DWORD v0; // eax
  int v1; // [esp+10h] [ebp-1Ch]

  v0 = GetVersion();
  dword_4052D4 = BYTE1(v0);
  dword_4052D0 = (unsigned __int8)v0;
  dword_4052CC = BYTE1(v0) + ((unsigned __int8)v0 << 8);
  dword_4052C8 = HIWORD(v0);
  if ( !_heap_init(0) )
    fast_error_exit(0x1Cu);
  _ioinit();
  dword_4057D8 = (int)GetCommandLineA();
  Block = (char *)__crtGetEnvironmentStringsA();
  _setargv();
  _setenvp();
  _cinit();
  dword_4052E8 = (int)envp;
  v1 = main(argc, (const char **)argv, (const char **)envp);
  exit(v1);
}
```

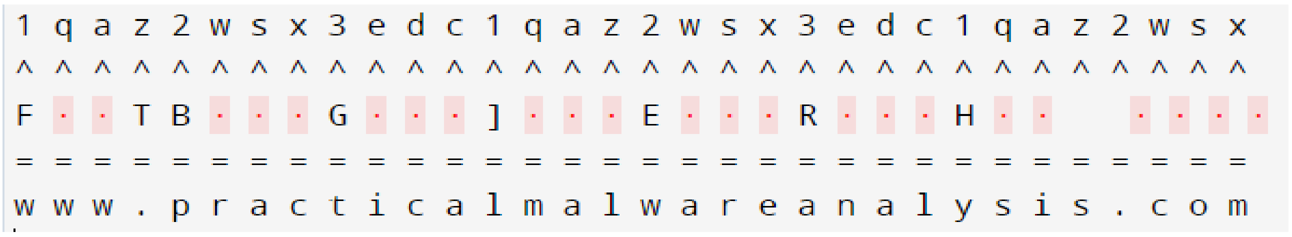
Figure 2: decompiled code for function start

I first analysed the function start and, as the name says, it should be the first function to be executed. Here there is the check of the current OS version of the machine where the code is running, then retrieves the command-line string for the current process (which is itself, the executable) and calls for the main.

In the main two strings are initialized (Str and Str1) first, then the array v7 is filled in with hexadecimal values found at memory area .data:00405034 (unk_405034) and array v6 is filled in with null values. Then the API GetModuleFileNameA is called to retrieve the path of the executable and only the last part (the name of the executable with the extension) is saved and compared with a string: if the name of the executable is not “ocl.exe” it will exit. So, from here I realised that the executable, to continue, must be called “ocl.exe”. In fact, if its name is “ocl.exe”, initiates the use of Winsock.dll by the current process, a socket is created with specific parameters (it belongs to IPv4 family, its type is sock_stream and uses TCP protocol).

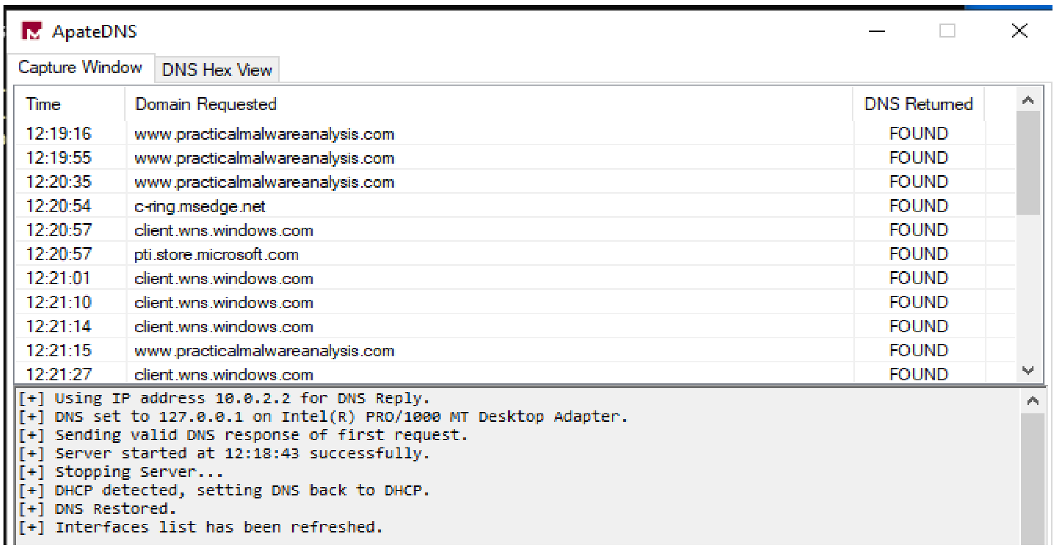
After this the function 401089 is called where a xor is made between the values found in v7 (each integer contains 4 chars retrieved from memory and stored in little endian, as they are read from the memory area) and the string Str initialized before in the first lines of the memory: this xor is repeated 32 times as the length of characters in v7 and so Str is repeated to fill in the 32 length as it is composed only by 12 characters and its length is 16. So, each character of Str is xored with the correspondent i-th character of v7, which as is in hexadecimal, is converted to char. With a little reverse script, I can know the value of this string "www.practicalmalwareanalysis.com": of course, it will not be present in the strings or data section as is not in cleartext.


```c
#include <stdio.h>
#include <string.h>
int main() {
    // Write C code here
    int v7[9] = {0x54160646, 0x1B120542, 0x2070C47,
    0x16001C5D, 0x1D011645, 0xF050B52,
    0x9080248, 0x151C141C, 0x8B9A8900};

    int a2[36] = {
    0x46, 0x06, 0x16, 0x54,
    0x42, 0x05, 0x12, 0x1B,
    0x47, 0x0C, 0x07, 0x02,
    0x5D, 0x1C, 0x00, 0x16,
    0x45, 0x16, 0x01, 0x1D,
    0x52, 0x0B, 0x05, 0x0F,
    0x48, 0x02, 0x08, 0x09,
    0x1C, 0x14, 0x1C, 0x15,
    0x00, 0x89, 0x9A, 0x8B};

    char str[16] = "1qaz2wsx3edc";
    char v5 [32];
    int v4 = strlen(str);
    for(int i=0; i<32; ++i){
        v5[i] = (str[i%v4]) ^ (char)(a2[i]);
        printf("%c", v5[i]);
    }
```
Figure 3: reverse script in C made by me to understand what happens in function 401089


```
1 q a z 2 w s x 3 e d c 1 q a z 2 w s x 3 e d c 1 q a z 2 w s x
^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
F · · T B · · · G · · · ] · · · E · · · R · · · H · ·   · · · ·
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
w w w . p r a c t i c a l m a l w a r e a n a l y s i s . c o m
```
Figure 4: output values of script in figure 3 (in the first line there are all characters of Str repeated almost 3 times to fill in length 32; in the second line I put a line of xor symbols; the third line contains the values of v7 represented in char format; in the fourth line I put a line of equals to express the result of the following line; in the last line there is the output of the function. So, you can also read this vertically: ie 1^F=w and so on).

Then, in the main, the value of this string is passed to gethostbyname which will check if that host name is present in the hostnames database. If it is present, will set parameters data and family in the socket address and make the connection to the socket using the just set socket address. If no error in the connection appears, it will call function 401000 which will set the values of two structures: StartupInfo and ProcessInfo that will be used to create a process with the CommandLine (here explained the use of “cmd” string) and waits until a specific process or time elapses. When the process “appears”, it will close the function and come back in the while loop in main and starts again with socket connection. If there is some error with the connection or with the hostname, the loop will continue looking for the connection: it will exit only if it is not able to initialize WSA or connect to the socket.

To confirm this, I made some dynamic analysis, of course changing the name of the executable. I used ApateDNS to confirm the name of the connection and how many times it will be repeated. As in the image below, 3 connections to the “www.practicalmalwareanalysis.com” are made but there are also other connections to different hostnames. I did not understand why the malware (the only process running) made this connections but googling them I discovered that it makes something like to update the system (even if the check for the current OS version is made in function start that should be executed before main).



Time	Domain Requested	DNS Returned
12:19:16	www.practicalmalwareanalysis.com	FOUND
12:19:55	www.practicalmalwareanalysis.com	FOUND
12:20:35	www.practicalmalwareanalysis.com	FOUND
12:20:54	c-ring.msedge.net	FOUND
12:20:57	client.wns.windows.com	FOUND
12:20:57	pti.store.microsoft.com	FOUND
12:21:01	client.wns.windows.com	FOUND
12:21:10	client.wns.windows.com	FOUND
12:21:14	client.wns.windows.com	FOUND
12:21:15	www.practicalmalwareanalysis.com	FOUND
12:21:27	client.wns.windows.com	FOUND

```
[+] Using IP address 10.0.2.2 for DNS Reply.
[+] DNS set to 127.0.0.1 on Intel(R) PRO/1000 MT Desktop Adapter.
[+] Sending valid DNS response of first request.
[+] Server started at 12:18:43 successfully.
[+] Stopping Server...
[+] DHCP detected, setting DNS back to DHCP.
[+] DNS Restored.
[+] Interfaces list has been refreshed.
```

Figure 5: output of ApateDNS

In this task I do not find useful to analyse registers as if everything goes fine this program is an infinite loop, so the value of the changed registers depends strictly on the time of the acquisition.

To confirm the flow of the program, I decided to debug it. If I debug Lab09-02.exe, as previously said, it blocks when makes the check of the executable name, which must be ocl.exe. When debugging ocl.exe, the name output of xor function 401089 is exactly what I computed with my function, which is “www.practicalmalwareanalysis.com”.

Continuing debugging, the next instruction is the one that performs the connection to the socket (for which I discovered the socket addresses: sa_family=2u, sa_data=”’”), but the exit code of function connect is -1 so it will not call function 401000 and comes back to WSAStartup. As I want to know if function 401000 really opens the command line and waits for a specific object, I can set the IP to .text:004013A9 (call to sub_401000), jump to IP and then F7 to step into the function and analyse it. Analysing this function, I can know the values of the process to be opened and the value of the waiting object. The StartupInfo is a string very big and I cannot visualize it entirely but only a portion up to value cbReserved2=0u, but I can know the value of ProcessInfo: hProcess=0x140, hThread=0x13C, dwProcessId=0x484, dwThreadId=0x1620. To better understand the process, I tried to google this values but I did not find something interesting.

In conclusion, this executable makes a connection to a specific host “www.practicalmalwareanalysis.com”, makes a socket connection and then opens the shell and waits for a specific process. As it remains opened, I can classify this as a backdoor, above all a reverse shell as it is created through CreateProcess API and also a socket is created. To confirm my thesis, I used virustotal which says that it is a malware, probably a trojan.

Task 2 – Windows API Functionality

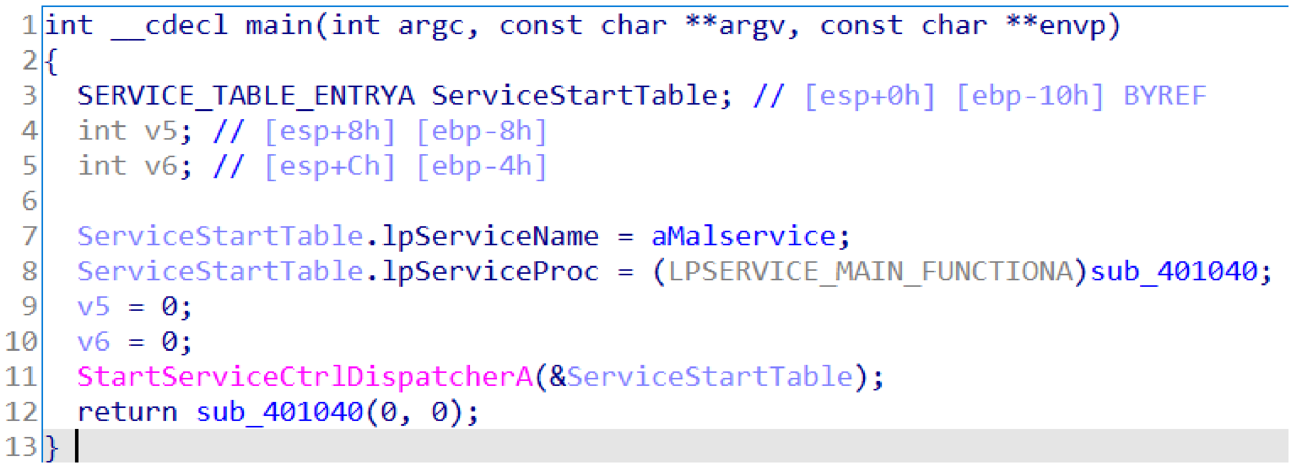
In this task I have to analyse a malware paying attention to some Windows API-related structures like mutex, threads, services, communication API, analysing them statically or dynamically. Before analysing those structures, I decided to analyse the executable statically and understand how it works.

In the main function there is the initialization of the structure ServiceStartTable, setting the name and the process, respectively with value “MalService” and the output of function

401040, so this structure is used to define the service to be runned in the calling process. Then, this structure will be used as input to StartServiceCtrlDispatcherA that connects the thread of the service to SCM and this thread is used as a dispatcher for the calling process. So, to know which is the name of the calling process I have to analyse the function 401040.

In function 401040, first of all the executable tries to open mutex “HGL345” and if it does not succeed it creates it, then connects to SCM on local computer opening the default database and creates a service at the path of the current running executable. Then sets a specific time (month, day, week day, hours…) and waits for it: while waiting creates 20 threads that will execute Internet Explorer 8.0 connecting to “http://malwareanalysisbook.com”; when the time is reached will sleep for 49 days. So, the process set in the ServiceStartTable is a count-down up to a specific day meanwhile creates some threads to a specific URL.

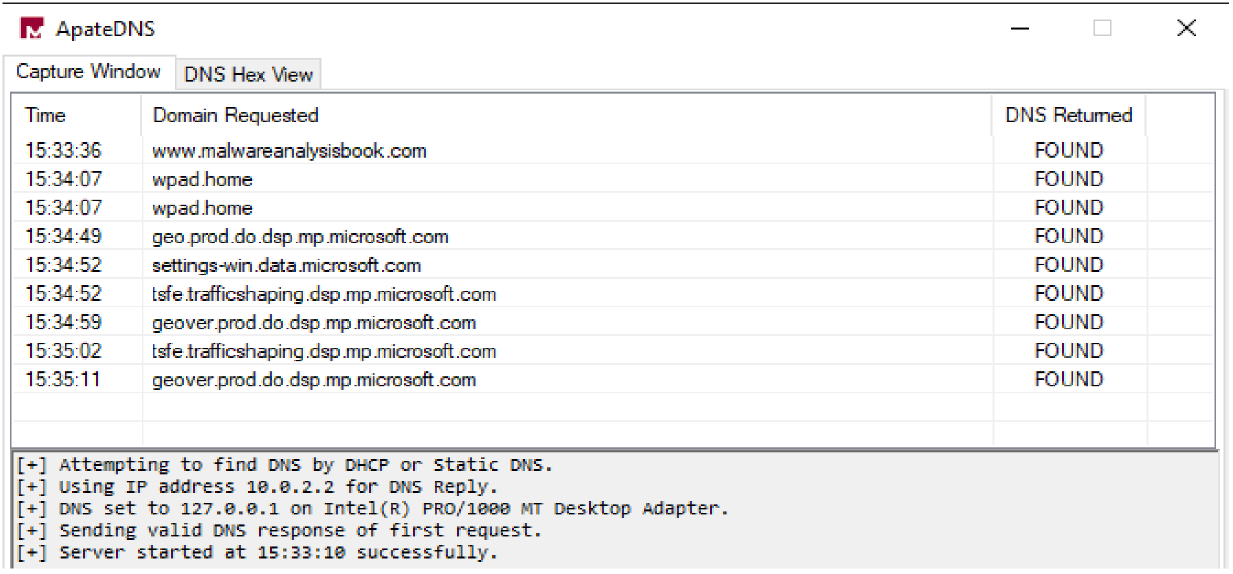
Coming back to main, after this, there is the call to StartServiceCtrlDispatcherA that will execute the service set in the table, executed by calling process (the one set at 401040). The main function will return the output value of 401040, or better return 0 as it returns 0 after all the actions described previously.



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  SERVICE_TABLE_ENTRYA ServiceStartTable; // [esp+0h] [ebp-10h] BYREF
  int v5; // [esp+8h] [ebp-8h]
  int v6; // [esp+Ch] [ebp-4h]

  ServiceStartTable.lpServiceName = aMalservice;
  ServiceStartTable.lpServiceProc = (LPSERVICE_MAIN_FUNCTIONA)sub_401040;
  v5 = 0;
  v6 = 0;
  StartServiceCtrlDispatcherA(&ServiceStartTable);
  return sub_401040(0, 0);
}
```

Figure 6: decompiled code for function main

Here again I can confirm this using ApateDNS and checking the connections. As in the previous sample, there is not only the real connection (in this case to “http://malwareanalysisbook.com”) but also to other sites that googling them tourns out that are proxy connections, diagnostic data sent to Microsoft, connection to Windows updates as for example to get Pro version.



Time	Domain Requested	DNS Returned
15:33:36	www.malwareanalysisbook.com	FOUND
15:34:07	wpad.home	FOUND
15:34:07	wpad.home	FOUND
15:34:49	geo.prod.do.dsp.mp.microsoft.com	FOUND
15:34:52	settings-win.data.microsoft.com	FOUND
15:34:52	tsfe.trafficshaping.dsp.mp.microsoft.com	FOUND
15:34:59	geover.prod.do.dsp.mp.microsoft.com	FOUND
15:35:02	tsfe.trafficshaping.dsp.mp.microsoft.com	FOUND
15:35:11	geover.prod.do.dsp.mp.microsoft.com	FOUND

```
[+] Attempting to find DNS by DHCP or Static DNS.
[+] Using IP address 10.0.2.2 for DNS Reply.
[+] DNS set to 127.0.0.1 on Intel(R) PRO/1000 MT Desktop Adapter.
[+] Sending valid DNS response of first request.
[+] Server started at 15:33:10 successfully.
```

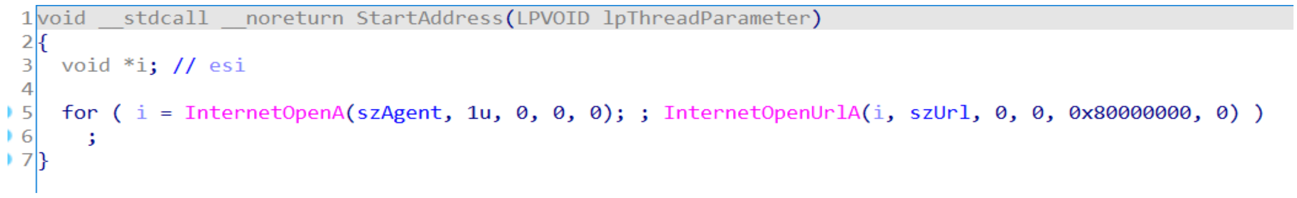
Figure 7: output of ApateDNS

An interesting thing is that when I execute the program, in the command line there is no output and it remains opened as waiting for some input, but if you try to write something it do not get it. With the previous knowledge I can say that it is maybe waiting for the specified time.

To confirm this, I made some debugging and I discovered that when makes the connection inside function 401040 and when calls StartServiceCtrlDispatcherA inside main, there is like a block in the debugging: makes connections to some dlls KernelBase.dll, ntdll.dll (run only if, when asked, you allow to execute application’s exception handler if one) and to RPCRT4.dll (run if you do not allow to execute the exception handler). Above all it goes stuck in ntdll.dll:77814E91 where makes some download and waits for something (I do not think is the wait object that I found in the function 401040, but it is like a wait for some connection and creates also some threads).

To understand the flow, I can force the execution of some parts setting and jumping to that specific IP. Doing so I understood the time to wait for (Thursday 7 June 5506 15:17:19), the time to sleep (49 days) and the threads created (20 before calling

Sleep function, whose name changed for different trials of debugging). What I could not know is the output of function StartAddress and how exactly it works, because I was not able to debug it as it is the input of function CreateThread and if I make a step into, it leads me to Kernel.dll. The second thing I was not able to analyse due to lack of time (49 days) is what happens after the Sleep but analysing the assembly, it ends the procedure. So, a question arises: how does it wait for that date if the process ended? What does it do when that date is reached? Statically I have no answers for it and dynamically I have no time to test it.



```
void __stdcall __noreturn StartAddress(LPVOID lpThreadParameter)
{
  void *i; // esi

  for ( i = InternetOpenA(szAgent, 1u, 0, 0, 0); ; InternetOpenUrlA(i, szUrl, 0, 0, 0x80000000, 0) )
    ;
}
```

Figure 8: decompiled code for function StartAddress, one of the input parameters to CreateThread API

Briefly, this program creates 20 threads I think to Internet Explorer connecting to that specific webpage “http://malwareanalysisbook.com” while is waiting to date “Thursday 7 June 5506 15:17:19” and sleep for 49 days. Even if I understood how it works, I am not able to classify it but using virustotal I know it is a trojan.

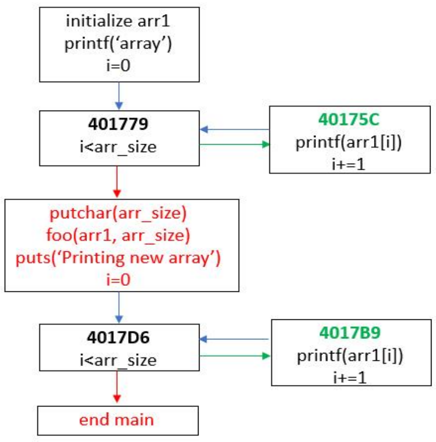
Coming back to the main question of this task (windows API), here we have a creation/opening of mutex “HGL345”; creation of 20 different threads (for 2 times as function 401040 is called to times in the main, one at line 8 and one at line 12) but also other threads as I saw in the debugger when called dlls; a service called MalService located in the path of the running file, which is also the name of the SCM service process to which connect the main thread; a network communication in function StartAddress to webpage “http://malwareanalysisbook.com”.

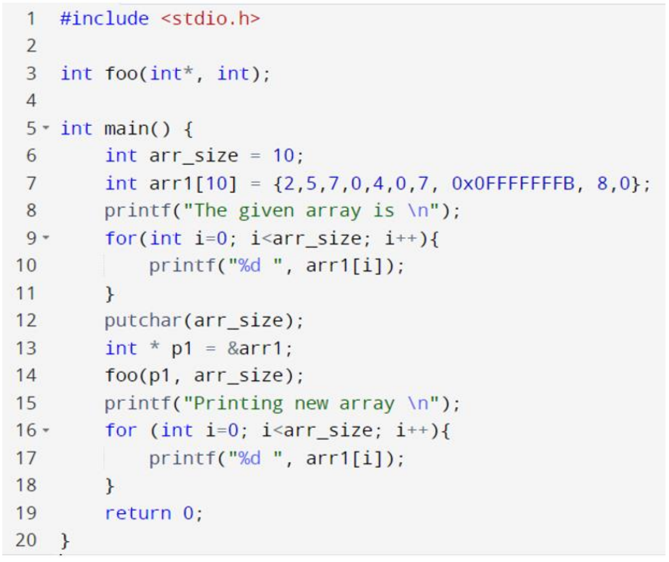
Task 3 – More Static Analysis

In this task I have to analyse some assembly code of two different subroutines.

In the first part of main function (subroutine 004016E0) I can see the stack preparation and the initialization of some parameters and the 10 values of arr1 with also its size, then a printf with message for the array, the initialization of index i to zero (i=0) and then immediately jumps to 401779.

- 401779 compares i and arr_size, so checks if i<arr_size and if yes, jump to 40175C; otherwise calls for function foo passing as input arr1 and arr_size (before calls putchar for this last parameter). Then calls puts with the message to print the new array, initializes index i to 0 (i=0) and jumps to 4017D6.
- 40175C makes a printf of the i-th value of the array (printf("%d", arr1[i])) and increments the index of 1 each time (i+=1), jumping back to 401779 to check if the inequality is still satisfied and make another iteration.
- 4017D6 where checks if i<arr_size, if yes jump to 4017B9, otherwise end the procedure.
- 4017B9 prints the array (printf("%d", arr1[i])), which should be the output of the function foo that received it as input; index i is incremented by 1 to print the next value if the inequality at next cycle is satisfied (so come back to 4017D6 to verify it).



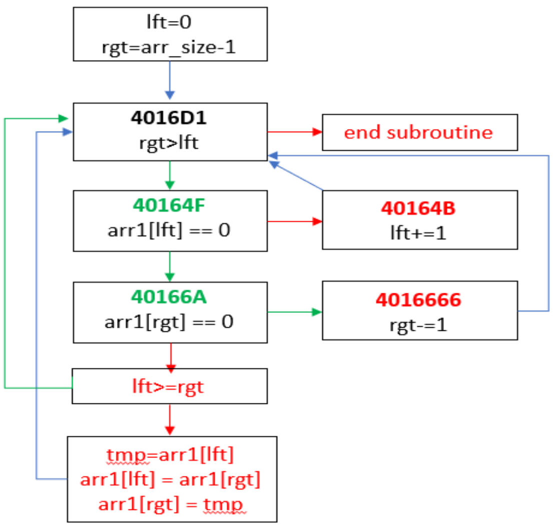


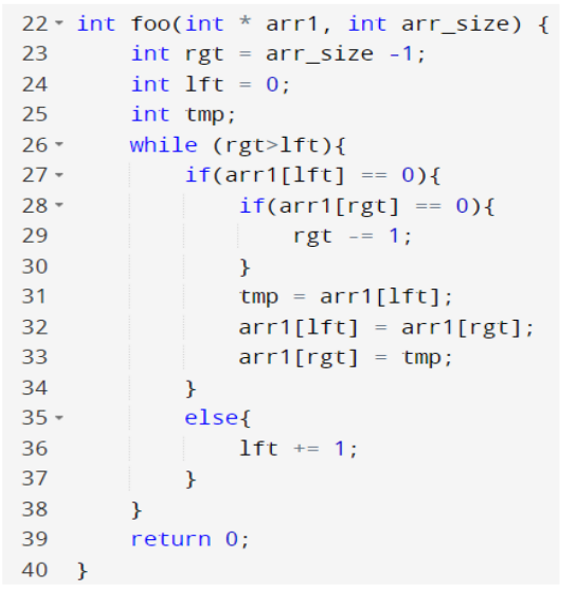
```c
#include <stdio.h>

int foo(int*, int);

int main() {
    int arr_size = 10;
    int arr1[10] = {2,5,7,0,4,0,7, 0x0FFFFFFFB, 8,0};
    printf("The given array is \n");
    for(int i=0; i<arr_size; i++){
        printf("%d ", arr1[i]);
    }
    putchar(arr_size);
    int * p1 = &arr1;
    foo(p1, arr_size);
    printf("Printing new array \n");
    for (int i=0; i<arr_size; i++){
        printf("%d ", arr1[i]);
    }
    return 0;
}
```

Figure 9: on the left there is the block representation (according to IDA standard) of main function, on the right the C code

After that, I analyse locations for procedure foo: first of all, the stack is prepared, variables are initialized lft=0 and rgt=arr_size-1, after that jumps immediately to 4016D1.

- 4016D1 checks if rgt>lft and if yes jumps to 40164F, otherwise ends the subroutine. When ending, the array, thanks to pointers, is passed to main function.
- 40164F checks if arr1[lft] is equal to 0, if not will jump to 40164B, otherwise jump to 4016AA.
- 40164B will increment of 1 lft (lft+=1) and comes back to 4016D1 to check if the inequality is still satisfied and continues looping.
- 4016AA checks if arr1[rgt] is equal to 0 and if yes will jump to 401666, otherwise checks if lft>=rgt. If lft>=rgt will jump to 4016D1 where will check if rgt>lft, but this is not satisfied so will end. If lft>=rgt is not satisfied, the function will jump to the last block where will switch the values between arr1[lft] and arr1[rgt] using a tmp variable. After this switch, will immediately jump to 4016D1 to know if it must continue or end.
- 401666 decrements by 1 the value of index rgt and after that will jump immediately back to 4016D1.

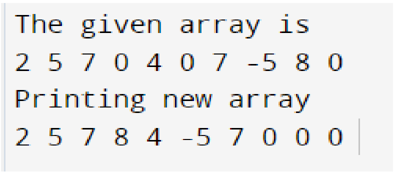




```
int foo(int * arr1, int arr_size) {
    int rgt = arr_size -1;
    int lft = 0;
    int tmp;
    while (rgt>lft){
        if(arr1[lft] == 0){
            if(arr1[rgt] == 0){
                rgt -= 1;
            }
            tmp = arr1[lft];
            arr1[lft] = arr1[rgt];
            arr1[rgt] = tmp;
        }
        else{
            lft += 1;
        }
    }
    return 0;
}
```

Figure 10: on the left there is the block view (following IDA standard for graph view) of function foo, while on the right the C code

This program makes a sort of the array, above all puts all the 0 values of the array at the end.



```
The given array is
2 5 7 0 4 0 7 -5 8 0
Printing new array
2 5 7 8 4 -5 7 0 0 0
```

Figure 11: printf of this code, first the given array, then the sorted array with the zeros at the end