SecurityXploded

Reversing and Malware Analysis Training Articles [2012]

A Free Training Project From SecurityXploded
(http://securityxploded.com/security-training.php)

SecurityXploded Research Group
11/24/2012

Contents

DISCLAIMER, ACKNOWLEDGMENT AND CREDITS **5**

Disclaimer 5
Acknowledgment 5
Credits 5

ASSEMBLY PROGRAMMING: A BEGINNERS GUIDE **6**

Introduction 6
Required Tools 6
Installation 6
Configuring WinAsm 7
Programming in ASM using MASM & WinAsm 7
My First ASM Program 8
Block 1 .... *9*
Block 2 .... *9*
Block 3 .... *9*
Block 4 .... *10*
Block 5 .... *10*
Build and Run the Program *10*
References 11

AUTOMATION OF REVERSING THROUGH SCRIPTING **12**

Introduction 12
OllyDbg - Playing with OllyScript 13
Problem Statement: *13*
Solution: *13*
Immunity Debugger 15
Problem statement: *15*
Solution Script: *15*
PyDbg 16
Problem Statement: *16*
Solution: *16*
WinDbg 18
Problem Statement: *18*
Solution: *18*
Conclusion 19
References 19

API CALL TRACING - PEFILE, PYDBG AND IDAPYTHON **20**

Introduction 20
API Call Tracing 20
API Calls Logging with PEfile & PyDbg 21
Unpacking UPX using API Call Tracing *25*
Binary Behaviour Profiling *26*
Finding Interesting Functions *26*
Extending API Tracing with IDAPython 27
Conclusion 28
References 28

MANUAL UNPACKING OF UPX USING OLLYDBG **29**
INTRODUCTION 29
PACKING EXE USING UPX 29
UPX UNPACKING PROCESS 30
MANUAL UNPACKING OF UPX 30
FIXING IMPORT TABLE 31
VIDEO DEMONSTRATION 33
REFERENCES 33
MALWARE MEMORY FORENSICS **34**
INTRODUCTION 34
WHY MEMORY FORENSICS? 34
VOLATILITY - A QUICK OVERVIEW 35
Volatility Syntax & Usage 35
DEMONSTRATION - MEMORY FORENSICS 35
Demo Scenario 35
Preparation Steps 35
DEMONSTRATION - MEMORY ANALYSIS 35
Step 1: Start with what you know 35
Step 2: Info about 208.91.197.54 36
Step 3: Who is Pid 1748? 36
Step 4: Process handles of explorer.exe 37
Step 5: API Hooks in explorer.exe 37
Step 6: Exploring the Hooks 37
Step 7: Embedded EXE in explorer.exe 38
Step 8: Dumping the embedded EXE 38
Step 9: VirusTotal Submission 39
Step 10: Can we get more info? 39
Step 11: Printing the Registry Key 40
Step 12: Finding the Malicious EXE on Infected Machine 40
CONCLUSION 41
REFERENCES 41
DLL INJECTION AND HOOKING **42**
INTRODUCTION 42
DLL INJECTION 42
DLL INJECTION USING CREATEREMOTETHREAD 43
HOOKING 47
INLINE HOOKING 48
CONCLUSION 52
REFERENCES 52
IN-MEMORY EXECUTION OF AN EXECUTABLE **53**
INTRODUCTION 53
TECHNICAL INTRODUCTION 53
IN-MEMORY EXECUTION 54
Address space 55
Sections mapped into Address Space 56
Imported API addresses 56

MEMORY EXECUTION – PROTOTYPE CODE 56
CONCLUSION..... 60
REFERENCES..... 60

Disclaimer, Acknowledgment and Credits

Disclaimer

The Content, Demonstration, Source Code and Programs presented here is "AS IS" without any warranty or conditions of any kind. Also the views/ideas/knowledge expressed here are solely of the trainer's only and nothing to do with the company or the organization in which the trainer is currently working.

However in no circumstances neither the trainer nor SecurityXploded is responsible for any damage or loss caused due to use or misuse of the information presented here.

Acknowledgment

Special thanks to **null** & **Garage4Hackers** community for their extended support and cooperation.

Thanks to all the trainers who have devoted their precious time and countless hours to make it happen.

Thanks to **Keon** and **Thoughtworks** for providing beautiful venue.

Credits

Following people served as the backbone of this project, without their efforts it was not possible to make it happen.

Monnappa	:	Infosec investigator, Cisco Inc.
Swapnil Pathak	:	Security Researcher, McAfee Inc.
Harsimran Walia	:	Security Researcher, McAfee Inc.
Nagareshwar Talekar	:	Security Researcher and Founder of SecurityXploded
Amit Malik	:	Security Researcher, McAfee Inc.

The trainers would also like to thank their employer and seniors for allowing them to deliver these lectures.

Assembly Programming: A Beginners Guide

Author: Amit Malik

Introduction

This article is specially designed to help beginners to understand and develop their **first Assembly Program** from scratch. Through step by step instructions it will help you to use tools, setup the environment and then build sample **'Hello World'** program in Assembly language with detailed explaination.

This article is the part of our free **"Reverse Engineering & Malware Analysis Course"** [Reference 4]. It is written as pre-learning guide for our session on **'Part 4 - Assembly Programming Basics'** where in we are going to cover Assembly Programming from the reverse engineering perspective.

Here we will be demonstrating Assembly programming using **MASM** as it is the Microsoft assembler and provide much flexibility when it comes to development on **Windows environment** over various other assemblers like NASM etc.

Required Tools

- **MASM** [Reference 2] - MASM is a Microsoft assembler.
- **WinAsm** [Reference 3] - WinAsm is IDE. It provides a nice interface for coding and moreover you don't have to type different-2 command for assembler and linker to compile a binary, with one click it will generate EXE for you.

Installation

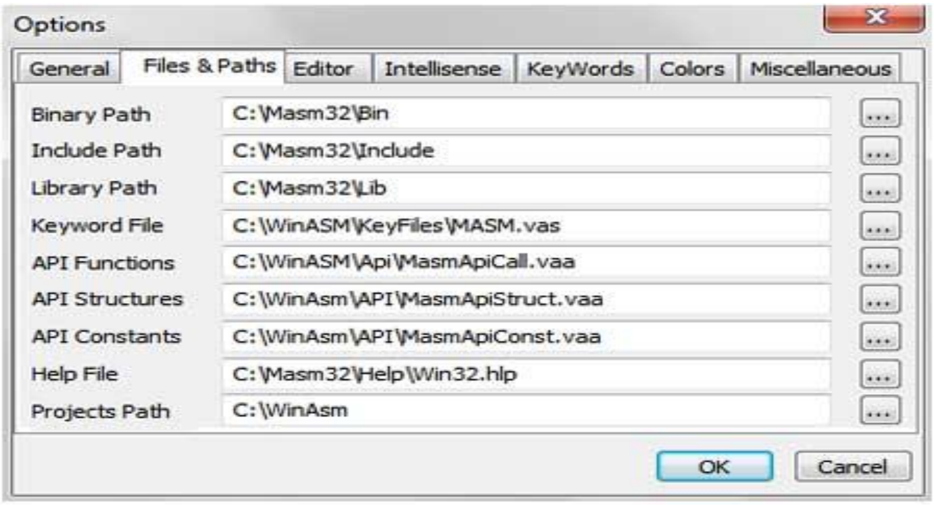
- **MASM** - By default MASM tries to install itself in windows drive mostly c drive but you can install it in any Drive/directory. We need the full path of MASM installation to configure WinAsm so note down the drive/directory where you installed MASM.

- **WinAsm** - Download and extract the WinAsm package. WinAsm comes with all files you require so you don't have to install it. Just copy the folder to "c:\program files\" and make a shortcut to desktop so that you can access directly from desktop.

Configuring WinAsm

Launch WinAsm by double clicking on the shortcut created on the desktop. In order to integrate it with MASM we need to setup the MASM path in WinAsm configurations. Here are the steps,

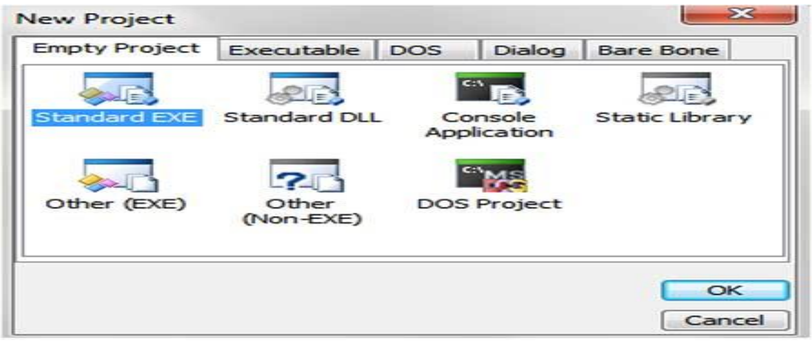
1. Click on the tools tab
2. In tools click on options
3. In options click on file & path tab
4. Change the all entries with path to MASM installation folder
5. Click on Ok.



After this you should be able to write programs in WinAsm.

Programming in ASM using MASM & WinAsm

Launch the WinAsm window, click on the "file" tab. Then click on the new projects and it will show you couple of options as shown below.



- **Console Application** - For creating console (command-line) applications
- **Standard EXE** - For creating GUI based applications

Here we willl use **Standard EXE** because we want to make a GUI Application. Now you will see the editor window in which you can write your programs.

My First ASM Program

Here is a typical assembly program structure,

1. **Architecture** - Define the architecture because assembly is Hardware (processor) dependent language so you have to tell to assembler the architecture for which you are writing your program.
2. **Data Section** - All your initialized and uninitialized variables reside in data section.
3. **Code Section** - Entire code of your program reside in this section.

Now we will write a program that will display the message box saying **"Hello World!"**

```
;------------Block 1----------
.386
.model flat,stdcall
option casemap:none

;------------Block 2----------
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib

;------------block 3----------
.data
szCaption db "Hello",0
szMsg db "Hello World!",0

;------------Block 4----------
.data?
retvalue dd ?

;------------Block 5----------
.code
start:
invoke MessageBox,NULL,addr szMsg,addr szCaption,MB_OK
mov retvalue,eax
xor eax,eax
invoke ExitProcess,eax
end start
```

I divided the above code in 5 blocks. Below I will explain the purpose and functionality of each block.

Block 1

```
1).386
2).model flat,stdcall
3)option casemap:none
```

#1 - This line defines the architecture for which we want to make this program. (.386) represent Intel architecture
#2 - This line defines the model and the calling convention that we want to use for this program. We will explain it in detail in our "Assembly Basics" session.
#3 - function names, variable names etc. are case sensitive

All these three lines are required in each program.

Block 2

```
1)include windows.inc
2)include user32.inc
3)includelib user32.lib
4)include kernel32.inc
5)includelib kernel32.lib
```

include and **includelib** are two keywords. Include is used with .inc files while includelib is used with .lib files.

.inc files are header files. for eg: windows.inc is windows.h, you can convert any .h file into .inc file using H2INC utility that comes with MASM.

.lib files are required by linker to link the used functions with the system dlls. In our program we used two .lib files (user32.lib & kernel32.lib). For each .lib file we have to include its corresponding .inc file.

Block 3

```
1).data
2)szCaption db "Hello",0
3)szMsg db "Hello World!",0
```

.data is the section for initialized variables. Every initialized variable should be initialized in this section. In our code we have two variables of char type <**string**>.

```
Syntax: <variable_name> <type> <value>
```

For eg: in #2 **szCaption** is the variable name, db is the type means char type, "Hello", 0 is the value.

Here important point to note is that every char or string value should be terminated with zero (0).

Block 4

```
1).data?
2)retvalue dd ?
```

.data? is the section for uninitialized variables. Every uninitialized variable should be declared in this section.

Block 5

```
1).code
2)start:
3)invoke MessageBox,NULL,addr szMsg,addr szCaption,MB_OK
4)mov retvalue,eax
5)xor eax,eax
6)invoke ExitProcess,eax
7)end start
```

.code represents the start of code. All your code should be written in this section

#2 start: It is a label and it is like main function. You can name it anything but you have to use the same name in #7 otherwise linker will generate an error.

Fore.g.:
main:
...
end main

#3 **invoke** - is the keyword, its operation is similar to "call". But in call you have to manually push parameters on the stack while invoke will do everything for you.

```
Syntax: function_name parameter1, parameter2, parameter3, etc.
```

In our code **MessageBox** is the API from user32.dll and it requires 4 arguments.

Here important point to note is that we used "addr" with some of our variables. addr will give address of the variable instead of its value, it is like pointer in c.

#7 end start - it says the end of the code and file.

Build and Run the Program

Now paste the above code in **WinAsm** and click on "make" tab, in "make" click on "Assemble". After that click on "link" which will be the executable for this program.

Finally run the EXE file by double clicking on it, it should display "Hello World!".

This is a basic program to help you to learn Assembly Language in easier way. For more advanced details refer/attend our **FREE Reversing/Malware Analysis** course [Reference 4]

References

1. Icezelion's Win32 Assembly Tutorials
2. MASM - http://www.masm32.com/
3. WinASM - http://www.winasm.net/
4. Reverse Engineering & Malware Analysis Course

Automation of Reversing Through Scripting

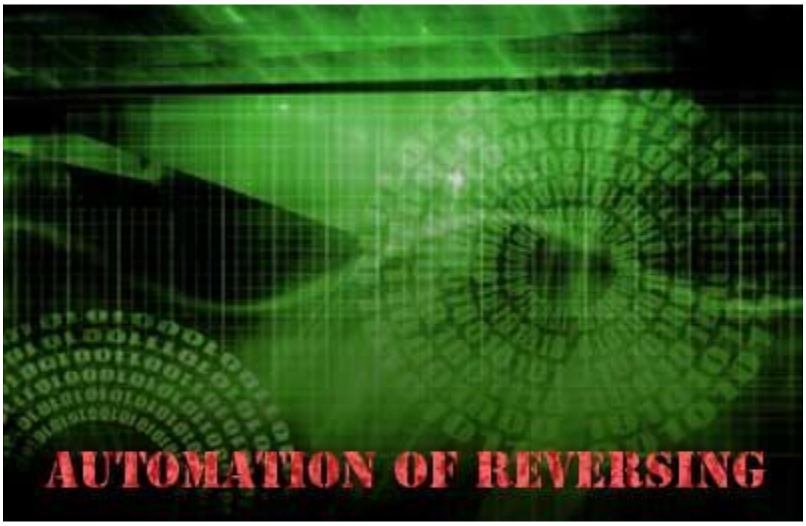
Author: Amit Malik

Introduction

This article teaches you how to become smart reverser by **automating your reverse engineering** tasks through **Scripting**.

It is the part of our free **"Reverse Engineering & Malware Analysis Course"** [Reference 1]. It is primarily written to act as additional learning material for our session on 'Part 5 - Reverse Engineering Tools' where in we are going to demonstrate important reversing tools.

You can visit our training page here [Reference 1] and all the presentations of previous sessions here [Reference 2]



Reverse engineering is a sophisticated task especially when we analyse large applications or packed files like **malware** or normal applications for vulnerabilities.

Some of the common tasks include

- Tracking memory allocation
- Tracking specific API calls
- Unpacking a family of malwares
- Intelligent decision making based on some specific events

These are just some simple examples where automation will help in a great way. For example, lets say that we want to monitor**HeapAlloc** calls in an application and application may call HeapAlloc for hundreds of times but we want to log the call for some specific values like if allocation request is greater than 1024 bytes etc.

A simple script will give us all the information virtually on the spot while in manual task we have to manually create breakpoints on HeapAlloc and have to check if the allocation size is greater than 1024 bytes or not which eventually increase the analysis time for such a simple task.

In this article, I will show you how to automate some of these common tasks **through Scripting** for main **reversing debuggers** i.e Ollydbg, Immunity Debugger, Pydbg & Windbg with practical code samples.

Ollydbg - Playing with OllyScript

Ollydbg [Reference 3] is one of the best ring 3 (user-land) debugger. It has a very nice gui interface. It is one of the most popular debugger on the planet and has very mature community support. Ollydbg is my all time favourite debugger :)

But ollydbg doesn't support scripting natively instead ollydbg support plugins. So people written scripting plugins for ollydbg, the one that i will use in this article is Ollyscript by ShaG.

You can **download Ollyscript** from here [Reference 4].

Ollyscript comes with a nice help file. It has similar syntax like assembly programming and very easy to understand. It supports almost all functionalities like dumping memory, decision making etc.

But when you compare it with other debuggers scripting environment then it will seems to be a rigid type of scripting environment, I will discuss more about it later in this article.

So let's understand Ollydbg scripting environment i.e Ollyscript with the help of a simple example.

Problem Statement:

Let say we are analysing an application for a simple bug and we want to identify the function that is actually causing the problem. But the function is deep inside the application and manually it will take hours of analysis time.

So here we want to track the execution flow after a specific point up to the function that is causing the problem, more precisely I want to log the return address of each function.

Solution:

The above problem can be solved by multiple methods but to demonstrate it in a very simple way I will use the following steps,

1. From current EIP, search for calls and create breakpoint on that call
2. Step into the call

3. Log the value at ESP (i.e return address) and search for calls at return address and
4. Breakpoint on the call
5. Repeat step 1, 2, 3 inside the call
6. Run

Below is the tiny script to accomplish this task. Please note that the script is just to demonstrate the concept, it may fail when call used after decision instructions.

```
/*
Author: Amit Malik
http://www.securityxploded.com
*/

EOB breakprocess
var return
var infunction
var x
var y
mov infunction,EIP
mov return,EIP

start:
findop return,#E8#
mov x,$RESULT
findop infunction,#E8#
mov y,$RESULT
cmp x,0
ja breaksetx
backx:
cmp y,0
ja breaksety
backy:
run

breakprocess:
sti
mov return,[esp]
msg return
sti
mov infunction,EIP
jmp start

breaksetx:
bp x
jmp backx
```

```
breaksety:
bp y
jmp backy
```

Please refer to the Ollyscript help file [Reference 4] for more details. Here I will explain only important keywords and terms.

The script start with **EOB (Execute over breakpoint)**, as name states it will execute the code inside the label that is specified with EOB when a breakpoint hit. In this code it will execute the breakprocess label code.

```
var - declares a variable.
mov - is similar to assembly
findop - search for opcode from the specified address & stores the results
into a $RESULT variable
run - is similar to F9 in ollydbg
sti - step into - similar to F7 in ollydbg
msg - will show a messagebox - (log should be used but I used msg just for
visual pleasure :))
```

As you can see that scripting is similar to assembly language. Most of the time people use ollyscripting for unpacking malwares. I have never seen anyone using it for vulnerability analysis. It is not very much flexible and also limited in its functionality. But it can be used for some stuff that we want to automate through ollydbg.

Immunity Debugger

Immunity debugger [Reference 3] is a pure python debugger with similar GUI interface as Ollydbg. It is developed by Immunity Inc. and according to immunity it is the only debugger designed specifically for **vulnerability research**.

It has some very powerful pycommands like heap, lookasidelist etc. one of the major advantage of this debugger is that it provides plethora of APIs for various reversing tasks and supports python which makes it one of the best debugger for reversing.

In the reference section [Reference 6] you can find some good tutorials and projects based on Immunity debuggers and also it comes with a nice help file so don't forget to check that as well.

Problem statement:

We want to search all "jmp esp" instruction addresses.

Solution Script:

You can use the below script directly on Immunity debugger python shell

```
data = "jmp esp"
asm = imm.assemble(data)    # imm is object of immlib class
results = imm.search(asm)

for addr in results:
        print "%s  %0.8x" % (data,addr)
```

The above 5 lines of code will give you all the "jmp esp" addresses. This is the beauty of scripting :)

Pydbg

Pydbg [Reference 3] is also a pure python based debugger. Pydbg is my favourite debugger, I use it in various automation tasks and it is extremely flexible and powerful.

Problem Statement:

We want to track **VirtualAlloc** API whenever VirtualAlloc is called, our script should display its arguments and the returned pointer.

```
VirtualAlloc:

LPVOID WINAPI VirtualAlloc(
__in_opt LPVOID lpAddress,
__in SIZE_T dwSize,
__in DWORD flAllocationType,
__in DWORD flProtect
);
```

Solution:

1. Put breakpoint on VirtualAlloc
2. Extract parameters from stack
3. Extract return address from stack and put breakpoint on that
4. Get the value from EAX register.

```
# Author: Amit Malik
# http://www.securityxploded.com
import sys
import pefile
import struct
from pydbg import *
from pydbg.defines import *
def ret_addr_handler(dbg):

    lpAddress = dbg.context.Eax    # Get value returned by VirtualAlloc
    print " Returned Pointer: ",hex(int(lpAddress))
```

```
    return DBG_CONTINUE
def virtual_handler(dbg):

    print "****************"
    pdwSize = dbg.context.Esp + 8       # 2nd argument to VirtualAlloc
    rdwSize = dbg.read_process_memory(pdwSize,4)
    dwSize  = struct.unpack("L",rdwSize)[0]
    dwSize  = int(dwSize)
    print "Allocation Size: ",hex(dwSize)
    pflAllocationType = dbg.context.Esp + 12     # 3rd argument to
VirtualAlloc
    rflAllocationType = dbg.read_process_memory(pflAllocationType,4)
    flAllocationType  = struct.unpack("L",rflAllocationType)[0]
    flAllocationType  = int(flAllocationType)
    print "Allocation Type: ",hex(flAllocationType)

    pflProtect = dbg.context.Esp + 16          # 4th Argument to
VirtualAlloc
    rflProtect = dbg.read_process_memory(pflProtect,4)
    flProtect  = struct.unpack("L",rflProtect)[0]
    flProtect  = int(flProtect)
    print "Protection Type: ",hex(flProtect)
    pret_addr = dbg.context.Esp                         # Get return
Address
    rret_addr = dbg.read_process_memory(pret_addr,4)
    ret_addr  = struct.unpack("L",rret_addr)[0]
    ret_addr  = int(ret_addr)
    dbg.bp_set(ret_addr,description="ret_addr breakpoint",restore =
True,handler = ret_addr_handler)

    return DBG_CONTINUE

def entry_handler(dbg):

    virtual_addr = dbg.func_resolve("kernel32.dll","VirtualAlloc")    #
Get VirtualAlloc address
    if virtual_addr:
            dbg.bp_set(virtual_addr,description="Virtualalloc
breakpoint",restore = True,handler = virtual_handler)

    return DBG_CONTINUE

def main():

    file = sys.argv[1]
    pe = pefile.PE(file)
    # get entry point
```

```
    entry_addr = pe.OPTIONAL_HEADER.AddressOfEntryPoint +
pe.OPTIONAL_HEADER.ImageBase
    dbg = pydbg()           # get pydbg object
    dbg.load(file)
    dbg.bp_set(entry_addr,description="Entry point breakpoint",restore
= True,handler = entry_handler)
    dbg.run()

if __name__ == '__main__':
    main()
```

Notice that in this script first i am setting breakpoint on entry point and then on VirtualAlloc not directly to VirtualAlloc because pydbg does not support deferred breakpoints. I am also ignoring 1st argument to VirtualAlloc i.e lpAddress, see VirtualAlloc specification in problem statement.

This script uses two modules **PEFile** and **Pydbg**, PEFile is used to get the entry point.

Windbg

Windbg [Reference 3] is the official Microsoft debugger. It is the most powerful debugger available for reversing on windows platform (mainly Kernel side of it) and it also supports symbols.

Windbg provides its own scripting language which is similar to C language, it also comes with a great help file. I highly recommend reading help file before we start with Windbg.

Problem Statement:

We want to track malloc, whenever malloc is called, our script should display requested size for allocation and returned pointer.

Solution:

On the same lines as previous example.

1. Breakpoint on malloc
2. Extract parameter from stack
3. Extract return address from stack and put breakpoint on it
4. Get value from EAX register

```
bp msvcrt!malloc ".printf \"Size: %x\n\",poi(esp+4);gu;.printf \"Returned
Pointer: %x\n\",eax;g"
```

When we use multiple commands in a single line then we have to separate them using semicolon (;)

```
bp - sets breakpoint
msvcrt!malloc - this is DLL!Method (here DLL name & function name are
separated by ! )
```

These are known as conditional breakpoints and in conditional breakpoints we want to perform something when breakpoint hit. In our case we want extract the size of allocation from stack.

So simple syntax is:

```
bp address or dll!method or dll!method+offset "block that should be
executed when breakpoint hits"
```

```
poi - is similar to pointer in c
gu - go up - execute until return
g - go or execute
```

For more interesting commands please check out the **Windbg help** file.

Conclusion

This article is an additional learning material to our next session on 'Part 5 - Reverse Engineering Tools' - part of our **FREE Reversing/Malware Analysis** course [Reference 1]

References

1. Reverse Engineering & Malware Analysis Course
2. Presentations of Reverse Engineering Course
3. Debuggers - OllyDbg, Immunity Debugger, PyDbg, Windbg
4. OllyScript - Scripting Plugin for OllyDbg
5. WinDbg Introduction
6. Starting to write Immunity Debugger PyCommands : My Cheatsheet
7. mona.py – the manual

API Call Tracing - PEfile, PyDbg and IDAPython

Author: Amit Malik

Introduction

In this article, we will learn how to perfrom **API Call Tracing** of Binary file through PyDbg and IDAPython.

This is the part of our free **"Reverse Engineering & Malware Analysis Course"**.

You can visit our training page **here** and all the presentations of previous sessions **here**

In my previous article, **"Automation of Reversing"** I have discussed on using PyDbg scripting environment. Here also we are going to use PyDbg extensively to trace or log the API calls from a binary file.

API Call Tracing

API Call Tracing is the powerful technique. It can provide a high level functional overview about an executable file. In some cases we only need API call logs to understand the application behaviour. I often use it to automate my Malware analysis tasks.

In this article I will discuss some of my techniques.

Some of the tasks that we can accelerate using this technique are,

1. Unpacking of Packed Binary File
2. Binary Behaviour profiling
3. Finding out the interesting functions in the binary

Here, I will use **PyDbg script** to log the API calls and finally **IDAPython** script to automate some of manual analysis.

API Calls Logging with PEfile & PyDbg

Based on the above tasks we need following information from our script.

1. Return Address - From where the API is called?
2. API Name - Which API is called?

It means we have to **breakpoint on every API call** and for that we need API name or API address. If we have API name then we can resolve its address and can breakpoint on that, In case of address we can directly breakpoint on that. But the question is how do we get the API names?

This can be solved by using **PEfile**. So we will first enumerate the executable import table and then we will resolve the addresses and put breakpoints using PyDbg.

But this approach has following limitations,

1. It will fail in the case of a DLL that will be loaded by binary at run time using LoadLibrary()
2. If binary is packed then unpacking stub will create the import table at run time which we can't control.

Before solving this problem let's talk about the ways used by unpacker stub or custom loaders to build an import table at run time.

Generally they use **LoadLibrary** API to load the dll and **GetProcAddress** to get the address of the API. LoadLibrary and GetProcAddress APIs are exported by kernel32.dll which is loaded into every Windows process by default.

So if we set breakpoint on GetProcAddress then we can get API Name from stack. Then we can set breakpoint on the address of respective API call. Here I am ignoring the call for GetProcAddress with API Ordinal because it is not a common approach.

But there is also another method for building import table at run time which is typically used by **malicious softwares**.

In assembly it will look like this,

```
push dword ptr fs:[30h] ; PEB
pop eax
```

```
mov eax,[eax+0ch] ; LDR
mov ecx,[eax+0ch] ; InLoadOrderModuleList
mov edx,[ecx]
push edx
mov eax,[ecx+30h]
```

Here is the screenshot of **PEB structure** of typical Windows Process (dumped in Windbg)

```
0:000> dt nt!_PEB -r2
ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged        : UChar
   +0x003 SpareBool            : UChar
   +0x004 Mutant               : Ptr32 Void
   +0x008 ImageBaseAddress     : Ptr32 Void
   +0x00c Ldr                  : Ptr32 _PEB_LDR_DATA
      +0x000 Length               : Uint4B
      +0x004 Initialized          : UChar
      +0x008 SsHandle             : Ptr32 Void
      +0x00c InLoadOrderModuleList : _LIST_ENTRY
         +0x000 Flink               : Ptr32 _LIST_ENTRY
         +0x004 Blink               : Ptr32 _LIST_ENTRY
      +0x014 InMemoryOrderModuleList : _LIST_ENTRY
         +0x000 Flink               : Ptr32 _LIST_ENTRY
         +0x004 Blink               : Ptr32 _LIST_ENTRY
```

In this method, custom loader first locate the **kernel32.dll** base address (2nd - after ntdll.dll in InLoadOrderModuleList link list] and then walk through the kernel32.dll export table to find out the LoadLibrary() address. After that custom loader will load all other dependent dlls and resolve the API Addresses using the following methods,

1. GetProcAddress - similar to previous method
2. Walking through the export table of each loaded dll.

Here to capture the activity of #2 we have to use global hooks or SSDT hooks which is beyond the scope of this article. We can also hook every exported API of all loaded DLLs but that can be very expensive.

Here are the step by step instructions for **API Call Tracing**,

1. Walk through the binary import table and put breakpoint on every API
2. Also put Breakpoint on GetProcAddress function.
3. If Breakpoint hits and it is not GetProcAddress then extract 'Return Address' from stack and log it with API name
4. If GetProcAddress hits then fetch API name and return address from stack and put breakpoint on 'Return Address'
5. If 'Return Address' breakpoint hits then get value from EAX register and set breakpoint on it.

Based on this approach, we will write **PyDbg script** and log every API with **'Return Address'**

```
'''
Author: Amit Malik
```

```
http://www.securityxploded.com
'''

import sys,struct
import pefile
from pydbg import *
from pydbg.defines import *

def log(str):
        global fpp
        print str
        fpp.write(str)
        fpp.write("\n")
        return

def addr_handler(dbg):
        global func_name
        ret_addr = dbg.context.Eax
        if ret_addr:
                dict[ret_addr] = func_name
                dbg.bp_set(ret_addr,handler=generic)
        return DBG_CONTINUE

def generic(dbg):
        global func_name
        eip = dbg.context.Eip
        esp = dbg.context.Esp
        paddr = dbg.read_process_memory(esp,4)
        addr = struct.unpack("L",paddr)[0]
        addr = int(addr)
        if addr < 70000000:
                log("RETURN ADDRESS: 0x%.8x\tCALL: %s" % (addr,dict[eip]))
        if dict[eip] == "KERNEL32!GetProcAddress" or dict[eip] ==
"GetProcAddress":
                try:
                        esp = dbg.context.Esp
                        addr = esp + 0x8
                        size = 50
                        pstring = dbg.read_process_memory(addr,4)
                        pstring = struct.unpack("L",pstring)[0]
                        pstring = int(pstring)
                        if pstring > 500:
                                data = dbg.read_process_memory(pstring,size)
                                func_name = dbg.get_ascii_string(data)
                        else:
                                func_name = "Ordinal entry"
                        paddr = dbg.read_process_memory(esp,4)
                        addr = struct.unpack("L",paddr)[0]
```

```
                        addr = int(addr)
                        dbg.bp_set(addr,handler=addr_handler)
                except:
                        pass
        return DBG_CONTINUE


def entryhandler(dbg):
        getaddr = dbg.func_resolve("kernel32.dll","GetProcAddress")
        dict[getaddr] = "kernel32!GetProcAddress"
        dbg.bp_set(getaddr,handler=generic)
        for entry in pe.DIRECTORY_ENTRY_IMPORT:
                DllName = entry.dll
                for imp in entry.imports:
                        api = imp.name
                        address = dbg.func_resolve(DllName,api)
                        if address:
                                try:
                                        Dllname = DllName.split(".")[0]
                                        dll_func = Dllname + "!" + api
                                        dict[address] = dll_func
                                        dbg.bp_set(address,handler=generic)
                                except:
                                        pass

        return DBG_CONTINUE

def main():
        global pe, DllName, func_name,fpp
        global dict
        dict = {}
        file = sys.argv[1]
        fpp = open("calls_log.txt",'a')
        pe = pefile.PE(file)
        dbg = pydbg()
        dbg.load(file)
        entrypoint = pe.OPTIONAL_HEADER.ImageBase +
pe.OPTIONAL_HEADER.AddressOfEntryPoint
        dbg.bp_set(entrypoint,handler=entryhandler)
        dbg.run()
        fpp.close()

if __name__ == '__main__':
        main()
```

The output will look like,

```
RETURN ADDRESS: 0x004030e8     CALL: kernel32!GetModuleHandleA
```

```
RETURN ADDRESS: 0x004030f3     CALL: kernel32!GetCommandLineA
RETURN ADDRESS: 0x00404587     CALL: kernel32!GetModuleHandleA
RETURN ADDRESS: 0x00404594     CALL: kernel32!GetProcAddress
RETURN ADDRESS: 0x004045aa     CALL: kernel32!GetProcAddress
RETURN ADDRESS: 0x004045c0     CALL: kernel32!GetProcAddress
```

So let's apply the logic to some real world reverse engineering scenarios.

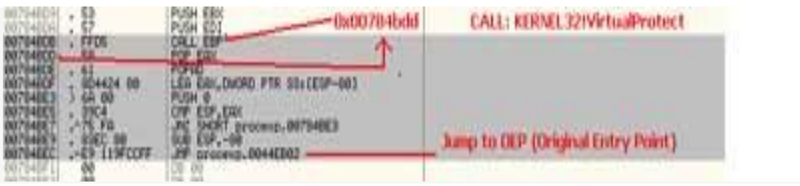
Unpacking UPX using API Call Tracing

Below is the log of a **UPX packed binary**. Look at it closely, can you say which function contains the OEP?

```
RETURN ADDRESS: 0x00784b9e     CALL: GetProcAddress
RETURN ADDRESS: 0x00784b9e     CALL: GetProcAddress
RETURN ADDRESS: 0x00784b9e     CALL: GetProcAddress
RETURN ADDRESS: 0x00784b9e     CALL: GetProcAddress
RETURN ADDRESS: 0x00784b9e     CALL: GetProcAddress
RETURN ADDRESS: 0x00784bc8     CALL: KERNEL32!VirtualProtect
RETURN ADDRESS: 0x00784bdd     CALL: KERNEL32!VirtualProtect          --> 1
RETURN ADDRESS: 0x0045ac09     CALL: GetSystemTimeAsFileTime  --> 2
RETURN ADDRESS: 0x0045ac15     CALL: GetCurrentProcessId
RETURN ADDRESS: 0x0045ac1d     CALL: GetCurrentThreadId
RETURN ADDRESS: 0x0045ac25     CALL: GetTickCount
RETURN ADDRESS: 0x0045ac31     CALL: QueryPerformanceCounter
RETURN ADDRESS: 0x0044e99f     CALL: GetStartupInfoA
RETURN ADDRESS: 0x0044fd9c     CALL: HeapCreate
```

Here API at location 1 has **'Return Address'** 0x00784bdd and API at location 2 has 'Return Address' 0x0045ac09. The difference between the addresses of both calls is huge which is an indication that the address 0x0045ac09 is in the function that contains OEP (original entry point).

This can be proved in the **Ollydbg** as shown in the below snapshot.



Most of the malwares these days have their own custom packers and I found this technique extremely useful in unpacking them.

Binary Behaviour Profiling

Look at the sample API Trace logs closely, Can you tell about the behaviour of this binary?

```
RETURN ADDRESS: 0x004012ce     CALL: msvcrt!fopen                                    --> 1
RETURN ADDRESS: 0x00401311     CALL: msvcrt!fseek
RETURN ADDRESS: 0x0040131c     CALL: msvcrt!ftell
RETURN ADDRESS: 0x0040133a     CALL: msvcrt!fseek
RETURN ADDRESS: 0x00401346     CALL: msvcrt!malloc                                  --> 2
RETURN ADDRESS: 0x00401387     CALL: msvcrt!fread                                   --> 3
RETURN ADDRESS: 0x00401392     CALL: msvcrt!fclose
RETURN ADDRESS: 0x004013b4     CALL: KERNEL32!OpenProcess                   --> 4
RETURN ADDRESS: 0x004013ee     CALL: KERNEL32!VirtualAllocEx                --> 5
RETURN ADDRESS: 0x00401425     CALL: KERNEL32!WriteProcessMemory    --> 6
RETURN ADDRESS: 0x0040146b     CALL: KERNEL32!CreateRemoteThread    --> 7
```

This is a clear indication of this binary reading a file and **injecting code** into another process.

Finding Interesting Functions

Here's the API Trace log of another binary,

```
RETURN ADDRESS: 0x00443c29     CALL: inet_ntoa                     --> point 1
RETURN ADDRESS: 0x0044a6ee     CALL: KERNEL32!HeapAlloc
RETURN ADDRESS: 0x00446866     CALL: KERNEL32!GetLocalTime
RETURN ADDRESS: 0x0044a6ee     CALL: KERNEL32!HeapAlloc
RETURN ADDRESS: 0x00443f79     CALL: socket                        --> point 2
RETURN ADDRESS: 0x00443fb5     CALL: setsockop
RETURN ADDRESS: 0x00443fd0     CALL: setsockopt
RETURN ADDRESS: 0x00444045     CALL: ntohl
RETURN ADDRESS: 0x0044404f     CALL: ntohs
RETURN ADDRESS: 0x00444063     CALL: bind                          --> point 3
RETURN ADDRESS: 0x0044412c     CALL: ntohl
RETURN ADDRESS: 0x0044413c     CALL: ntohs
RETURN ADDRESS: 0x0043adf6     CALL: WSAAsyncSelect
RETURN ADDRESS: 0x0044416b     CALL: connect                       -->  point 4
RETURN ADDRESS: 0x00444176     CALL: WSAGetLastError
RETURN ADDRESS: 0x00441979     CALL: USER32!DispatchMessageA
RETURN ADDRESS: 0x00444ce0     CALL: KERNEL32!GetTickCount
RETURN ADDRESS: 0x00444cfa     CALL: KERNEL32!QueryPerformanceCounter
RETURN ADDRESS: 0x00444499     CALL: recv                          --> point 5
RETURN ADDRESS: 0x0044a8c6     CALL: KERNEL32!HeapFre
RETURN ADDRESS: 0x0043adf6     CALL: WSAAsyncSelect
RETURN ADDRESS: 0x004441f7     CALL: closesocket
RETURN ADDRESS: 0x0044a8c6     CALL: KERNEL32!HeapFree
```

Marked points here reflects interesting functions used by this binary revealing **network activity.**

Extending API Tracing with IDAPython

We can further use these Addresses from 'API Trace Log file' in **IDA** to identify functions and cross references.

Below is the simple **IDAPython script** that will read the above script log file and colour the calls in IDA database.

```
'''
Author: Amit Malik
http://www.securityxploded.com
'''
from idaapi import *
from idc import *
import sys
class logparse():
        def __init__(self,file_path):
                self.file_path = file_path
                self.fp = open(self.file_path,'r')
                self.data = self.fp.readlines()

        def parser(self):
                dict = {}
                for line in self.data:
                        line_slice = line.split()
                        address = line_slice[2]
                        name = line_slice[4]
                        dict[address] = name

                for ea in dict.keys():
                        print dict[ea]
                        ea_c = PrevHead(ea)
                        SetColor(ea_c,CIC_ITEM,0x8CE6F0)
                return

def main():
        file_path = AskFile(0,"*.*","Enter file name: ")
        logobj = logparse(file_path)
        logobj.parser()
        return

if __name__ == '__main__':
        main()
```

Conclusion

In this article, you have learnt how to do **'API Call Tracing'** using PyDbg/IDAPython scripts and perform useful tasks such as Unpacking, Binary Profiling, Discovering Interesting functions etc.

There are lot more useful applications of API Tracing and this article just serve as startup guide.

References

1. Pydbg - http://code.google.com/p/paimei/
2. OllyDbg - http://www.ollydbg.de/
3. Windbg - http://msdn.microsoft.com/windbg
4. IDAPython - http://code.google.com/p/idapython/
5. Reference Guide - Reversing & Malware Analysis Training

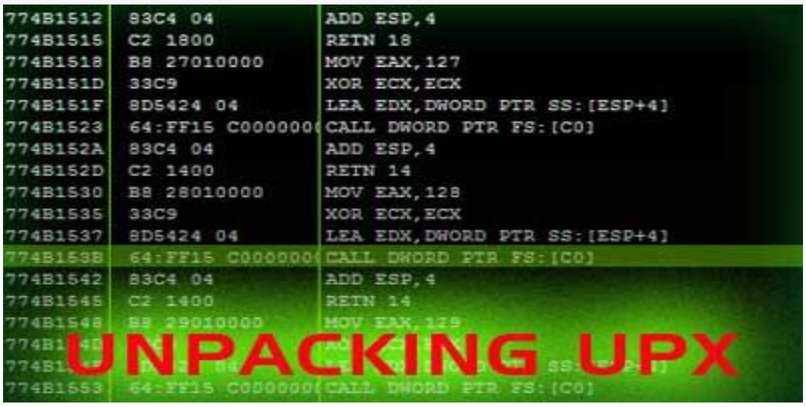
Manual Unpacking of UPX using OllyDbg

Author: Nagareshwar Talekar

Introduction

In this tutorial, you will learn how to unpack any **UPX** packed Executable file using **OllyDbg**

UPX is a free, portable, executable packer for several different executable formats. It achieves an excellent compression ratio and offers very fast decompression.



Here we will do live debugging using OllyDbg to fully unpack and produce the original executable FILE from the packed file.

Packing EXE using UPX

To start with, we need to pack sample EXE file with UPX. First you need to download latest UPX packer from UPX website and then use the following command to pack your sample EXE file.

```
upx -9 c:\sample.exe
```

If you already have UPX packed binary file then proceed further. In such case make sure to use **PEiD** or **'RDG Packer Detector'** to confirm if it is packed with UPX as shown in the screenshot below.

UPX Unpacking Process

Before we begin with unpacking exercise, lets try to understand the working of UPX.

When you pack any Executable with UPX, all existing sections (text, data, rsrc etc) are compressed. Each of these sections are named as **UPX0, UPX1** etc. Then it adds new code section at the end of file which will actually decompress all the packed sections at execution time.

Here is what happens during the execution of UPX packed EXE file.

- Execution starts from **new OEP** (from newly added code section at the end of file)
- First it saves the current Register Status using **PUSHAD** instruction
- All the Packed Sections are Unpacked in memory
- Resolve the **import table** of original executable file.
- Restore the original Register Status using **POPAD** instruction
- Finally Jumps to Original Entry point to begin the actual execution

Manual Unpacking of UPX

Here are the standard steps involved in any Unpacking operation

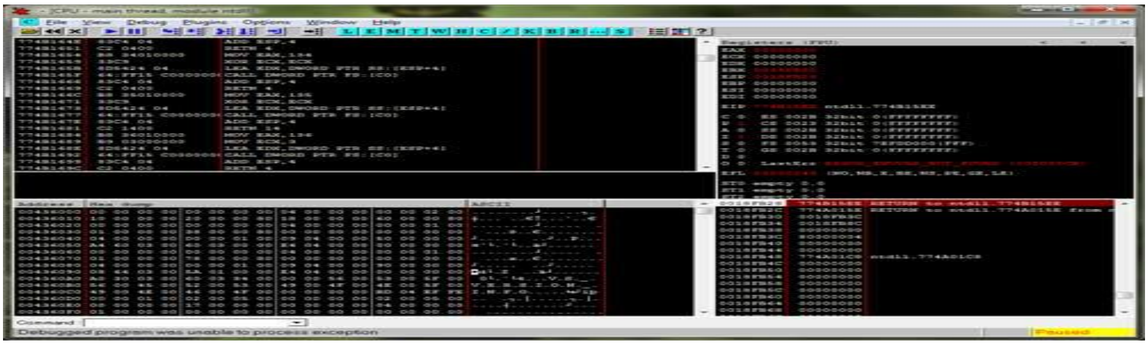
- Debug the EXE to find the real OEP (Original Entry Point)
- At OEP, Dump the fully Unpacked Program to Disk
- Fix the Import Table

Based on type and complexity of Packer, unpacking operation may vary in terms of time and difficulty.

UPX is the basic Packer and serves as great example for anyone who wants to learn Unpacking.

Here we will use **OllyDbg** to debug & unpack the UPX packed EXE file. Although you can use any debugger, OllyDbg is one of the **best ring 3 debugger** for Reverse Engineering with its useful plugins.

Here is the screenshot of OllyDbg in action



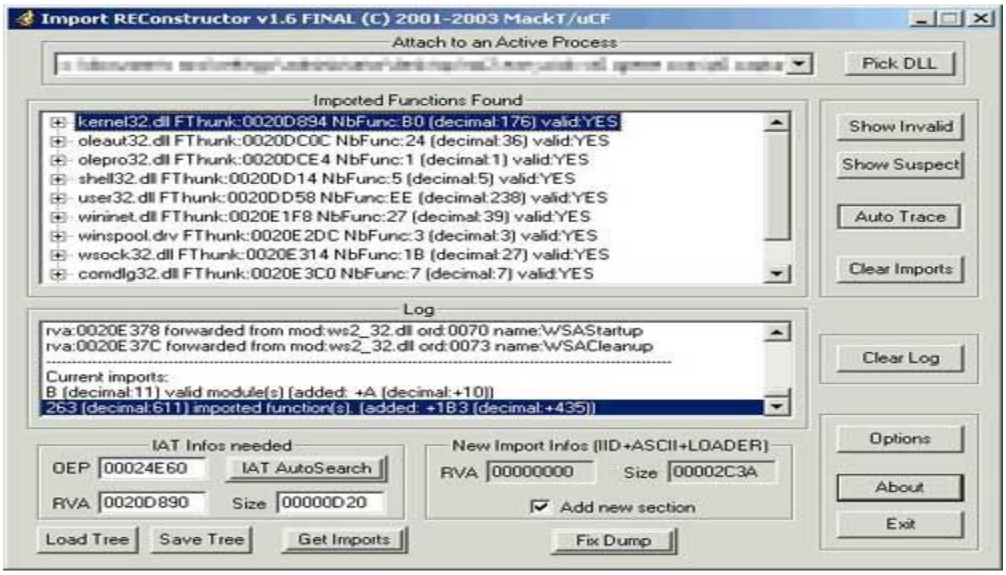
Lets start the unpacking operation

- Load the UPX packed EXE file into the OllyDbg
- Start tracing the EXE, until you encounter a **PUSHAD** instruction. Usually this is the first instruction or it will be present after first few instructions based on the UPX version.
- When you reach PUSHAD instruction, put the Hardware Breakpoint (type **'hr esp-4'** at command bar) so as to stop at POPAD instruction. This will help us to stop the execution when the POPAD instruction is executed later on.
- Other way is to manually search for POPAD (**Opcode 61**) instruction and then set Breakpoint on it.
- Once you set up the breakpoint, continue the execution (press F9).
- Shortly, it will break on the instruction which is immediately after POPAD or on POPAD instruction based on the method you have chosen.
- Now start step by step tracing with F7 and soon you will encounter a **JMP instruction** which will take us to actual OEP in the original program.
- When you reach OEP, dump the whole program using **OllyDmp plugin** (use default settings). It will automatically fix all the Import table as well.
- That is it, you have just unpacked UPX !!!

Fixing Import Table

In the current example, OllyDmp plugin will take care of fixing the **Import table**.

However for most of the packers, we need to use advanced tool called **ImpRec** (Import Reconstructor). ImpREC is highly advanced tool used for fixing the import table. It provides multiple methods to trace the API functions as well as allow writing custom plugins.



For interested users, here are simple instructions on how to fix Import Table using ImpRec.

- When you are at the OEP of the program, just dump the memory image of binary file using Ollydmp **WITHOUT**asking it to fix the Import table.
- Now launch the **ImpREC** tool and select the process that you are currently debugging.
- Then in the ImpREC, enter the actual OEP (enter only RVA, not a complete address).
- Next click on **'IAT Autosearch'** button to automatically search for Import table.
- Now click on **'Get Imports'** to retrieve all the imported functions. You will see all the import functions listed under their respective DLL names.
- If you find any import function which is invalid (marked as **VALID: NO**) then remove it by by right clicking on it and then from the popup menu, click on **'Delete Thunks'**.
- Once all the import functions are identified, click on **"Fix Dump"** button in ImpREC and then select the previously dumped file from OllyDbg.
- Now run the final fixed executable to see if everything is alright.

For advanced packers, you may have to use different methods in ImpRec and some times need to write your own custom plugin to resolve the import table functions.

For more interesting details refer to our PESpin ImpRec plugin.

Video Demonstration

http://vimeo.com/42197903

This video demonstration uses slightly different way to put a hardware breakpoint than described in the article. Also it uses **ImpREC** to fix import table which is useful while unpacking advanced packers.

- Load your EXE in Ollydbg
- Step Over (Shortcut-F8) **PUSHAD** instruction
- Next Go to ESP (right click and follow in DUMP Window)
- Put Hardware Read Breakpoint (Access) on first dword at ESP. (This is similar 'hr esp-4 at PUSHAD instruction as described earlier)
- Now Run EXE until we hit breakpoint (shortcut-F9)
- It will break right after POPAD instruction.
- You will see a JMP instruction few lines below the current instructions. Put breakpoint on JMP
- Run exe again until it stops at JMP instruction (shortcut-F9)
- Step Over JMP (Shortcut- F8)
- Now we are at **OEP**, Here just Dump Process using OllyDump **without fixing** Import table.
- Here we will use ImpREC to fix the import table as mentioned in **'Fixing Import Table'** section.
- Finally after fixing import table, run the new unpacked EXE to make sure it is perfect !

References

1. UPX: Ultimate Packer for Executables.
2. OllyDbg: Popular Ring 3 Debugger.
3. ImpREC: Import Table Reconstruction Tool
4. PESpin Plugin for ImpREC
5. RDG Packer Detector
6. PEid Packer Detector

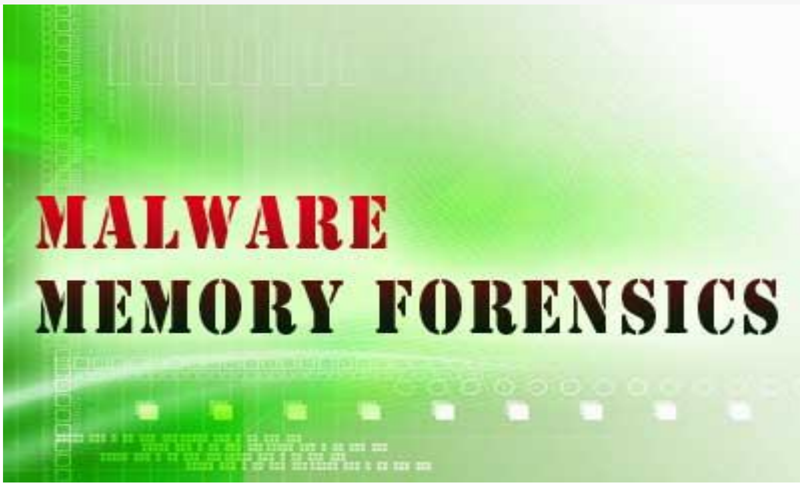
Malware Memory Forensics

Author: Monnappa

Introduction

Memory Forensics is the analysis of the memory image taken from the running computer.

In this article, we will learn how to use Memory Forensic Toolkits such as **Volatility** to analyze the memory artifacts with practical real life forensics scenario.



This article is the part of our free **"Reverse Engineering & Malware Analysis Course"**. You can visit our training page **here** and all the presentations of previous sessions **here**

Why Memory Forensics?

Memory forensics can help in extracting forensics artifacts from a computer's memory like running process, network connections, loaded modules etc etc. It can also help in unpacking, **rootkit detection** and reverse engineering.

Below are the list of steps involved in memory forensics

1. **Memory Acquistion** - This step involves dumping the memory of the target machine. on the physical machine you can use tools like Win32dd/Win64dd, Memoryze, DumpIt, FastDump on the virtual machine, acquiring the memory image is easy, you can do it by suspending the VM and grabbing the ".vmem" file.
2. **Memory Analysis** - once a memory image is acquired, the next step is analyze the grabbed memory dump for forensic artifacts. tools like Volatility and Memoryze can be used to analyze the memory

Volatility - A Quick Overview

Volatility is an advanced memory forensic framework written in python. It can be installed on multiple operating systems (Windows, Linux, Mac OS X), Installation details of volatility can be found here.

Volatility Syntax & Usage

```
* using -h or --help option will display help options and list of a
available plugins
        example: python vol.py -h
* Use -f   and --profile to indicate the memory dump you are analyzing
        example: python vol.py -f mem.dmp --profile=WinXPSP3x86
* To know the --profile  info use below command:
        example: python vol.py -f mem.dmp imageinfo
```

Demonstration - Memory Forensics

In order to understand memory forensics and the steps involved. I have created a scenario, our analysis and flow will be based on the below scenario.

Demo Scenario

Your security device alerts, show malicious http connection to ip address 208.91.197.54 from a source ip 192.168.1.100 on 8th june 2012 at around 13:30hrs...you are asked to investigate and do memory forensics on that machine 192.168.1.100

Preparation Steps

To start with, acquire the memory image from 192.168.1.100, using memory acquistion tools. for the sake of demo, the memory dump file is named as **"infected.dmp"**.

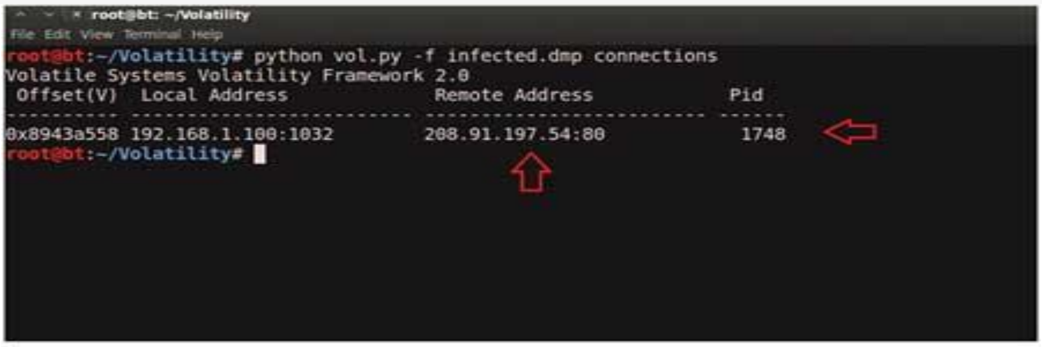
Demonstration - Memory Analysis

Now that we have acquired "infected.dmp", lets start our analysis

Step 1: Start with what you know

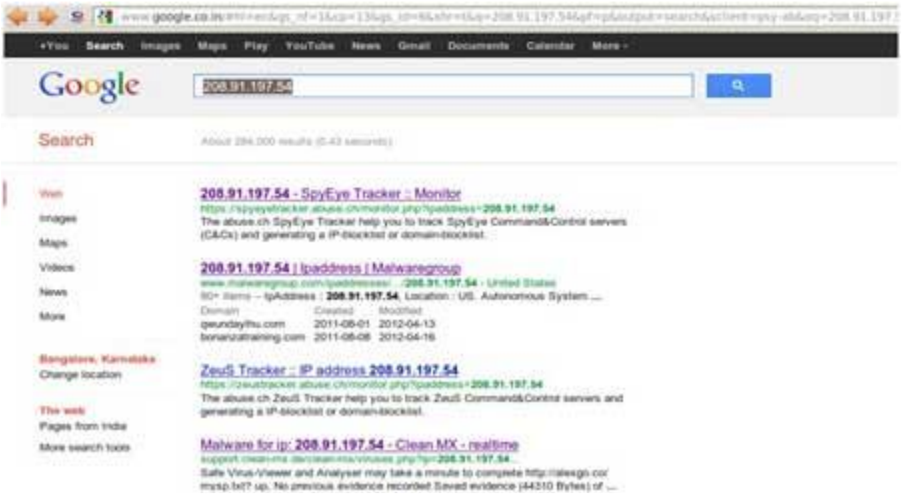
We know from the security device alert that the host was making an http connection to **208.91.197.54**. so lets look at the network connections.

Volatility's connections module, shows connection to the malicious ip made by pid 1748



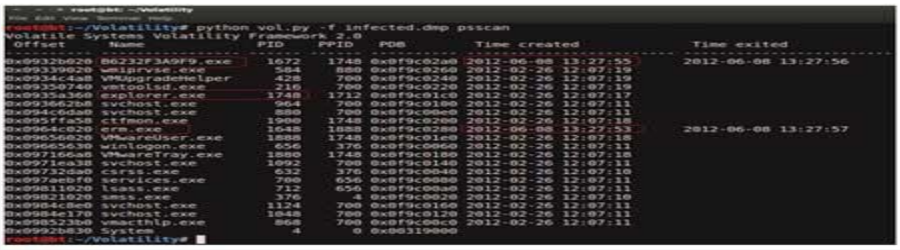
Step 2: Info about 208.91.197.54

Google search shows this ip 208.91.197.54 to be associated with malware, probably "**SpyEye**", we need to confirm that yet.



Step 3: Who is Pid 1748?

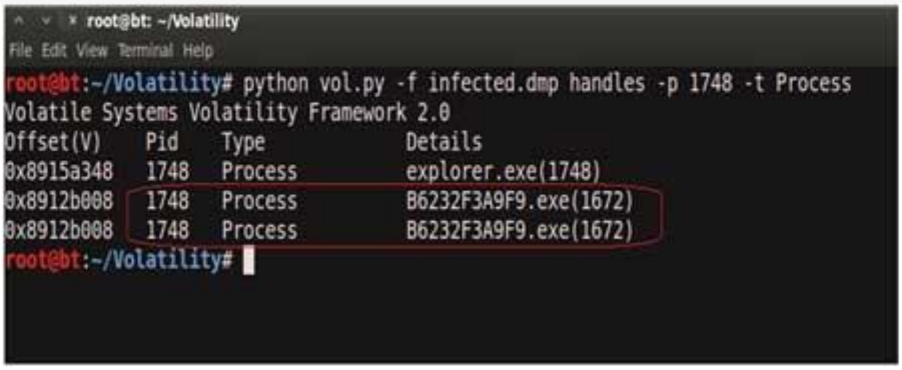
Since the network connection to the ip 208.91.197.54 was made by pid 1748, we need to determine which process is associated with pid 1748. "**psscan**" shows pid 1748 belongs to explorer.exe, also two process created during same time reported by security device (i.e june 8th 2012)



Step 4: Process handles of explorer.exe

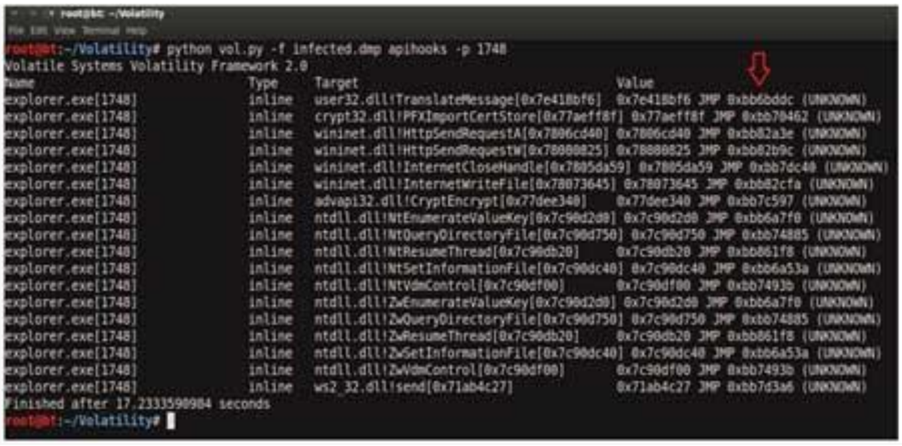
Now that we know explorer.exe (which is an operating system process) was making connections to the malicious ip, there is a possibility that explorer.exe is infected.

Lets looks at the process handles of explorer.exe. The below screenshot shows Explorer.exe opens a handle to the B6232F3A9F9.exe, indicating explorer.exe might have created that process, which might also be malicious…Lets focus on explorer.exe for now



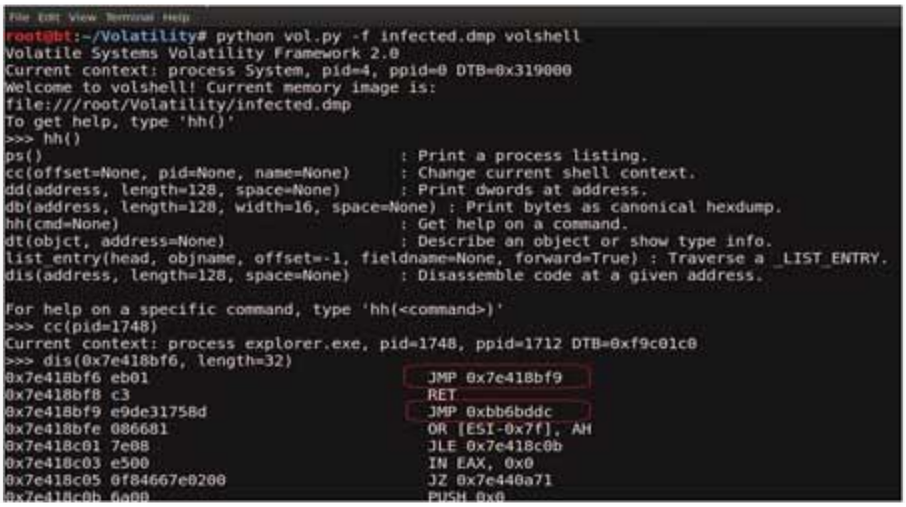
Step 5: API Hooks in explorer.exe

APIhooks module show, inline API hooks in explorer.exe and jump to an unknown location



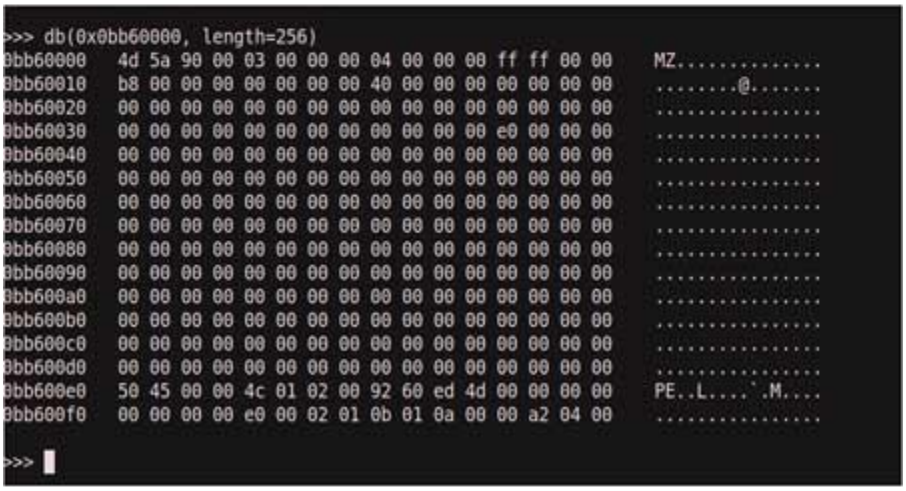
Step 6: Exploring the Hooks

Disassembled hooked function (TranslateMessage), shows a short jump and then a long jump to malware location



Step 7: Embedded EXE in explorer.exe

Printing the bytes at the hooked location, show the presence of **embedded executable** in explorer.exe

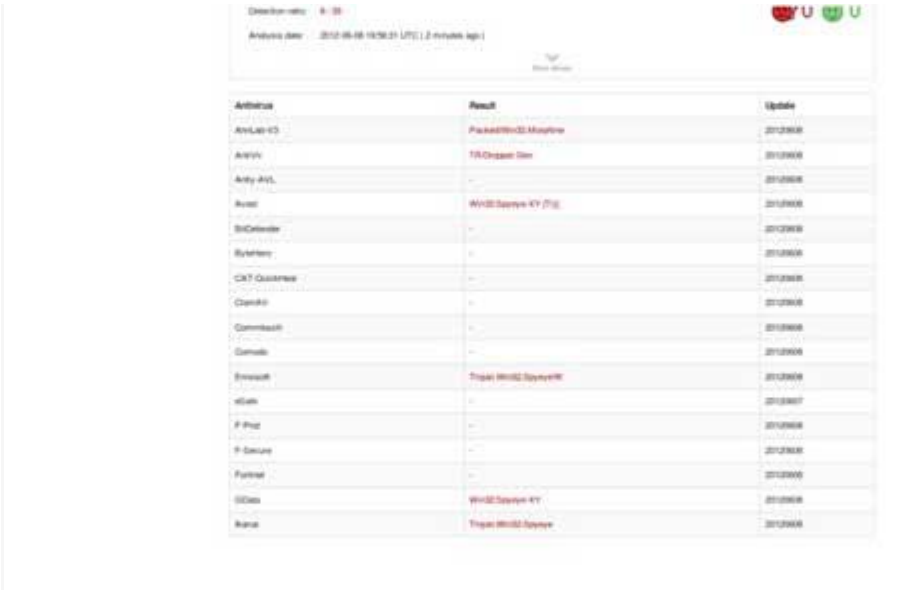


Step 8: Dumping the embedded EXE

VadDump tool dumps the embedded exe from explorer.exe

Step 9: VirusTotal Submission

Submission to VirusTotal, confirms the dumped executable as component of **"SpyEye"**

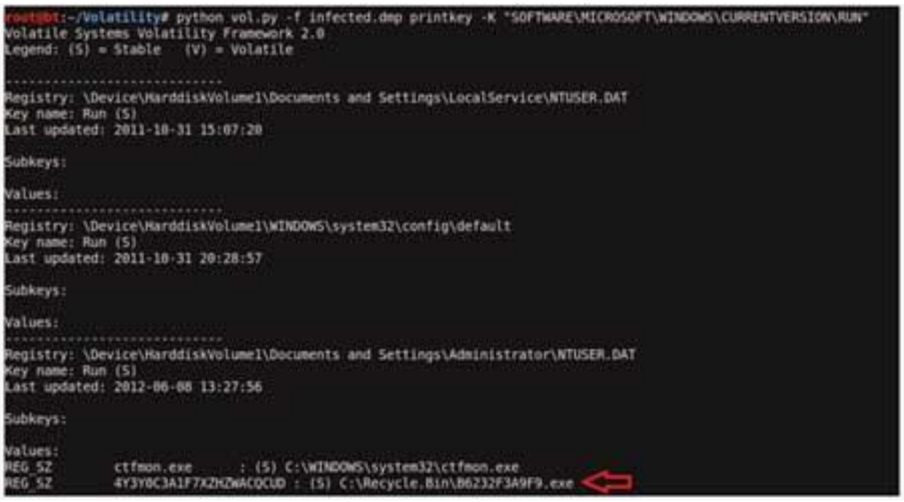


Step 10: Can we get more info?

Strings extracted from the dumped executable, show reference to interesting artifacts (executable and the registry key), it also shows the path to the suspicious executable B6232F3A9F9.exe.

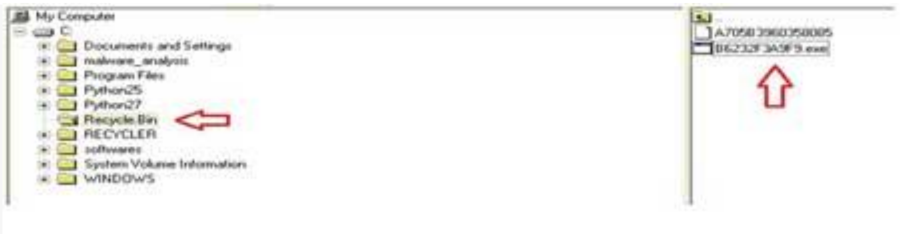
Step 11: Printing the Registry Key

Printing the registry key determined from the above step(step 10) shows that, malware creates registry key to survive the reboot



Step 12: Finding the Malicious EXE on Infected Machine

Now that we know the path to the **suspicious executable**, lets find it on the infected machine. Finding malicious sample from infected host and virustotal submission confirms SpyEye infection.



Conclusion

Memory forensics is a powerful technique and with a tool like **Volatility** it is possible to find and extract the forensic artifacts from the memory which helps in incident response, malware analysis and reverse engineering.

References

1. Reversing Training Session 6 – Malware Memory Forensics
2. Volatility - An advanced memory forensics framework
3. Volatility - Volatile memory analysis research
4. MoonSols Windows Memory Toolkit

DLL Injection and Hooking

Author: Amit Malik

Introduction

In this article we will learn about **DLL Injection** and then using it to perform **Inline Hooking** in remote process with practical step by step illustrations.

This is the part of our free **"Reverse Engineering & Malware Analysis Course"**.

You can visit our training page **here** and all the presentations of previous sessions **here**

In windows each process has its own virtual address space in which it can load and unload any DLL at any time. But that loading and unloading of DLL is initiated by the process itself. Sometimes we may want to **load a DLL into a process** without the process knowledge.

There are many reasons (legitimate or otherwise) to do it. For example a malware author may want to **hide the malicious activity** by loading a DLL into a trusted process or may want to bypass security devices while on the other hand a person may want to extend the functionality of the original program. But for both the activities steps are same.

Here we will discuss on various way to **Inject our code/DLL** into remote process with practical examples. Then we will extend it to **hook specific API** function in the target process to perform our own tasks.

DLL Injection

If I am not mistaken then approximately 45-50% malwares these days use **code injection** to carry out the malicious activities. So it is very crucial to understand the concept of DLL injection for a **malware analyst**.

I will demonstrate the technique using assembly programming language. If your development environment is not ready then i would highly recommend reading my previous article on "Assembly programming basics – A beginner's guide" to get starting with assembly programming language.

There are couple of method by which we can inject DLL into a process. The latest versions of windows enforce **session separation** so some of the methods may not work on the latest version of windows like windows 7/8.

Couple of Dll Injection Methods:

```
1. Window hooks (SetWindowsHookEX)
2. CreateRemoteThread
3. App_Init registry key
4. ZwCreateThread or NtCreateThreadEx ? Global method (works well on all
   versions of windows)
5. Via APC (Asynchronous procedure calls)
```

In this article I will use **CreateRemoteThread** [Reference 1] method because it is the simplest approach and explains the overall logic. CreateRemoteThread will not work from windows vista onwards due to **Session Separation/Isolation** [Reference 4]. In such case you can use similar but undocumented function, NtCreateThread [Reference 2]

In fact it is not the problem with the CreateRemoteThread, it is the CsrClientCallServer method from Ntdll that returns false. If we can patch CsrClientCallServer to return success then we can inject DLL into a process using CreateRemoteThread itself. You can read more about it here.

Here I will focus on CreateRemoteThread on windows XP.

DLL Injection using CreateRemoteThread

There are primarily two situations

```
1. Inject DLL into a running process
2. Create a process and Inject DLL into it.
```

#2 is more suitable for this article because in later section I will cover hooking as well. While #1 is just the part of #2.
Below is the line from MSDN about the CreateRemoteThread API.

> Creates a thread that runs in the virtual address space of another process.

So it means **CreateRemoteThread** can create a thread into another process or we can say that it can execute a function into another process.

Let's look into its syntax.

```
HANDLE WINAPI CreateRemoteThread(
  __in   HANDLE hProcess,                                             ?--
------ 1
  __in   LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in   SIZE_T dwStackSize,
  __in   LPTHREAD_START_ROUTINE lpStartAddress, ?---------2
  __in   LPVOID lpParameter,                                          ?---
------3
  __in   DWORD dwCreationFlags,                                 ?---------
4
  __out  LPDWORD lpThreadId
);
```

Mentioned parameters are critical for our task

```
#1 - handle to the process in which the thread is to be created.
#2 - A pointer to function or entry point of the thread that is going to be
executed
#3 - parameters to the function
#4 - Creation state of the thread
```

We all know that kernerl32.dll export **LoadLibrary** API to load DLL at run time and also kernel32.dll is loaded by default into every process. So we can pass LoadLibrary address to #2 and parameter to LoadLibrary in #3. When we pass arguments in this order then CreateRemoteThread will execute LoadLibrary with its parameter in another process and hence loads the DLL into external process.

The only problem here is that parameter to LoadLibrary must be in target process. For example if we use LoadLibrary (#2) with "mydll.dll"(#3) as parameter to Loadlibrary then the name "mydll.dll" must be in our target process.

Fortunately windows provide API to do that as well. We can write into any process using **WriteProcessMemory** and can allocate space into another process using VirtualAllocEx API. But Before that we need handle to our process, we can get that using OpenProcess or **CreateProcess** API.

So our order will be:

```
1. Use OpenProcess or CreateProcess API to get the handle of our target
   process
2. Use VirtualAllocEx to allocate space into our target process
3. Use WriteProcessMemory to write our DLL name into our target process
4. Use CreateRemoteThread to inject our DLL into our target process
```

Above steps are enough to inject our DLL into a process. Although to inject into a system process we first have to set **se_debug privilege** to our process (means the process that will inject DLL into another process) but for simplicity I am ignoring that part.

If you remember "two situations" from the beginning of this part then we need a bit of more work for #2 i.e Create a process and Inject DLL into it.

We first have to create a process and after that we will use above steps to inject our DLL into newly created process.

Let's look into CreateProcess syntax:

```
BOOL WINAPI CreateProcess(
  __in_opt      LPCTSTR lpApplicationName,
  __inout_opt   LPTSTR lpCommandLine,
  __in_opt      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  __in_opt      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in          BOOL bInheritHandles,
  __in          DWORD dwCreationFlags,                 ?-------- 1
  __in_opt      LPVOID lpEnvironment,
  __in_opt      LPCTSTR lpCurrentDirectory,
  __in          LPSTARTUPINFO lpStartupInfo,
  __out         LPPROCESS_INFORMATION lpProcessInformation
);
```

Here **dwCreationFlags** is the important parameter. If you look into its definition on MSDN then you will see that it is used to control the creation of a process. We can set it to "CREATE_SUSPENDED" to create a process into suspended mode.

With **CREATE_SUSPENDED** flag CreateProcess will create the process and stop the execution of the main thread at the entry point of the thread. To start the process we can use **ResumeThread** API.

So our steps will be

```
1. Create Process in suspended state
2. Inject DLL into the process using above steps
3. Resume the process
```

Here is the complete program which mimics above steps

```
;Author: Amit Malik
;http://www.SecurityXploded.com
;No error checking

.386
.model flat, stdcall
option casemap:none
```

```
include windows.inc
include msvcrt.inc
include kernel32.inc

includelib kernel32.lib
includelib msvcrt.lib

.data
greet   db      "enter file name: ",0
sgreet  db      "%s",0
dreet   db      "enter DLL name: ",0
dgreet  db      "%s",0
apiname db      "LoadLibraryA",0
dllname db      "kernel32.dll",0

.data?
processinfo      PROCESS_INFORMATION <>
startupinfo     STARTUPINFO <>
fname   db      20      dup(?)
dname   db      20      dup(?)
dllLen  dd      ?
mAddr   dd      ?
vpointer        dd      ?
lpAddr  dd      ?


.code
start:

invoke crt_printf,addr greet
invoke crt_scanf,addr sgreet,addr fname
invoke crt_printf,addr dreet
invoke crt_scanf,addr dgreet,addr dname
invoke LoadLibrary, addr dllname
ov mAddr,eax
invoke GetProcAddress,mAddr,addr apiname
mov lpAddr,eax

;create process in suspended state
invoke CreateProcess,addr fname,0,0,0,0,CREATE_SUSPENDED,0,0,addr
startupinfo,addr processinfo
invoke crt_strlen,addr dname
mov dllLen,eax

; Allocate the space into the newly created process
```

```
invoke
VirtualAllocEx,processinfo.hProcess,NULL,dllLen,MEM_COMMIT,PAGE_EXECUTE_REA
DWRITE
mov vpointer,eax

; Write DLL name into the allocated space
invoke WriteProcessMemory,processinfo.hProcess,vpointer,addr
dname,dllLen,NULL

; Execute the LoadLibrary function using CreateRemoteThread into the
previously created process
invoke
CreateRemoteThread,processinfo.hProcess,NULL,0,lpAddr,vpointer,0,NULL
invoke Sleep,1000d

; Finally resume the process main thread.
invoke ResumeThread,processinfo.hThread
xor eax,eax
invoke ExitProcess,eax

end start
```

Select console application in **WinAsm** and assemble the above code. It should create a process and inject our DLL into it.

For eg: you can create calc.exe process and can inject urlmon.dll into it, by default calc.exe doesn't load urlmon.dll.

Hooking

Here is definition of **Hooking** from Wikipedia

In computer programming, the term hooking covers a range of techniques used to alter or augment the behaviour of an operating system, of applications, or of other software components by intercepting function calls or messages or events passed between software components. Code that handles such intercepted function calls, events or messages is called a "hook"

Hooking is the most powerful technique available in computer software. A person can do almost everything on a system by applying hooks on the right locations.

As stated in the definition that in hooking we intercept function calls or messages or events. Because it is taking the advantage of flow of execution so we can apply hooks on multiple locations from original file to system calls.

Primarily Hooks can be divided into two parts

```
1. User mode hooks
        1. IAT (Import Address Table) Hooking
        2. Inline Hooking
        3. Call Patching in binary etc..
2. Kernel Mode hooks
        1. IDT Hooking
        2. SSDT Hooking etc..
```

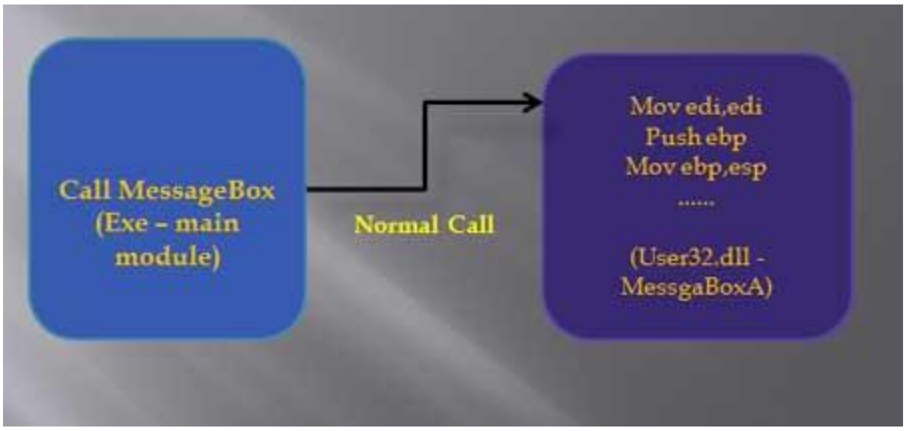
In this article I will discuss **Inline hooking** technique which is one of the more effective hooking techniques.

Inline Hooking

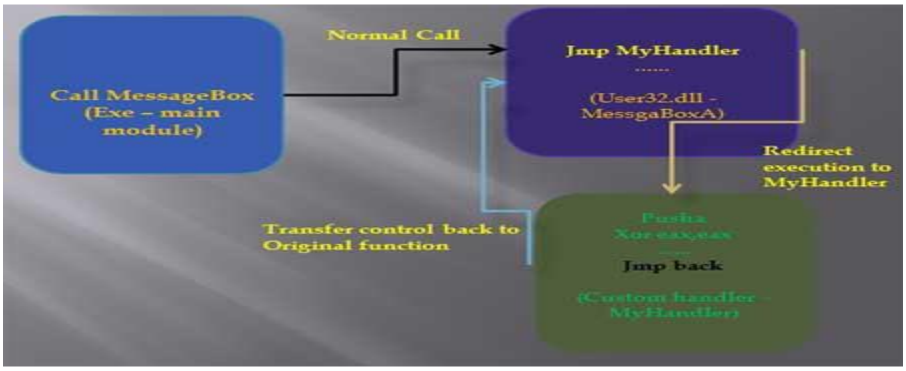
In Inline hooking we overwrite the **first 5 byte of the function** or API to redirect the flow of execution to our code. The 5 bytes can be JMP, PUSH RET or CALL instruction.

Visually it can be explained by the following figures

Screenshot 1: Normal Call (Without hooking)



Screenshot 2: Call after hooking



As you can see in the above picture that the MessageBox function starting bytes are overwritten by JMP to MyHandler function. In MyHandler function we do our stuff and then transfer the control back to original function i.e MessageBox.

Now let's create a DLL that will hook MessageBox API and display our custom message instead of the real message.

To make a DLL we need following things:

```
1. MessageBoxA API address i.e pointer
2. Our function or code address i.e pointer
```

We can get MessageBoxA Api address using GetProcAddress.

Here are the steps:

```
1. Get MessageBoxA address
2. Get custom code or function address
3. Overwrite bytes at #1 with JMP to #2
4. Modify the parameter of original call
5. Transfer control back to #1
```

Here is the complete code deomonstrating Inline Hooking MessageBox function

```
;Author: Amit Malik
;http://www.SecurityXploded.com
;No error checking

.386
.model flat,stdcall
option casemap:none

include windows.inc
include kernel32.inc
include msvcrt.inc
include user32.inc


includelib kernel32.lib
includelib msvcrt.lib
includelib user32.lib


.data

tszMsg          db      "Hello from Hooking Function",0
userDll         db      "user32.dll",0
msgapi          db      "MessageBoxA",0
```

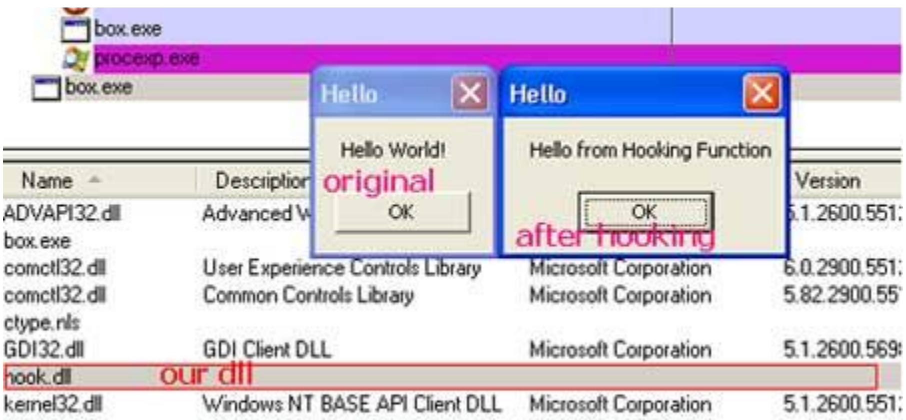
```
.data?
oByte1  dd       ?
oByte2  dd       ?
userAddr         dd       ?
msgAddr dd       ?
nOldProt         dd       ?

.code
LibMain proc hInstDLL:DWORD, reason:DWORD, unused:DWORD
 .if reason == DLL_PROCESS_ATTACH
        invoke LoadLibrary,addr userDll
        mov userAddr,eax

        ; Get MessageBoxA address from user32.dll
        invoke GetProcAddress,userAddr,addr msgapi
        mov msgAddr, eax

    ; Set permission to write at the MessageBoxA address
        invoke VirtualProtect,msgAddr,20d,PAGE_EXECUTE_READWRITE,OFFSET
nOldProt

    ; Store first 8 byte from the MessageBoxA address
        mov eax,msgAddr
        mov ebx, dword ptr DS:[eax]
        mov oByte1,ebx
        mov ebx, dword ptr DS:[eax+4]
        mov oByte2,ebx

        patchlmessagebox:
                ; Write JMP MyHandler (pointer) at MessageBoxA address
                mov byte ptr DS:[eax],0E9h
                ; move MyHandler address into ecx
                mov ecx,MyHandler
                add eax,5
                sub ecx,eax
                sub eax,4
                mov dword ptr ds:[eax],ecx

    .elseif reason == DLL_PROCESS_DETACH
    .elseif reason == DLL_THREAD_ATTACH
    .elseif reason == DLL_THREAD_DETACH
    .endif
    ret
LibMain endp
```

```
MyHandler proc
                pusha
                xor eax,eax
                mov eax,msgAddr

        ; change the lpText parameter to MessageBoxA with our text
                mov dword ptr ss:[esp+028h],offset tszMsg

        ; Restore the bytes at MessageBoxA address
                mov ebx,oByte1
                mov dword ptr ds:[eax],ebx
                mov ebx,oByte2
                mov dword ptr ds:[eax+4],ebx

        ; Restore all registers
                popa

        ;jump to MessageBoxA address (Transfer control back to MessageBoxA)
                jmp msgAddr
MyHandler endp

end LibMain
```

Select standard DLL under "New Project" tab in WinAsm and paste the above code into the editor area and assemble it.

Now we have our DLL that will hook MessageBoxA and change the lpText parameter to our message.
We will inject this DLL into a "Hello world" program that I shown in my previous article "Assembly Programming – A beginner's guide" with the help of our DLL inject program.

The output is shown in the below picture:



Conclusion

Both **DLL injection and Hooking** are powerful techniques and popularly used by malicious software as well as legitimate software from the years.

But as the saying goes if you have **nuclear power** then it is entirely depends on you whether you make a nuclear missile or use that power for solving problems.

References

1. Three ways to inject code into another process
2. Remote Thread Execution in System Process using NtCreateThreadEx for Vista & Windows7
3. MSR Detour Project - Hook SDK
4. Impact of Session 0 Isolation on Injection

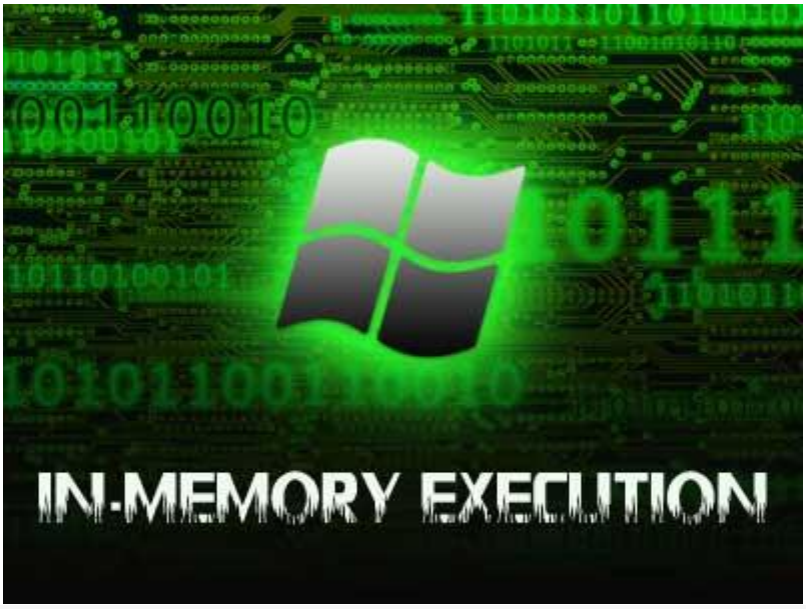
In-Memory Execution of an Executable

Author: Amit Malik

Introduction

This article is the part of our free **"Reverse Engineering & Malware Analysis Course"**.

You can visit our training page **here** and all the presentations of previous sessions **here**



In this article, we will learn how to perform **in-memory or file-less execution** of executable with practical code example.

Here I will explain about some of the fancy techniques used by **exploits** and **malwares** from shellcode perspective. This article requires a strong understanding of PE file format. If you are not comfortable with PE file format then first visit our first training session on **PE Format Basics**.

Technical Introduction

Technically an exploit is the combination of two things

1. Vulnerability – the software security bug
2. Shellcode – the actual malicious payload

Vulnerability gives us control over execution flow while shellcode is the actual payload that carries out the malicious activity. Without the shellcode vulnerability is just a simple software bug.

Further we can divide shellcodes into two parts:

1. Normal shellcodes
2. Staged shellcodes (often times termed as drive by download)

In a **normal shellcode**, shellcode itself carry out the malicious activity for eg: bind shell, reverse shell shellcodes etc. They do not require any other payload to be downloaded for their working. On the other hand **staged shellcodes** require another payload for their working and are often divided into two stages.

Stage 1 – that will download stage 2.
Stage 2 – It is the actual malicious payload

Stage 1 downloads the stage 2 payload and executes it. After that stage 2 will perform all kind of malicious activity. Here the interesting part is how stage 1 executes stage 2 payloads. In this article I will discuss about it in detail.

The two possibilities for the stage 1 shellcode to execute stage 2 shellcode could be,

1. Download the payload, save it on the disk and create a new process
2. Download the payload and execute it directly from the memory

#1 will increase the footprints and moreover there is greater chances of detection by the host based security softwares like **antivirus**.

However in #2, as the payload is executed directly from the memory so it can **bypass** host based security softwares very easily. But unfortunately no windows API provides mechanism to execute file directly from memory. All windows API like CreateProcess, WinExec, ShellExcute etc. requires file to be locally present.

So the question is how we can do that if there is no such API?

In-Memory Execution

I think in this regard the first known work on In-memory execution was done by **ZomBie of 29A labs** and then the Nologin also published its own version of the same. Later on Stephen Fewer from harmony security applied the logic on the DLL and coined a new term **reflective DLL injection** which is the integral part of Metasploit framework.

Interestingly it is possible because the structure of a PE file is exactly the same on disk as in **mapped memory**. So we can easily calculate the offsets or addresses in memory if we know the offset on disk and vice-versa. It makes it possible to mimic the actual operating system loader that loads the executable in memory.

Operating system loader is responsible for process initialization, so if we can make a prototype of it then we can also create a process probably directly from the memory. But before that, we need to take a look into the **OS loader**working especially how it map executable in memory.

Following are the simplified steps that carried out by OS loader when you launch Executables.

1. Read first page of the file which includes DOS header, PE header, section headers etc.
2. Fetch Image Base address from PE header and determine if that address is available else allocate another area. (Case of relocation)
3. Map the sections into the allocated area
4. Read information from import table and load the DLLs
5. Resolve the function addresses and create Import Address Table (IAT).
6. Create initial heap and stack using values from PE header.
7. Create main thread and start the process.

If we can create a programme that can mimic some of the above steps then we can execute exe directly from memory.

For example, consider a situation: you download an exe/dll from internet so until you save it on the disk it will remain in the volatile memory. This means we can read the header information of that file directly from memory and based on the above steps we can execute that file **directly from memory**, in short it is possible to execute an exe/dll without its file or **file-less execution** is possible.

If you take a close look on the above steps then we can easily say that most of the information is stored in the **PE header** itself, which we can read programmatically.

Technically the minimum information required to run any executable is as follows,

1. Address space
2. Proper sections (exe sections) placement into the address space
3. Imported API addresses

Address space

In PE, everything is relative to **Image Base** so if we can get Image Base address allocation then we can proceed to next steps easily else we have to add relocation support to our loader prototype but for this article, I am ignoring that part and will be assuming that we have an allocation with Image Base.

Sections mapped into Address Space

In PE File header, **NumberOfSections** field can give us the total number of sections, after that we can read section's headers and can write on to the proper address in the memory. (We read the offset from PointerToRawData and copy that data at **VirtualAddress** by taking length from SizeOfRawData field).

Imported API addresses

Again by reading **Import Table** structure we can get the names of DLLs and APIs used by the executable. Remember FirstThunk in the import table structure is actually IAT after name resolution.

Memory Execution – Prototype Code

Based on the above information we can write a basic **loader prototype**. Please note that I am ignoring couple of important things in the code intentionally like relocation case, section permissions, ordinal based entries fixes etc.

```
/* In memory execution example */
/*
Author: Amit Malik
http://www.securityxploded.com
Compile in Dev C++
*/

#include
#include
#include

#define DEREF_32( name )*(DWORD *)(name)

int main()
{
     char file[20];
     HANDLE handle;
     PVOID vpointer;
     HINSTANCE laddress;
     LPSTR libname;
     DWORD size;
     DWORD EntryAddr;
     int state;
     DWORD byteread;
     PIMAGE_NT_HEADERS nt;
```

```
	PIMAGE_SECTION_HEADER section;
	DWORD dwValueA;
	DWORD dwValueB;
	DWORD dwValueC;
	DWORD dwValueD;

	printf("Enter file name: ");
	scanf("%s",&file);



	// read the file
	printf("Reading file..\n");
	handle =
CreateFile(file,GENERIC_READ,0,0,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0);

	// get the file size
	size = GetFileSize(handle,NULL);

	// Allocate the space
	vpointer = VirtualAlloc(NULL,size,MEM_COMMIT,PAGE_READWRITE);

	// read file on the allocated space
	state = ReadFile(handle,vpointer,size,&byteread,NULL);
	CloseHandle(handle);
	printf("You can delete the file now!\n");
	system("pause");

	// read NT header of the file
	nt = PIMAGE_NT_HEADERS(PCHAR(vpointer) + PIMAGE_DOS_HEADER(vpointer)-
>e_lfanew);
	handle = GetCurrentProcess();

	// get VA of entry point
	EntryAddr = nt->OptionalHeader.ImageBase + nt-
>OptionalHeader.AddressOfEntryPoint;

	// Allocate the space with Imagebase as a desired address allocation
request
	PVOID memalloc = VirtualAllocEx(
				handle,
				PVOID(nt->OptionalHeader.ImageBase),
				nt->OptionalHeader.SizeOfImage,
				MEM_RESERVE | MEM_COMMIT,
				PAGE_READWRITE
				);

	// Write headers on the allocated space
```

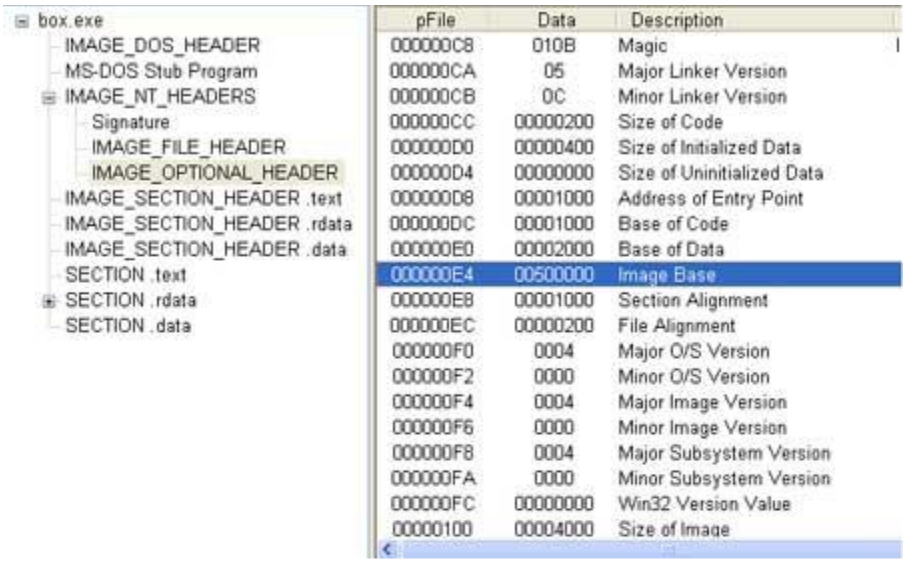
```
    WriteProcessMemory(handle,
    memalloc,
    vpointer,
    nt->OptionalHeader.SizeOfHeaders,
    0
    );


    // write sections on the allocated space
    section = IMAGE_FIRST_SECTION(nt);
    for (ULONG i = 0; i < nt->FileHeader.NumberOfSections; i++)
    {
        WriteProcessMemory(
                        handle,
                        PCHAR(memalloc) + section[i].VirtualAddress,
                        PCHAR(vpointer) + section[i].PointerToRawData,
                        section[i].SizeOfRawData,
                        0
                        );
    }

    // read import dirctory
    dwValueB = (DWORD) &(nt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT]);

    // get the VA
    dwValueC = (DWORD)(nt->OptionalHeader.ImageBase) +
                       ((PIMAGE_DATA_DIRECTORY)dwValueB)-
>VirtualAddress;


    while(((PIMAGE_IMPORT_DESCRIPTOR)dwValueC)->Name)
    {
           // get DLL name
           libname = (LPSTR)(nt->OptionalHeader.ImageBase +
                           ((PIMAGE_IMPORT_DESCRIPTOR)dwValueC)->Name);

           // Load dll
           laddress = LoadLibrary(libname);

           // get first thunk, it will become our IAT
           dwValueA = nt->OptionalHeader.ImageBase +
                             ((PIMAGE_IMPORT_DESCRIPTOR)dwValueC)-
>FirstThunk;

           // resolve function addresses
           while(DEREF_32(dwValueA))
           {
```

```
                dwValueD = nt->OptionalHeader.ImageBase +
DEREF_32(dwValueA);
                // get function name
                LPSTR Fname = (LPSTR)((PIMAGE_IMPORT_BY_NAME)dwValueD)-
>Name;
                // get function addresses
                DEREF_32(dwValueA) = (DWORD)GetProcAddress(laddress,Fname);
                dwValueA += 4;
            }

            dwValueC += sizeof( IMAGE_IMPORT_DESCRIPTOR );

      }


     // call the entry point :: here we assume that everything is ok.
     ((void(*)(void))EntryAddr)();

}
```

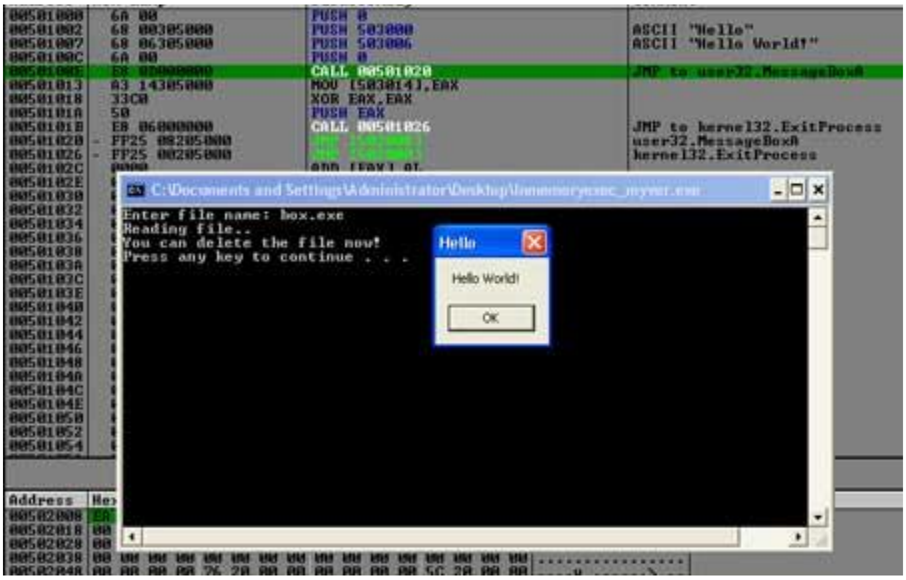
Compile the above code in Dev C++. For proof of concept, I will execute the MessageBox code that I had shown in my 'Assembly Basics' article.

Now perform the following steps,

1. Compile the MessageBox code again but before that select project properties in WinAsm (project->Project Properties->Release) and in Link block add the following command: /BASE:0x500000
2. Click on ok.
3. Now assemble and link the code you will get EXE with 500000 Image Base which is good for our POC



Below snapshot shows you the execution directly from memory,



Conclusion

Recently Kaspersky said that they saw a file less worm, actually these things are not new. Metasploit has file less Trojan from years in terms of reflective DLL injection. Many **malicious codes and packers** use heavily these things. It is also strongly known for security softwares bypassing.

Overall it is very powerful mechanism and must be known to a **malware analyst**.

References

1. Nologin - Remote Library Injection
2. Harmony Security - Reflective DLL Injection
3. In Memory Execution – Zombie