

Applying a Debugger for Windows

Offensive and Defensive Tool Construction

Table of Contents

[Objectives 4](#_Toc477255276)

[Background Reading 4](#_Toc477255277)

[Important Information 4](#_Toc477255278)

[Introduction 5](#_Toc477255279)

[Problem 1 5](#_Toc477255280)

[Problem 2 5](#_Toc477255281)

[Problem 3 5](#_Toc477255282)

[Problem 4 6](#_Toc477255283)

[Problem 5 6](#_Toc477255284)

[Problem 6 6](#_Toc477255285)

[Problem 7 6](#_Toc477255286)

[Problem 8 6](#_Toc477255287)

[Problem 9 7](#_Toc477255288)

Offensive and Defensive Tool Construction

Applying a Debugger for Windows

Objectives

This lab focuses on the following objectives:

* Outline the basics of the Immunity Debugger.
* Find exploit-friendly instructions.
* Describe bad character filtering.
* Defeat anti-debugging code in malware.

Background Reading

Read chapter 5 in the *Gray Hat Python* textbook. The following links are also useful:

* https://docs.python.org/3/library/pdb.html
* http://www.gnu.org/software/gdb/documentation/
* http://sourceware.org/gdb/current/onlinedocs/gdb.pdf.gz
* https://docs.python.org/2/extending/extending.html
* https://support.microsoft.com/en-us/kb/875352
* https://www.sans.org/reading-room/whitepapers/malicious/basic-reverse-engineering-immunity-debugger-36982
* https://sgros-students.blogspot.ca/2014/05/immunity-debugger-basics-part-1.html

# Important Information

* Download the immunity Debugger ***http://debugger.immunityinc.com***
* Download and install gcc for windows: ***http://www.mingw.org/wiki/Getting\_Started***
* Listen to lecture on Immunity
* Read a bit on your own to enhance your knowledge
  + ***https://sgros-students.blogspot.com/2014/09/immunity-debugger-basics-part-2.html***
  + ***https://sgros-students.blogspot.com/2014/05/immunity-debugger-basics-part-1.html***

# Introduction

In this lab we will explore the **Immunity Debugger** and its relationship to python.

Note that you need to have a C development environment, example mingw, installed on your computer for this lab. mingw is an open source toolchain that and you can download from https://sourceforge.net/projects/mingw/. After you install it, add **c:\mingw\bin** to your system path so you can use the gcc compiler without typing in the full path.

# Setup

1. YOu should already have downloaded and installed Immunity Debugger.

Confirm that is was installed in its default location.

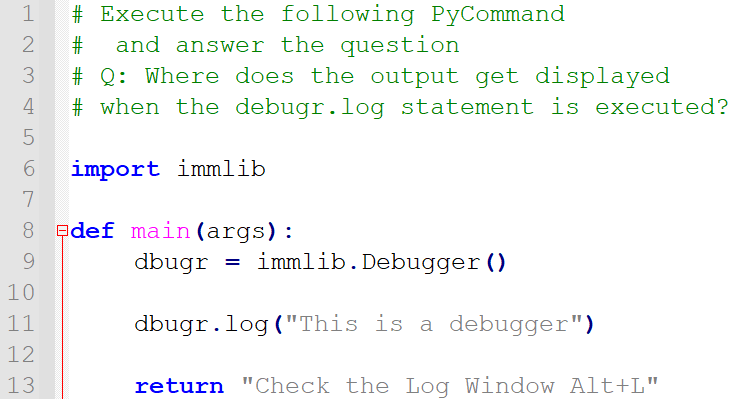
1. Locate the PyCommands folder in **C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands**.

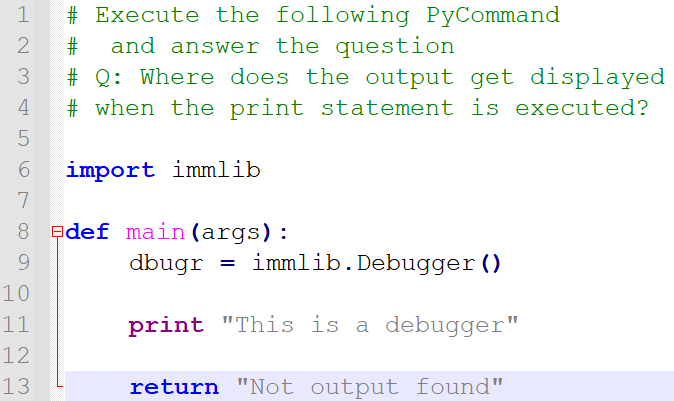
## Questions 1 (5)

1. In your report, show a screenshot of the directory contents. (**To capture all contents, change display to small icons**)
2. What type of files (**file extensions**) are located in this directory?
3. Who is the owner of the folder?
4. How will the owner permissions affect who can save to this folder?

## Sanity Test (No Marks. Do not submit answers for Sanity Test!!!)

Create the following 2 PyCommands and answer the question.

**IMPORTANT: Clear the Log Window prior to executing each PyCommand.**



**Question**: **Where is the output of the print, return and dbugr.log statements and function?**

# Problem 2 (15)

Write a **PyCommand** script named **m07p02.py** that outputs a single line in the **status bar** as shown:

" PYCOMMANDS Executed with X commands"

Your PyCommand should also place output to the Log Window. The requirements of the PyCommand are as follows:

1. Your program should allow run the following command forms (**5pts**)
   1. !m07p02.py
   2. !m07p02.py first
   3. !m07p02.py first second third
2. Your program will then print the following output to the **Log Window** (**5pts**)
   1. **!m07p02**

**Running m07p02.py ...**

**Program args are:**

**Pycommand executed with 0 arguments**

1. **!m07p02 first**

**Running m07p02.py ...**

**Program args are:**

**Arg[0]: first**

**Pycommand executed with 1 arguments**

1. **!m07p02 first second third**

**Running m07p02.py ...**

**Program args are:**

**Arg[0]: first**

**Arg[1]: second**

**Arg[2]: third**

**Pycommand executed with 3 arguments**

1. **!m07p02 first "second third\_forth"**

**Running m07p02.py ...**

**Program args are:**

**Arg[0]: first**

**Arg[1]: "second**

**Arg[2]: third\_forth"**

**Pycommand executed with 3 arguments**

Show a screen shot of the **PyCommand** and the output of step d above, in your report.

## Questions 2 (5pts)

1. **(2pts)** What function/method did you use to print the output to the status bar?
2. **(2pts)** What statement is required to make the output show up in the Logs Windows of Immunity?
3. **(1pts)** Can you use the python print statement to display information?

# Problem 3 (10)

1. Write a C program named **callprintf.c**:

#include<stdio.h>

#include<string.h>

char label[]="The address of main is ";

void print\_all (char \*label,void \*addr)

{

printf ("%s: 0x%08x\n", label,addr);

}

int main (int argc, char \*\*argv)

{

char input[10];

char ans[] = "yes";

printf("Do you want to know the address of main: ");

scanf(" %s", input);

void \*addr\_of\_main; // void pointer

addr\_of\_main = (void \*) &main; // Address of main

if(strcmp(input, ans) == 0)

print\_all(label,addr\_of\_main); // print functions

printf ("Done."); // Exit

return 0;

}

1. Compile it:

gcc -g -m32 -O0 -o callprintf callprintf.c

1. Debug the program, single step from the entry point until you find the main function. (**5pts**)
2. Locate the call to the **print\_all()** function. Make note of how the parameters are passed on the stack.

**Show a screenshot of the instructions used to place items on the stack and the stack to confirm that those values are on the stack. Showing the entire stack and the entire disassembly window will result in a score of 0. You MUST highlight the instructions and the values.**

1. Set up a breakpoint on the entry to the **print\_all**() function, and then single-step through the function up to the **prinf**() function call. (**5pts**)

**Show a screenshot of the instructions used to place items on the stack and the stack to confirm that those values are on the stack. Showing the entire stack and the entire disassembly window will result in a score of 0. You MUST highlight the instructions and the values.**

# Problem 4 (35)

Finding the main function manually was a bit involved. There has to be a better way to do this. Let’s see if this can be done programmatically using Immunity’s Python plugin.

**Create the following PyCommand and analyze it so that you understand what it is doing.**

import immlib

def main(args):

dbugr = immlib.Debugger()

allstrings = dbugr.getReferencedStrings(dbugr.getCurrentAddress())

dbugr.log('%s: %s' % (type(allstrings), allstrings), address=0xB337F00D)

for string in allstrings:

dbugr.log(' %s - %s - %s' % (string[0], string[1], string[2]))

return "Done !!!"

## Questions 4

1. What type of data is contained in the variable allstrings? (**5pts**)
2. What is the purpose of **address: 0xB337F00D** and where does it get displayed in the Log Window? (**5pts**)
3. The output of the for loop shows 3 pieces of information. What do the strings represent? (**5pts**)

|  |  |
| --- | --- |
| string[0] |  |
| string[1] |  |
| string[2] |  |

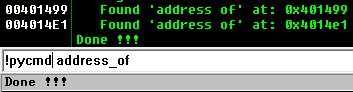
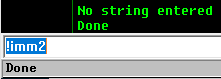
1. Modify the PyCommand by: (**5pts**)
   1. Creating a second function named findString that takes **2** arguments:

**def findString(allstrings, tofind):**

**pass**

* + 1. allstrings
    2. a string to find in allstrings.

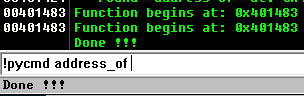
The string to find is entered at the same time as your pyCommand: **!pycmd string\_to\_find**

* + 1. The function returns the address of where the string is found.
  1. Change the output to look as follows:
     1. If the string is found:
     2. If the string is not found or no string entered:

1. Now that we know the address of where the strings were reference. We can proceed to find in what functions they exist.
   1. Search for getFunctionBegin in immlib.py file. Determine the requirements of the function.
   2. Create a function called findMyFun with the following features: (**5pts**)
      1. Parameter 1, the Debugger object
      2. Parameter 2, a list of addresses
      3. Returns the first address of the function.

**def findMyFun(immdbg, myaddrList):**

**pass**

* 1. The output of the program will be as follows:

1. Write a function that will set a breakpoint at that location and run the debugger until it hits that breakpoint. Call your function **setBpRun** with the following features: (**5pts**)
   1. Parameter 1: The Debugger Object
   2. Parameter 2: a list of addresses to set breakpoints at

Provide a screenshot of the **setBpRun** function.

1. Document (***screenshot(s) and code***) your observations of what happens after you run the PyCommand? (**5pts**)

# Problem 5 (See chapter 5 for an example) (25pts)

Write a simple **PyCommand** file named **m07p05.py** that searches for the instruction “**jmp esp**” and prints all addresses where the instruction is found. Confirm that the locations are in a page that is executable.

Submit your **PyCommand** code used to achieve this. (**5pts**)

Show a screenshot of the addresses that are executable only: (**5pts**)

## Questions 5

1. In the log window print the **search\_results** generated by your code. (**5pts**)
   1. What do the values printed represent?
   2. Show screenshot of your log window relevant to this output
2. What is the purpose of the function **getMemoryPageByAddress**? (**5pts**)
3. What is the value returned by **getAccess(human = True)**? In other words, what is in variable access? (**5pts**)
   1. Print the value of access for each returned page
   2. Determine the meaning of **human=True**? **HINT**: Is this parameter required?

**Information for problem 6**

Microsoft has done a great deal of work to make Windows 10 a lot more secure as compared to previous version of the operating system. If your shellcode doesn’t work you can try to disable some features to see if it helps. A bit of experimentation is required here.

**Under settings:**

1. Virus and Threat protection
   1. Manage Settings
   2. Exclusions **=>** Add or remove exclusions
2. App & Browser control
   1. Exploit protection settings **=>** Force Randomization for images (Mandatory ASLR)

# Problem 6 (40)

You will create shellcode from the following C code below. The shellcode can then be placed into the buffer of an exploitable program (Think buffer overflow).

Perform the following steps:

1. Copy the following code into your Windows VM and compile it with the following command:

gcc -m32 -g .\winexec.c -o .\winexec.exe -fno-stack-protector

#include<winbase.h>

#include<windows.h>

int main()

{

WinExec("C:\\windows\\system32\\calc.exe",5);

}

1. Load the program into Immunity Debugger and find the first instruction of the main function.
2. Write a PyCommand to find the address of WinExec.

**Hint:** Debugger.**getAddress** is a method to investigate.

**Hint:** Immunity can look for a name in all modules.

1. Record the address of the WinExec.

**Make note of this address:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. Reboot your VM and repeat the process of finding the WinExec address

**Make note of this address: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Question: Are the 2 recorded addresses different? \_\_\_\_\_**

**Let’s take stock of the INTERESTING information known this point. In no particular order we know:**

1. Memory location(s) where the **JMP ESP** instruction exists, within the executing process.
2. The address of WinExec
3. At the end of main the return instruction causes the EIP to be modified with an address (ie the **RETURN ADDRESS**).
4. Finally, when a function returns to the calling function the ESP gets decremented

**What is unknown:**

1. How to convert the WinExec program to shellcode?
   1. Does the shellcode need a prolog/epilog?
   2. Can some code be generated to place ***instructions*** (exploit code) on the stack?
   3. What is the calling convention used for x86 - 32bit systems (**CDECL**)?
2. How can the EIP be redirected to the stack?
3. How can the WinExec shellcode be called?

**Submission:**

1. Screen capture of your exploit working (**5pts**)
2. Code required and used; includes any of (shellcode, C code, PyCommand) used. (**5pts**)
3. Screen capture of the stack showing shellcode. (**5pts**)

## Questions 6

1. What does the option **-fno-stack-protector** do? (**5pts**)
2. What is shellcode? (**5pts**)
3. Is shellcode always malicious? (**5pts**)
4. What Operating System feature(s) make it difficult for malicious code to execute? (**5pts**)
5. Why (is/are) the OS feature(s) able to stop malicious code from executing? (**5pts**)