

IDA Python Script for Function Tracking and Breakpoint Setting

Shane Irons

CAP 5137 Software Reverse Engineering and Malware Analysis – Term Project

Dr. Xiuwen Liu

12/9/2022

Abstract

As a new user of IDA and practitioner of malware analysis, it is easy to get lost in a program with the many functions that may reside amongst the instructions. Knowing which functions are called at runtime can be an important step in analyzing a potentially malicious file. This paper presents an IDA Python script that tracks function execution, specifically by setting breakpoints at the initial instruction of each function and coloring said function if the breakpoint is tripped. The script was written in IDA using the Python language importing *idaapi* and *idautils* to successfully color functions and explicitly show a path of execution. The script performs well, however, testing shows that it has trouble coloring functions if they are not executed at runtime. This paper introduces the problem and description, then details the techniques in the script, and discusses the results as well as shows screenshots meant to be a demonstration. The script files are attached at the end of the document for individual testing and further investigation.

I. Project Proposal and Description

In class CAP 5137 Fall22, we have used IDA for static analysis mainly on the homework program requiring input to defuse the bomb. This program had tons of functions, so it got confusing determining what was going to happen next. The goal of this project is to write a script in IDA that tracks executed functions, making static analysis much easier. A VM with IDA installed as well as the malicious program used in the homework for testing and development purposes are configured. Using the homework program, the goal is to be able to better track function execution paths, possibly detecting the phases in order.

Regarding the VM, VMware Workstation Pro is being used with a Windows 10 VM. This software is on the free trial, if it ends during the project, there are saved copies of the script and another VM on VirtualBox to be used as a backup with the same configurations. This is a solo project so the script will not be extremely complex, however, it does address a problem I personally had in the homework in that there are so many functions that IDA displays it is easy to get lost when performing static analysis. This script will help newer IDA users like myself when following along the execution path of functions.

The script is written in Python language and straightforwardly adds breakpoints to determined functions and colors them as they are executed when the program is run. This allows the user to see exactly what functions are being called during program execution.

II. Techniques

The script for this project has gone through two versions to achieve functionality. In version 1 (Figures 1 and 2), errors were stopping execution namely by an undefined *idautils*. This was not the only issue, however, as my first version set breakpoints for itself but did not remove them. This meant that the script was messy and left the program altered with tons of breakpoints all over the functions. In version 2 (working version; Figure 3), I import *idautils* at the beginning of the file and add a function *bp_delete* at the end of the function definitions. The following are a description of my created functions in the script:

a. *bp_MainFunction*

This function does not take a condition in its definition; it serves as the “main” function on line 6 (Figure 3). Line 7 begins a for loop that looks at all the determined functions from the disassembly. Line 8 is the loop where the script looks at the entire function where the breakpoint was set and applies a color if the breakpoint is triggered by calling *color_function*. Line 8 takes the parameter (*function*) that refers to the disassembled function shown in IDA.

b. *color_function*

This function is defined on line 10 and takes a condition (*function*) from the *idaapi*. Line 11 calls *idc.set_color* which is an IDA utility command that prepares to set the color given the parameters. Here, *function* as a parameter is given to *set_color* followed by *CIC_FUNC*, and the hex of the color I want to set: 0x43E287 (green). After this is run, *bp_delete* is called (a later defined function) to delete the breakpoint, cleaning up.

c. `bpSetting`

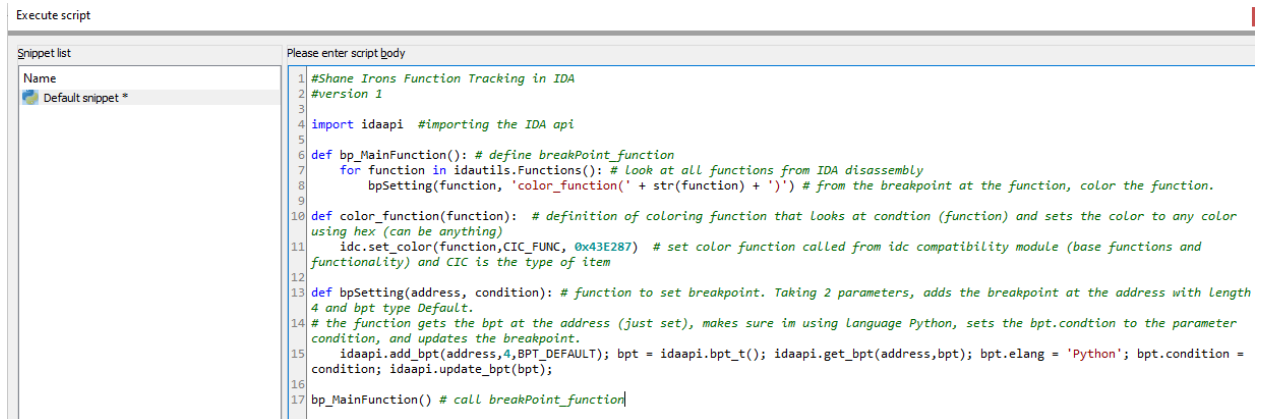
This function definition takes two parameters: *address* and *condition*. Following this is a long string of commands that work to set breakpoints on the functions (lines 14 and 15; Figure 3). First, the function accesses the *idaapi*, adds the breakpoint at the address of the first instruction in each function, with the default size and default breakpoint (software breakpoint). Then, I assign the *idaapi.bpt_t()* call to *bpt* for future use. In this command, *bpt_t* adds trace information and acts as a method. Next, I call *idaapi* again invoking *get_bpt* with the parameters *address* and the previously mentioned *bpt*. *Get_bpt* here again acts as a method in the *idaapi* that gets the *bpt* at the address with the trace information from *bpt_t*. Then, I am making sure the language of the breakpoint is interacting with Python language by using *bpt.elang = 'Python'*. Following this, I set the *bpt.condition* to the condition from the function definition parameter. Finally, I update the breakpoint with the parameter of *bpt* which contains trace information about the function.

d. `bp_delete`

This is a simple function at line 17 that takes the parameter *address* and calls the *idaapi* to delete the breakpoint at that address. This is called in *color_function* so that after the function is colored, the breakpoint can be deleted and allow the program to run smoothly. The script would work the same without this function, however, there would be tons of breakpoints in the program, and you would have to manually click through them.

In summary, techniques involving the IDA API that is imported at the start of the script as well as IDA UTILS (also imported; Figure 3) are used to set breakpoints on the first instruction of each function in the program. Then, when the program is run, the script looks at any function where the breakpoint is tripped, it colors that function green, and it removes the breakpoint. With this technique, all the called functions are colored green and other functions that remain uncalled will have a breakpoint on the first instruction, but no color. This is useful for static analysis because the user can note which functions exactly are called when the program is executed (in a safe environment) for a general understanding of its execution path.

III. Results and Demonstration

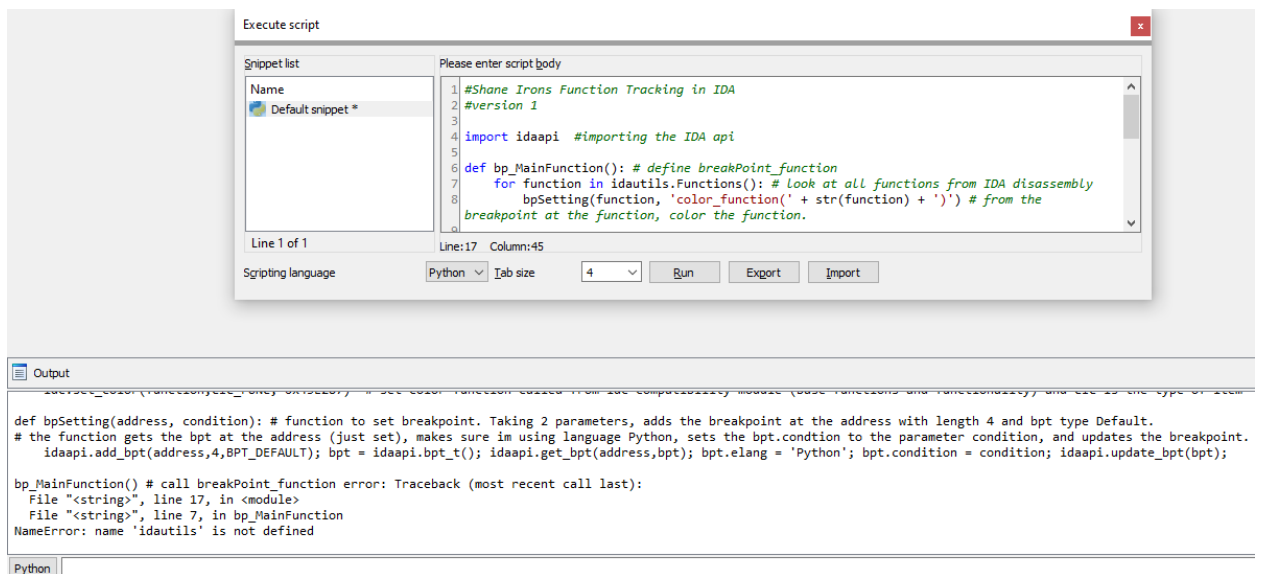


The screenshot shows the 'Execute script' dialog box with a snippet list on the left and a text area for the script body on the right. The script body contains the following Python code:

```

1 #Shane Irons Function Tracking in IDA
2 #version 1
3
4 import idaapi #importing the IDA api
5
6 def bp_MainFunction(): # define breakPoint_function
7     for function in idautils.Functions(): # Look at all functions from IDA disassembly
8         bpSetting(function, 'color_function(' + str(function) + ')') # from the breakpoint at the function, color the function.
9
10 def color_function(function): # definition of coloring function that looks at condition (function) and sets the color to any color
11     # using hex (can be anything)
12     ida.set_color(function, CIC_FUNC, 0x43E287) # set color function called from ida compatibility module (base functions and
13     # functionality) and CIC is the type of item
14
15 def bpSetting(address, condition): # function to set breakpoint. Taking 2 parameters, adds the breakpoint at the address with length
16     # 4 and bpt type Default.
17     # the function gets the bpt at the address (just set), makes sure im using language Python, sets the bpt.condition to the parameter
18     # condition, and updates the breakpoint.
19     idaapi.add_bpt(address, 4, BPT_DEFAULT); bpt = idaapi.bpt_t(); idaapi.get_bpt(address, bpt); bpt.elang = 'Python'; bpt.condition =
20     condition; idaapi.update_bpt(bpt);
21
22 bp_MainFunction() # call breakPoint_function
  
```

Figure 1: Version 1 of function tracking script



The screenshot shows the 'Execute script' dialog box with the same script code as in Figure 1. Below the script body, there is an 'Output' section showing the following error message:

```

def bpSetting(address, condition): # function to set breakpoint. Taking 2 parameters, adds the breakpoint at the address with length 4 and bpt type Default.
# the function gets the bpt at the address (just set), makes sure im using language Python, sets the bpt.condition to the parameter condition, and updates the breakpoint.
    idaapi.add_bpt(address, 4, BPT_DEFAULT); bpt = idaapi.bpt_t(); idaapi.get_bpt(address, bpt); bpt.elang = 'Python'; bpt.condition = condition; idaapi.update_bpt(bpt);

bp_MainFunction() # call breakPoint_function error: Traceback (most recent call last):
  File "<string>", line 17, in <module>
  File "<string>", line 7, in bp_MainFunction
NameError: name 'idautils' is not defined
  
```

Figure 2: Output from running version 1 of the script

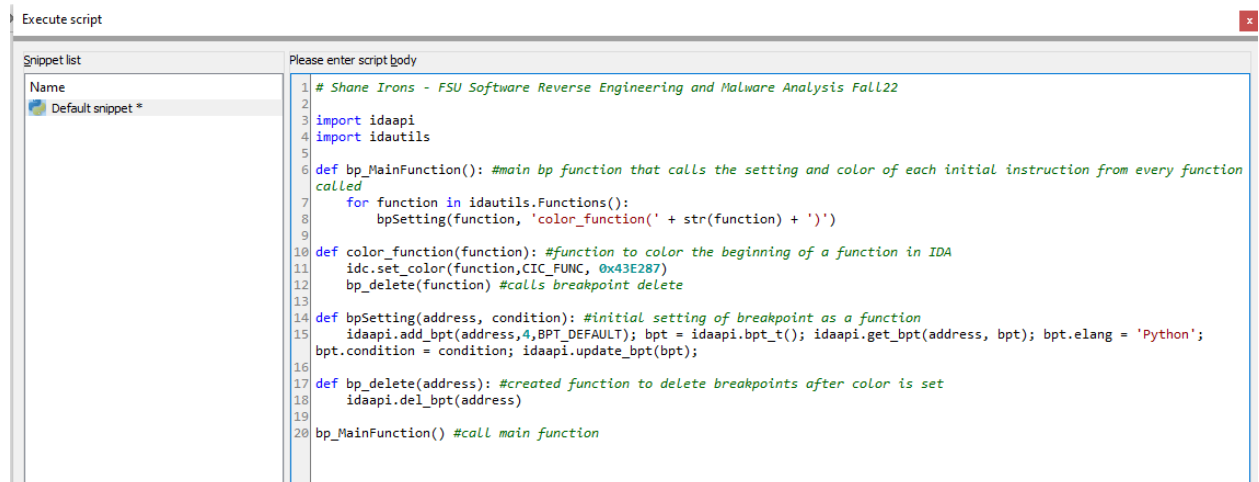


Figure 3: Version 2 of Function Highlighting for Visibility Script

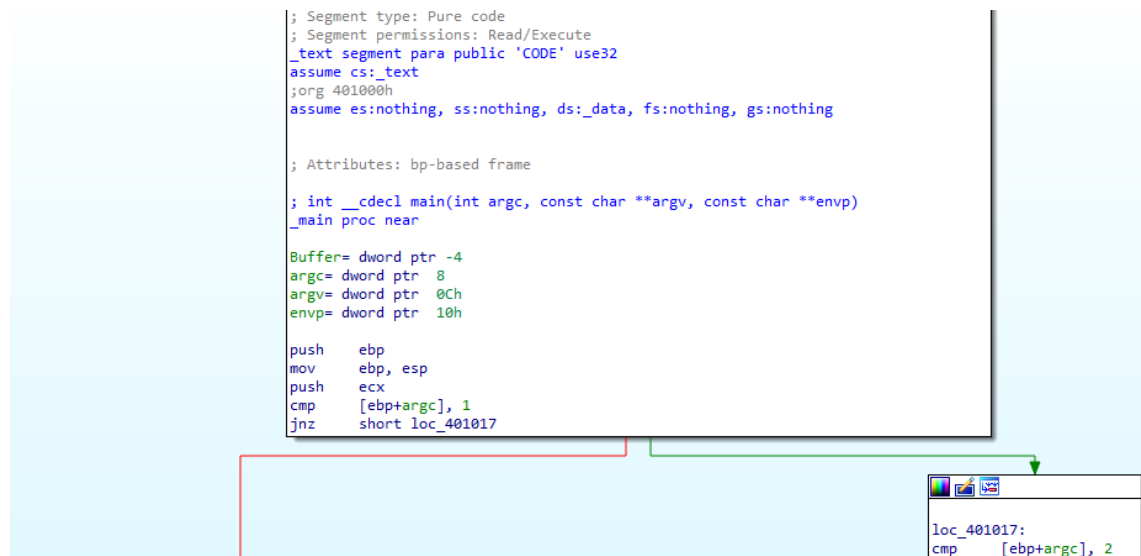


Figure 4: IDA Pro disassembly screenshot showing the beginning of the file

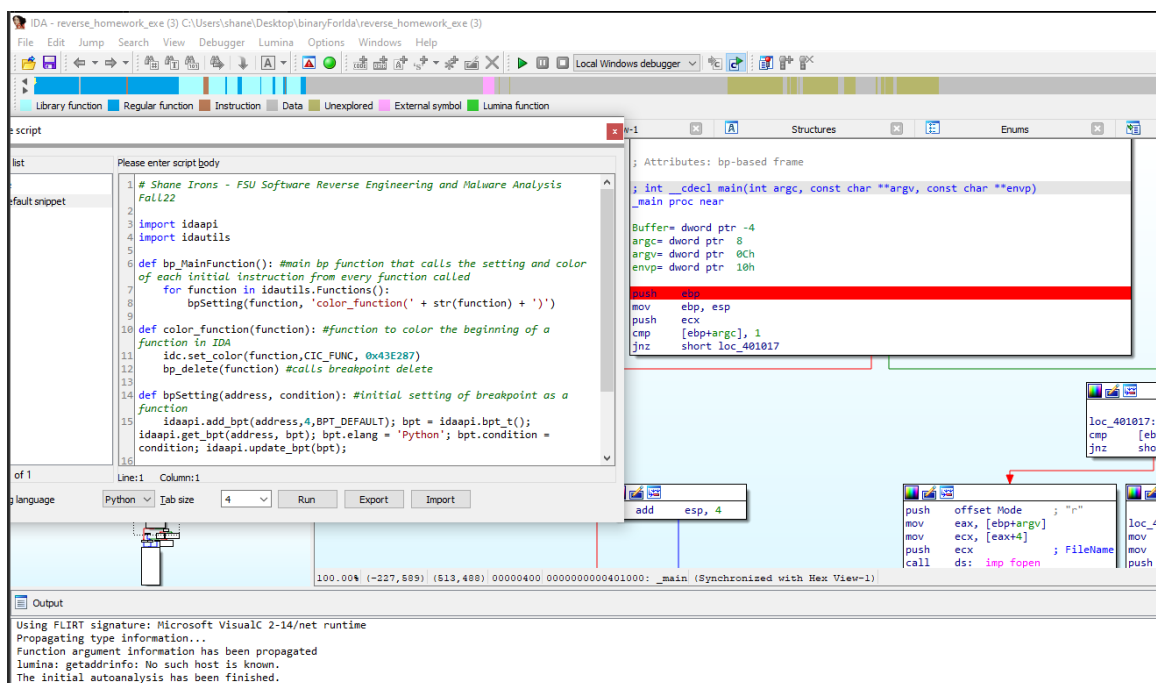


Figure 5: Aftermath of running the script version 2 (breakpoint)

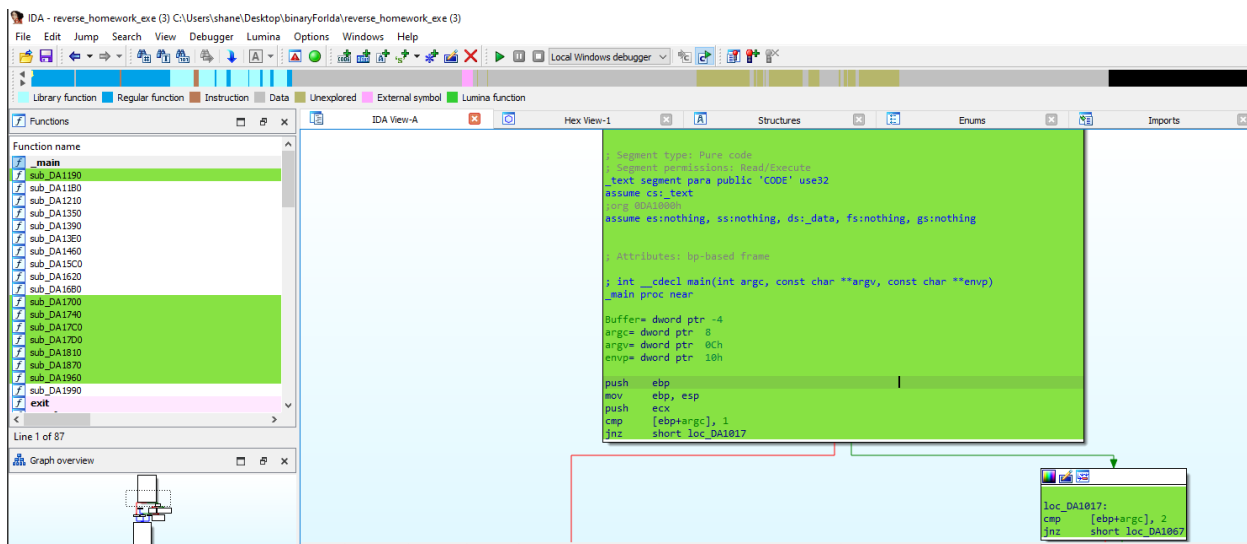


Figure 6: Running “reverse_homework_exe” file with script version 2

Function name	Segment	Start	Length	Locals	Arguments	R	F	L	M	O	S	B	T	=
terminate(void)	.text	0000000000A21B8A	00000006			R	-	L	-	-	-	-	T	-
sub_DA1208	.text	0000000000A21D8	0000000F	00000000		R	-	-	-	-	-	-	-	-
sub_DA1209	.text	0000000000A21E9	00000003	00000000		R	-	-	-	-	-	-	-	-
sub_DA1E98	.text	0000000000A1E98	00000026	00000008		R	-	-	-	-	-	T	-	-
sub_DA1E72	.text	0000000000A1E72	00000026	00000008		R	-	-	-	-	-	-	-	-
sub_DA129C	.text	0000000000A129C	00000006	00000000		R	-	-	-	-	-	-	-	-
sub_DA1990	.text	0000000000A1990	00000078	00000060		R	-	-	-	-	-	B	-	-
sub_DA1960	.text	0000000000A1960	00000027	00000004		R	-	-	-	-	-	B	-	-
sub_DA1870	.text	0000000000A1870	00000089	00000010		R	-	-	-	-	-	B	-	-
sub_DA1810	.text	0000000000A1810	00000053	00000008		R	-	-	-	-	-	-	-	-
sub_DA1700	.text	0000000000A1700	00000040	00000008	00000004	R	-	-	-	-	-	B	-	-
sub_DA17C0	.text	0000000000A17C0	00000055	00000004		R	-	-	-	-	-	B	-	-
sub_DA1740	.text	0000000000A1740	00000074	00000010	00000008	R	-	-	-	-	-	B	-	-
sub_DA1700	.text	0000000000A1700	00000038	0000000C	00000004	R	-	-	-	-	-	B	-	-
sub_DA1680	.text	0000000000A1680	0000004F	00000008	00000008	R	-	-	-	-	-	B	T	-
sub_DA1620	.text	0000000000A1620	00000068	00000010		R	-	-	-	-	-	-	-	-
sub_DA15C0	.text	0000000000A15C0	0000005A	00000004	00000008	R	-	-	-	-	-	B	-	-
sub_DA1460	.text	0000000000A1460	00000138	00000044	00000004	R	-	-	-	-	-	B	T	-
sub_DA13C0	.text	0000000000A13C0	00000075	00000014	00000008	R	-	-	-	-	-	B	-	-
sub_DA1390	.text	0000000000A1390	0000004E	00000010	00000004	R	-	-	-	-	-	B	T	-
sub_DA1330	.text	0000000000A1330	00000036	00000008	00000004	R	-	-	-	-	-	B	-	-
sub_DA1210	.text	0000000000A1210	00000117	0000001C	00000004	R	-	-	-	-	-	B	T	-
sub_DA1180	.text	0000000000A1180	00000098	00000024	00000004	R	-	-	-	-	-	B	T	-
sub_DA1190	.text	0000000000A1190	0000001F	00000004	00000004	R	-	-	-	-	-	B	-	-
strdup	.text	0000000000A1A3C	00000006	00000000	00000004	R	-	-	-	-	-	-	-	-
start	.text	0000000000A1D50	0000000A	00000000		R	-	L	-	-	-	-	-	-
scanf	.text	0000000000A1A24	00000006	00000000	00000008	R	-	-	-	-	-	T	-	-
printf	.text	0000000000A1A12	00000006	00000000	00000004	R	-	-	-	-	-	T	-	-
isspace	.text	0000000000A1A30	00000006	00000000	00000004	R	-	-	-	-	-	T	-	-
getenv	.text	0000000000A1A42	00000006	00000000	00000004	R	-	-	-	-	-	T	-	-
fopen	.text	0000000000A1A18	00000006	00000000	00000008	R	-	-	-	-	-	T	-	-
fgets	.text	0000000000A1A36	00000006	00000000	0000000C	R	-	-	-	-	-	T	-	-
exit	.text	0000000000A1A0C	00000006	00000000	00000004	R	-	-	-	-	-	T	-	-
atoi	.text	0000000000A1A2A	00000006	00000000	00000004	R	-	-	-	-	-	T	-	-
ungetc	.text	0000000000A11C3	00000006	00000000		R	-	-	-	-	-	-	-	-
printf_stdio_int	.text	0000000000A1A4F	0000004B	00000004		R	-	L	-	-	-	-	-	-
printf_stdio_int	.text	0000000000A1C3F	000000E1	00000000		R	-	L	-	-	-	-	-	-
main	.text	0000000000A1A09	00000114	00000008	0000000C	R	-	-	-	-	-	B	T	-
_lock	.text	0000000000A12CC	00000006	00000000		R	-	-	-	-	-	-	-	-
invoke_watson	.text	0000000000A12EE	00000006	00000000	00000014	R	-	-	-	-	-	T	-	-
_initterm_xc	.text	0000000000A2024	00000006	00000000	00000008	R	-	-	-	-	-	T	-	-
_initterm	.text	0000000000A201E	00000006	00000000	00000008	R	-	-	-	-	-	T	-	-
_exit	.text	0000000000A12E4	00000006	00000000	00000004	R	-	-	-	-	-	T	-	-
_except_handler4_common	.text	0000000000A12E8	00000006	00000000		R	-	-	-	-	-	-	-	-
_encode_pointer	.text	0000000000A2118	00000006	00000000		R	-	-	-	-	-	-	-	-
_decode_pointer	.text	0000000000A21D2	00000006	00000000		R	-	-	-	-	-	-	-	-
_crt_debugger_hook	.text	0000000000A2200	00000006	00000000		R	-	-	-	-	-	-	-	-
_controlfp_s	.text	0000000000A21F4	00000006	00000000	0000000C	R	-	-	-	-	-	T	-	-
_configthreadlocale	.text	0000000000A20AA	00000006	00000000	00000004	R	-	-	-	-	-	T	-	-
_cexit	.text	0000000000A12E6	00000006	00000000		R	-	-	-	-	-	T	-	-
_atexit	.text	0000000000A1258	00000017	00000004	00000004	R	-	L	-	-	-	B	T	-
_amsg_exit	.text	0000000000A1DAA	00000006	00000000		R	-	-	-	-	-	-	-	-

Figure 7: Showcasing highlighted functions after program was run

Function name	Segment	Start	Length	Locals	Arguments	R	F	L	M	O	S	B	T	=
f WinMain(x,x,x,x)	.text	0000000000401000	00000090	00000020	00000010	R	T	.
f start	.text	0000000000401090	000000F6	00000078		.	.	L	.	.	.	B	.	.
f _amsg_exit	.text	0000000000401186	00000025	00000000	00000004	.	.	L	T	.
f _fast_error_exit	.text	00000000004011AB	00000023	00000000	00000004	.	.	L	.	.	S	.	T	.
f nullsub_1	.text	00000000004011CE	00000001	00000000		R
f _cinit	.text	00000000004011CF	0000002D	00000000		R	.	L
f _exit	.text	00000000004011FC	00000011	00000000	00000004	.	.	L	T	.
f _doexit	.text	000000000040120D	00000011	00000000	00000004	.	.	L	T	.
f _initterm	.text	000000000040121E	00000099	00000004	0000000C	R	.	L	.	.	S	.	T	.
f _xcpfilter	.text	00000000004012B7	0000001A	00000004	00000008	R	.	L	.	.	S	.	T	.
f _xcplookup	.text	00000000004012D1	00000141	00000008	00000008	R	.	L	.	.	.	B	T	.
f _winmdln	.text	0000000000401412	00000043	00000004	00000004	R	.	L	.	.	S	.	.	.
f _setenvp	.text	0000000000401455	00000058	00000000		R	.	L
f _setargv	.text	00000000004014AD	000000B9	0000000C		R	.	L
f _parse_cmdline	.text	0000000000401566	00000099	00000018		R	.	L	.	.	.	B	.	.
f _crtGetEnvironmentStringsA	.text	00000000004015FF	000001B4	00000010	00000014	R	.	L	.	.	S	B	.	.
f _joinit	.text	00000000004017B3	00000132	00000018		R	.	L
f _heap_init	.text	00000000004018E5	000001AB	00000054		R	.	L
f _global_unwind2	.text	0000000000401A90	0000003C	00000000	00000004	R	.	L
f _unwind_handler	.text	0000000000401ACC	00000020	00000014	00000004	R	.	L	.	.	.	B	T	.
f _local_unwind2	.text	0000000000401AEC	00000022	00000000	00000010	R	.	L	.	.	S	.	.	.
f _abnormal_termination	.text	0000000000401B0E	00000068	00000014	00000008	R	.	L
f _at_done	.text	0000000000401B76	00000022	00000000		R	.	L	.	.	.	T	.	.
f _NLG_Notify1	.text	0000000000401B98	00000001	00000000		R	.	L	.	.	S	.	.	.
f _NLG_Notify	.text	0000000000401B99	00000009	00000004	00000004	R	.	L
f _except_handler3	.text	0000000000401BA2	00000018	00000004	00000004	R	.	L
f _seh_jongjmp_unwind(x)	.text	0000000000401BC4	000000BD	0000001C	0000000C	R	.	L	.	.	.	B	T	.
f _FF_MSGBANNER	.text	0000000000401C81	0000001B	00000004	00000004	R	.	L
f _NMSG_WRITE	.text	0000000000401C9C	00000039	00000000		R	.	L
f _ismbblead	.text	0000000000401CD5	00000153	000001AB	00000004	R	.	L	.	.	.	B	T	.
f _x_ismbctype	.text	0000000000401E28	00000011	00000000	00000004	R	.	L	T	.
f _setmbcp	.text	0000000000401E39	00000031	00000000	00000009	R	.	L	.	.	S	.	.	.
f _getSystemCP	.text	0000000000401E6A	00000199	00000028	00000004	R	.	L	.	.	.	B	T	.
f _CPtoLCD	.text	0000000000402003	0000004A	00000000	00000004	R	.	L	.	.	S	.	.	.
f _setSCS	.text	000000000040204D	00000033	00000000	00000004	R	.	L	.	.	S	.	.	.
f _setSBUPLow	.text	0000000000402080	00000029	00000004		R	.	L	.	.	S	.	.	.
f _initmbctable	.text	00000000004020A9	00000185	0000051C		R	.	L	.	.	S	B	.	.
f _free	.text	000000000040222E	0000001C	00000000		R	.	L
f _strcpy	.text	000000000040224A	0000002F	00000004	00000004	R	.	L	T	.
f _strcat	.text	0000000000402280	00000007	00000004	00000008	R	.	L	T	.
f _malloc	.text	0000000000402290	000000E0	00000004	00000008	R	.	L	T	.
f _rh_malloc	.text	0000000000402370	00000012	00000000	00000004	R	.	L	T	.
f _heap_alloc	.text	0000000000402382	0000002C	00000000	00000008	R	.	L	T	.
f _strlen	.text	00000000004023AE	00000036	00000004	00000004	R	.	L	T	.
f _memcpy	.text	00000000004023F0	0000007B	00000000	00000004	R	.	L	T	.
f _sbh_heap_init	.text	0000000000402470	00000335	0000000C	0000000C	R	.	L	.	.	.	B	T	.
f _sbh_find_block	.text	00000000004027A5	0000003E	00000000		R	.	L
f _sbh_free_block	.text	00000000004027E3	0000002B	00000000	00000004	R	.	L
f _sbh_alloc_block	.text	000000000040280E	0000032B	00000024	00000008	R	.	L	.	.	.	B	.	.
f _sbh_alloc_new_region	.text	0000000000402839	00000309	00000024	00000004	R	.	L	.	.	.	B	.	.
f _sbh_alloc_new_group	.text	0000000000402E42	000000B1	00000008		R	.	L
f _rtMessageBoxA	.text	0000000000402EF3	000000FB	00000014	00000004	R	.	L	.	.	.	B	.	.
f _rtMessageBoxA	.text	0000000000402FEE	00000089	0000000C	0000000C	R	.	L

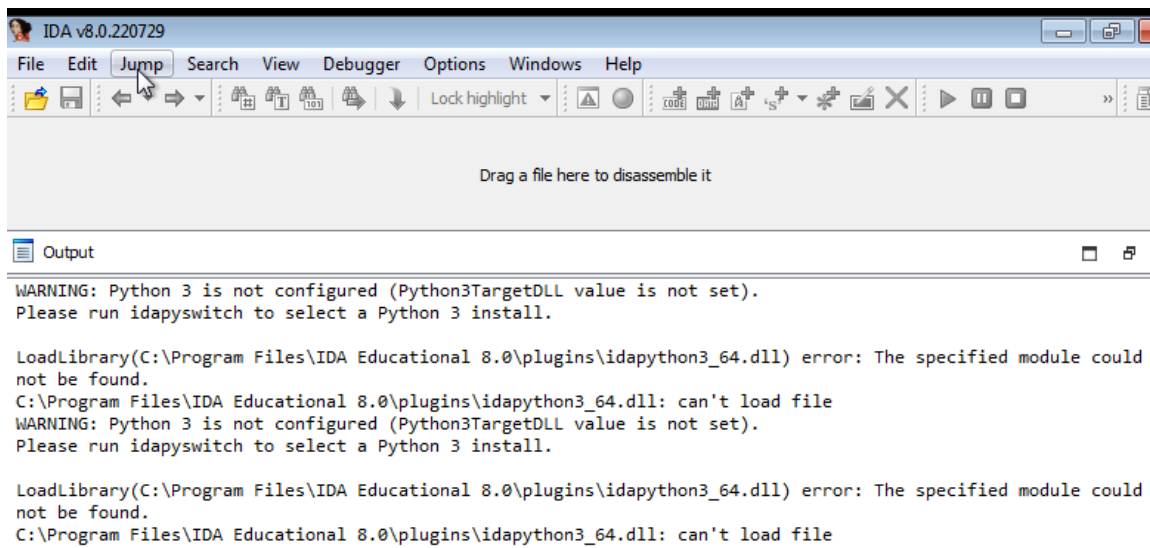


Figure 9: Windows 7 VM IDA PRO installation not working correctly with Python

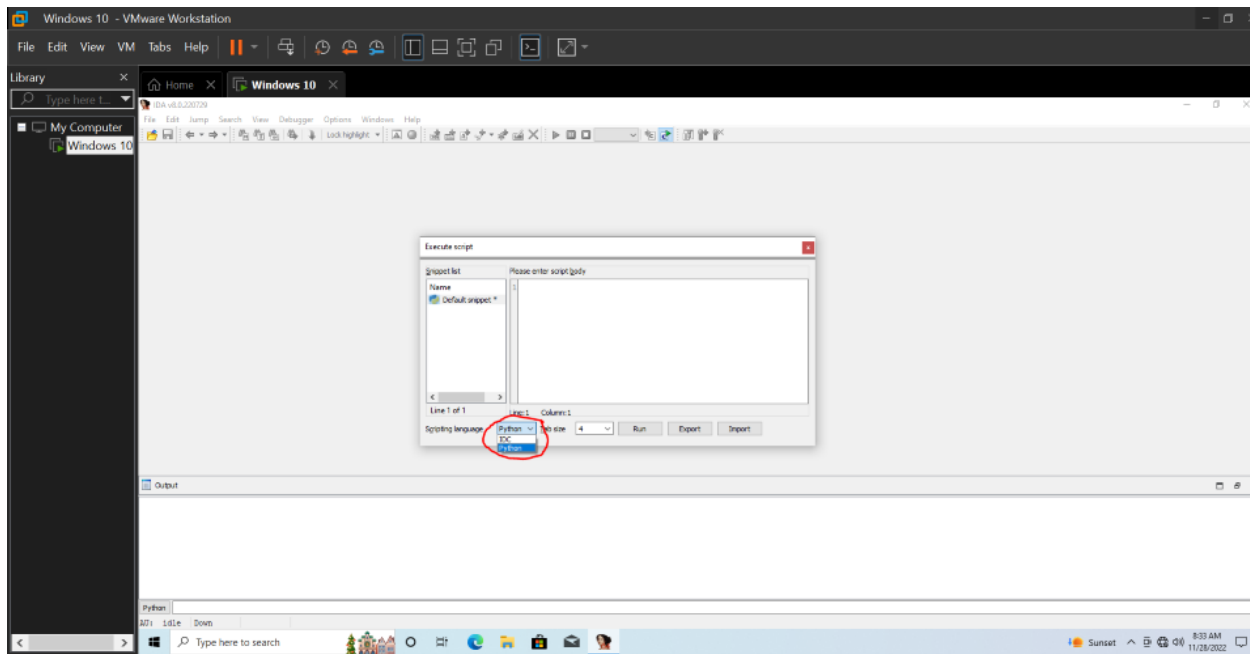


Figure 10: IDA installed on Windows 10 VM working fine

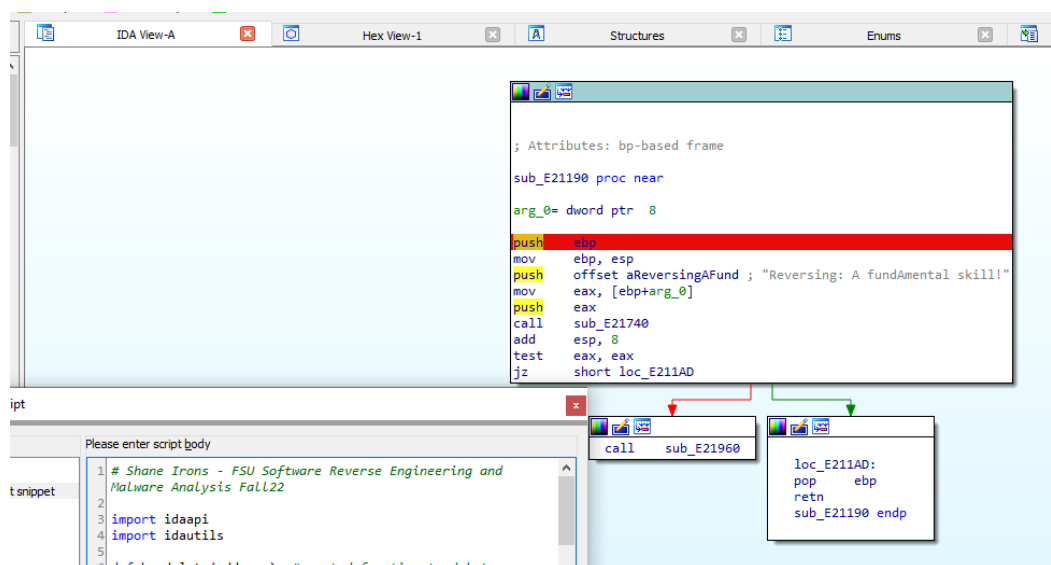


Figure 11: Issues with highlighting every used function as intended

Above are screenshots to share my demonstration of the script running in the safe Windows 10 VM. These figures serve to act as a showcase of the scripts' functionality. Before I began writing the script, my initial problems were encountered when setting up the originally planned Windows 7 VM. As seen in Figure 7, IDA Pro on Windows 7 was not installing Python correctly, so I could not start working on the script. I had tried to fix this multiple ways, but I gave up on this VM and configured a Windows 10 machine on VMWare Workstation Pro. As seen in Figure 8, this was working properly and I could start the project.

The 1st version of the script (Figure 1) was barebones and did not have the functionality of version 2 (Figure 3). A major issue that I had to change between versions was the inclusion of `"import idautils"` at the beginning of the file. From version 1, I kept a lot of the functions the same. I added the definition `"bp_delete"` and called it in the `"color_function"` function so that the script gets rid of the breakpoints and color the function. If this function is commented out, I can use the step over functionality in IDA Pro to manually step through the program and still color the functions without removing the breakpoints. This script is designed to color a function after the breakpoint is moved through, so without stepping through a breakpoint or having the `bp_delete` function, the script does not work as intended.

Additional issues were encountered with the script. It does not seem to pick up every function as desired (Figure 9). While other functions were highlighted, the first stage password (which I passed for testing) did not get highlighted, but does show a breakpoint. I got the second stage wrong and blew the bomb up intentionally and this function remained unchanged. This can be an issue as this is an important function that houses the phrase needed to pass phase one. Programs with tons of functions that are not called at runtime, or similarly, functions that require user input (like in Figure 11) can confuse this script. Following along the breakpoints set by the script can lead you here, but it seems the highlighting feature does not work fully. This would be a major issue that needs to be addressed in future iterations.

The script was tested on another program from the Malware folder: the malicious application from lab 10-01 in the book. 10-01 executable was loaded into IDA, the script was run, then the program was executed and the results are seen in Figure 8. The functions being called while the program runs are

highlighted. This would lead me to important areas in the program during initial analysis.

IV. Conclusion

The purpose of this project was to create a script that tracks function execution by coloring said function after a breakpoint is tripped. This script was written for IDA Pro in Python language and functions as intended, but not fully as desired. Issues regarding functions not executed at runtime or functions requiring user input (Figure 11) seem to cause issues for the overall performance of this project. A future iteration of this script needs to address this issue as well as the added functionality of showing the order of functions being executed for a better understanding of the path of execution.

V. Programs and Scripts



funcHigh.py

Version 1:



funcHighver2.py

Version 2:

Attached above to this document are my .py files for the script. It includes both versions 1 and 2, although version 2 is the only functioning one. I included version 1 as a comparison and for documentation. They can be loaded into IDA by selecting File, Script Command, Import, then selecting one of the file versions. Also, make sure to change Scripting language to Python (it defaults as IDC; See Figure 10 as reference).

VI. Resources

- a. Hex-Rays (2021). Module idc. Introduction to the idc module and the different functions to be called from the imported idaapi. https://www.hex-rays.com/products/ida/support/idadpython_docs/idc.html
- b. Grunzweig, J. (2015). Using IDAPython to Your your Life Easier: Part 1. Initial introduction to IDAPython scripting and tutorial. <https://unit42.paloaltonetworks.com/using-idapython-to-make-your-life-easier-part-1/>
- c. Scags (2022). Random IDA scripts. Reference for util and api calls as well as formatting for IDA scripts in general. <https://github.com/Scags/IDA-Scripts>