In Memory Fuzzing

Real Time Input Tracing and In Memory Fuzzing

by sinn3r – twitter.com/_sinn3r

2010

Introduction

In memory fuzzing is a technique that allows the analyst to bypass parsers; network-related limitations such as max connections, buit-in IDS or flooding protection; encrypted or unknown (poorly documented) protocol in order to fuzz the actual underlying assembly routines that are potentially vulnerable. This concept will be explained more in the InMemoryFuzzer.py section, but if you're really interested in it, Fuzzing: Brute Force Vulnerability Discovery by Pedram Amini is a great book to read.

Prior to the development of my fuzzing toolset, I was unsatisfied (for now) with all the publicly available in memory fuzzers, because most of them are just too basic and require too much prep time in advance – flow analysis, reverse code engineering, etc – which obviously has a high learning curve and time consuming tasks, and most people would rather just stick with traditional fuzzers (which usually can accomplish the exact same thing). Yes, you DO need some reversing skills to make in memory fuzzing useful, but honestly it doesn't really have to be that difficult to start fuzzing and find bugs... as long as you have the right approach.

One of the approaches we do here is by tracing user input automatically at real time, and log all the important functions that process that input, and then fuzz them. A proof of concept (Tracer.py and InMemoryFuzzer.py) is also available to download which can be found here:

http://redmine.corelan.be:8800/projects/inmemoryfuzzing

A video demonstration is also available here that shows how to use in memory fuzzing:
http://www.youtube.com/watch?v=YhyFuAfD7C4

Special thanks to:

- Peter Van Eeckhoutte, and members of Corelan Security
- Offensive Security Exploit Database
- dookie2000ca for all the feedback

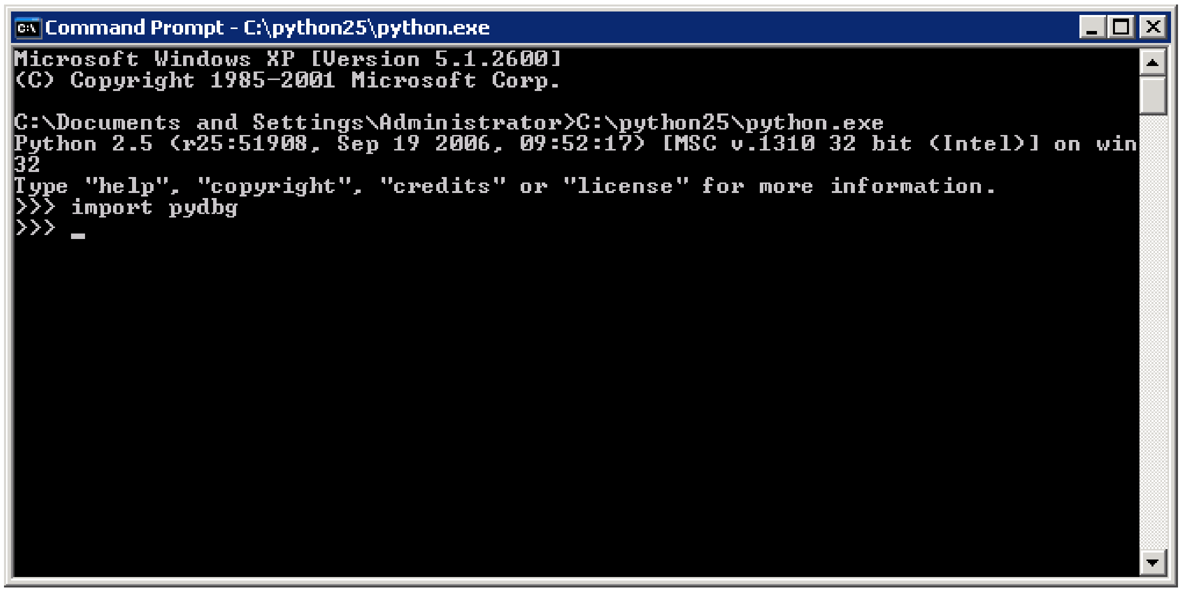
Requirements/Setup:

In order to use these tools, you should have:

- Windows XP SP2 or SP3 (Not tested on SP1), or newer.
- IDA 4.9: http://www.hex-rays.com/idapro/idadownfreeware.htm OR pvefindaddr.py (optional)
- Python 2.5.0 (installed from Immunity Debugger)
- Pydasm: http://therning.org/magnus/archives/278
- Paimei: http://www.openrce.org/downloads/details/208/PaiMei

Pydbg is probably the trickiest to install so we'll go through the steps briefly:

1. Install Python 2.5. The one I tested was: Python 2.5 (r25:51908, Sep 19 2006, 09:52:17)
2. Download Pydasm (for Python 2.5) from the URL above.
3. Download Paimei. Extract the package, go to the "installers" folder, and run the installer.
4. Remove C:\Python25\Lib\site-packages\pydbg\pydasm.pyd
5. Now you're ready to test out Pydbg. Open command prompt, do the following – if you see no errors after importing pydbg, that means your system now supports Pydbg:



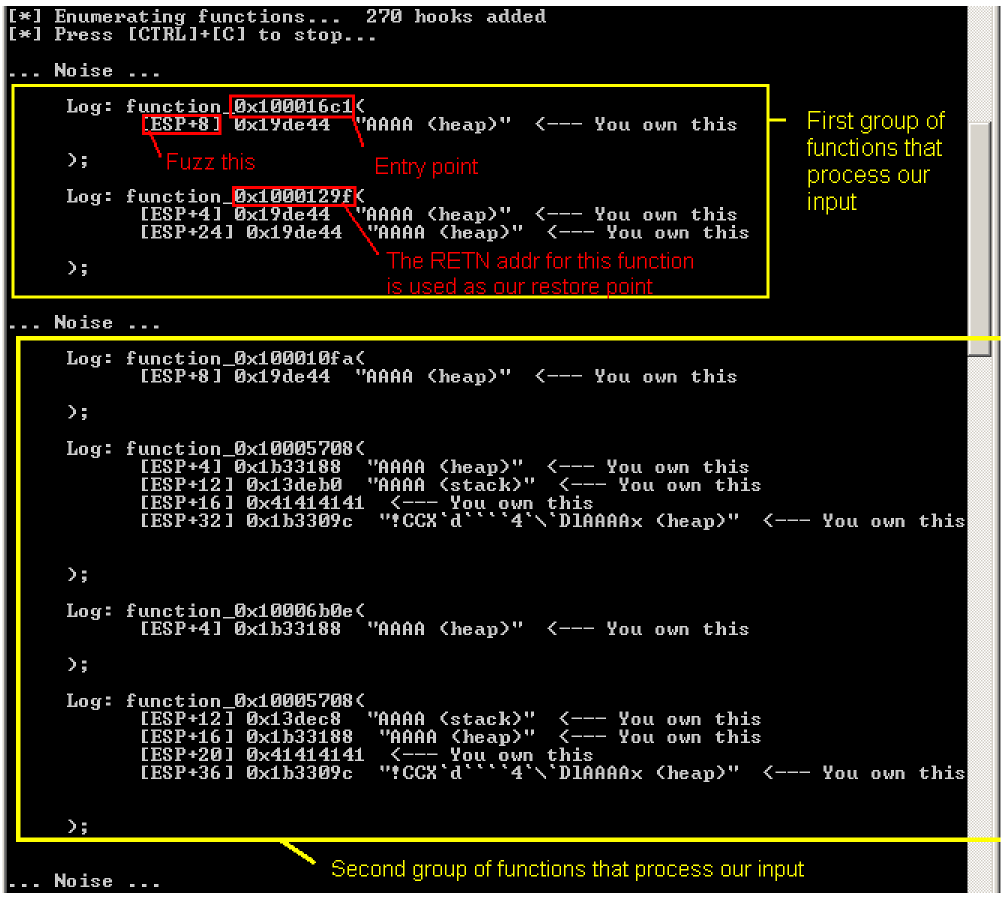
```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>C:\python25\python.exe
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> import pydbg
>>> _
```

Tracer.py: How it works

As I previously mentioned in the introduction, in order to deploy an in memory fuzzer, you must go through a good amount of analysis to identify all the functions that process your input, and log the function entry addresses, RETN addresses, and the argument you want to fuzz. This makes fuzzing very time consuming, simply not something that can be done in minutes.

The purpose of Tracer.py is to ease off this process, allowing the user to track the control flow and user input automatically at real-time. This is done by first searching all the function addresses in the application, put a hook point in every one of them, and then start monitoring. If a hooked function is detected, we log the function address and the argument, and keep listening. Since this happens at real time, even with the most basic tool like this can still see some kind of pattern in the log, which gives us an idea where to fuzz.

The following example shows how to recognize this pattern in Tracer.py:



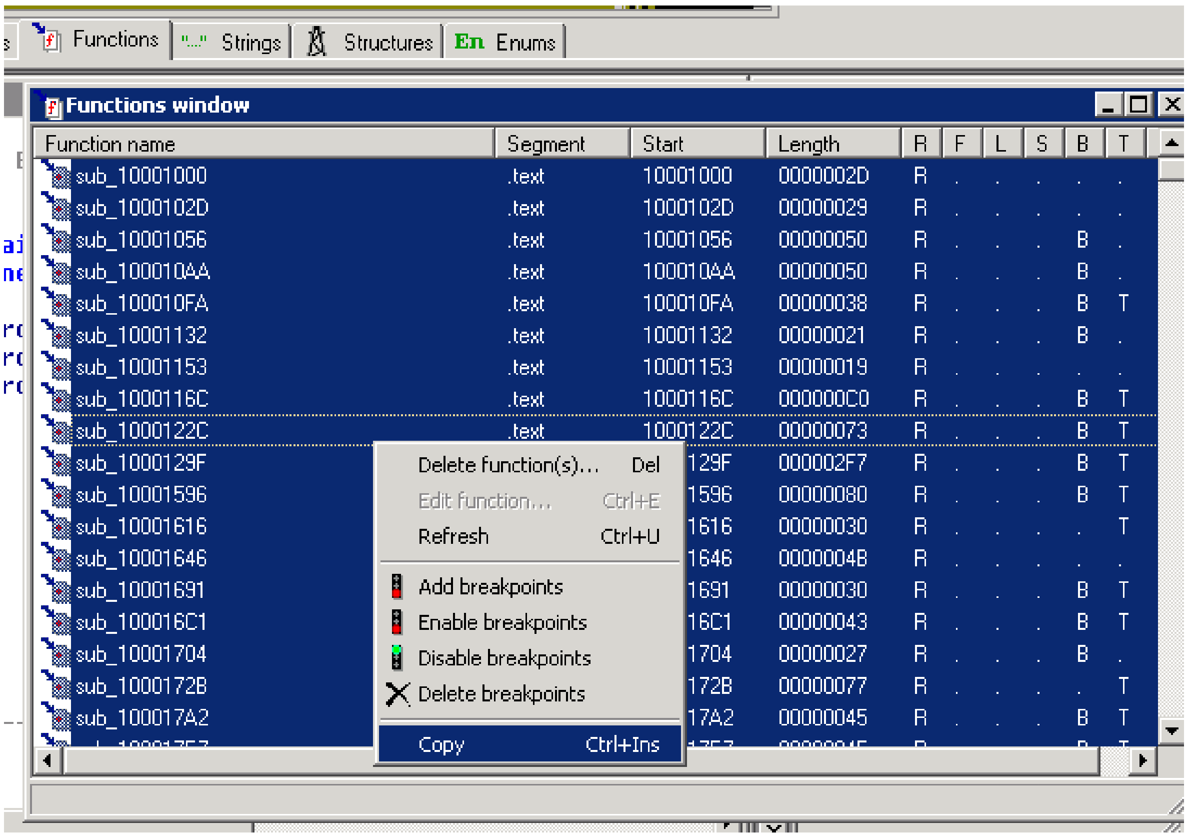
```
[*] Enumerating functions...  270 hooks added
[*] Press [CTRL]+[C] to stop...

... Noise ...

     Log: function_0x100016c1(
           [ESP+8] 0x19de44  "AAAA (heap)"  <--- You own this

     );

     Log: function_0x1000129f(
           [ESP+4] 0x19de44  "AAAA (heap)"  <--- You own this
           [ESP+24] 0x19de44  "AAAA (heap)"  <--- You own this

     );

... Noise ...

     Log: function_0x100010fa(
           [ESP+8] 0x19de44  "AAAA (heap)"  <--- You own this

     );

     Log: function_0x10005708(
           [ESP+4] 0x1b33188  "AAAA (heap)"  <--- You own this
           [ESP+12] 0x13deb0  "AAAA (stack)"  <--- You own this
           [ESP+16] 0x41414141  <--- You own this
           [ESP+32] 0x1b3309c  "!CCX`d```4`\`DlAAAAx (heap)"  <--- You own this


     );

     Log: function_0x10006b0e(
           [ESP+4] 0x1b33188  "AAAA (heap)"  <--- You own this

     );

     Log: function_0x10005708(
           [ESP+12] 0x13dec8  "AAAA (stack)"  <--- You own this
           [ESP+16] 0x1b33188  "AAAA (heap)"  <--- You own this
           [ESP+20] 0x41414141  <--- You own this
           [ESP+36] 0x1b3309c  "!CCX`d```4`\`DlAAAAx (heap)"  <--- You own this


     );

... Noise ...
```

Fuzz this
Entry point
The RETN addr for this function is used as our restore point
First group of functions that process our input
Second group of functions that process our input

Tracer.py: How to

First, open IDA. If you're using IDA 4.9 (see image):

1. Click on the Functions tab.
2. Select all the functions (click the first function → hold [shift] → select last function)
3. Right click → copy → paste on notepad. Save it as "**functions.txt**" under the same directory as the script.



Function name	Segment	Start	Length	R	F	L	S	B	T
sub_10001000	.text	10001000	0000002D	R	.	.	.	.	.
sub_1000102D	.text	1000102D	00000029	R	.	.	.	.	.
sub_10001056	.text	10001056	00000050	R	.	.	.	B	.
sub_100010AA	.text	100010AA	00000050	R	.	.	.	B	.
sub_100010FA	.text	100010FA	00000038	R	.	.	.	B	T
sub_10001132	.text	10001132	00000021	R	.	.	.	B	.
sub_10001153	.text	10001153	00000019	R	.	.	.	.	.
sub_1000116C	.text	1000116C	000000C0	R	.	.	.	B	T
sub_1000122C	.text	1000122C	00000073	R	.	.	.	B	T
sub_1000129F		129F	000002F7	R	.	.	.	B	T
sub_10001596		1596	00000080	R	.	.	.	B	T
sub_10001616		1616	00000030	R	.	.	.	.	T
sub_10001646		1646	0000004B	R	.	.	.	.	.
sub_10001691		1691	00000030	R	.	.	.	B	T
sub_100016C1		16C1	00000043	R	.	.	.	B	T
sub_10001704		1704	00000027	R	.	.	.	B	.
sub_1000172B		172B	00000077	R	.	.	.	.	T
sub_100017A2		17A2	00000045	R	.	.	.	B	T

If you're using IDA Pro 5.5 or higher, the Functions table should be on the left of the pretty graph. You can do the same thing (right click → copy and paste) to obtain all your functions that way. Keep in mind at this stage would be a good time to strip off unnecessary routines that you already know (or add more) from your list to reduce noises during sniffing.

Tip:
If you're not a fan of IDA, you can also use pvefindaddr to automatically generate functions.txt for you. Syntax:

```
!pvefindaddr functions -o -m <module>
```

Second, open the application you want to fuzz. You must do this before running the script because it needs to attach to the process first.

Third, now that you have a function list (functions.txt). Go to command prompt, and type the following (assuming you saved Tracer.py in C:\):

```
C:>C:\python25\python.exe Tracer.py
```

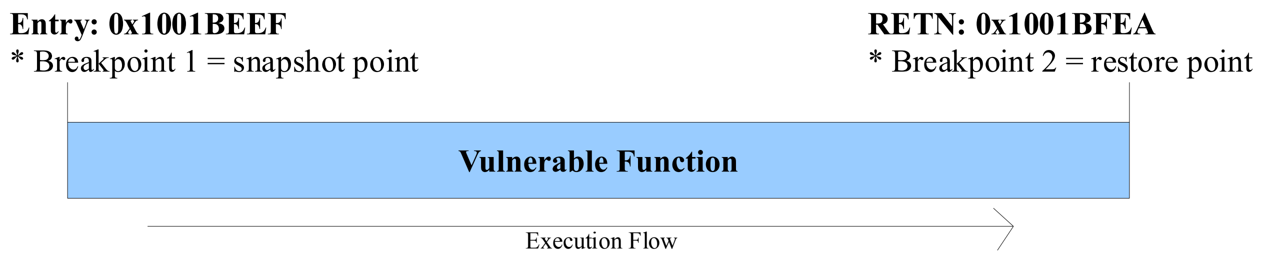
Fourth, the script should find the function list file without problems. Give it a pattern (user input) to look for, select the process you want to monitor, and the fun begins. Note that a file named "new_functions_addrs.txt" will be created – this file contains the same function addresses, and the correct RETN addresses. You can use this as a reference later for InMemoryFuzzer.py.

Fifth, now Tracer.py should be monitoring. Go back to the application, feed it the same pattern (user input), and then you'll see which functions get triggered. Press **[CTRL] + [C]** to terminate the script.

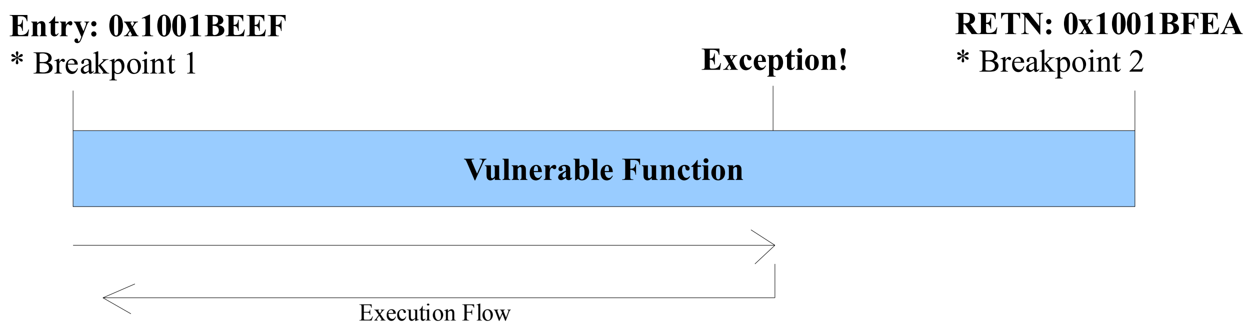
Tip:
It is best to close other unnecessary active processes before running Tracer.py...

InMemoryFuzzer.py: How it works

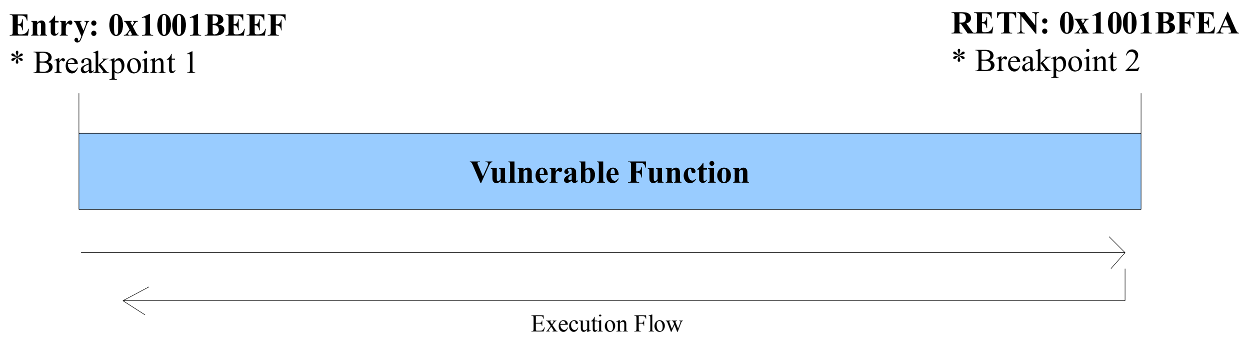
The idea of how the fuzzer works is simple. Say you have a vulnerable routine at entry 0x1001BEEF (aka **snapshot point**), which takes the user input as [ESP+4] at the beginning of the prologue, and that function ends at address 0x1001BFEA (**restore point**). We can put a breakpoint at 0x1001BEEF, another at 0x1001BFEA, and let the application run, as the following diagram demonstrates:



When the execution flow hits our first breakpoint (entry) for the first time, we take a snapshot of the state (threads, stack, registers, flags, etc), modify the user input in [ESP+4], and resume execution to let the function to process our data, and hope something crashes. If an exception is thrown somewhere in the code, we log that, restore the function state, and redirect the execution flow back to the entry (0x1001BEEF), and fuzz again with a new input, like this diagram:



Or, no exception is triggered, we end up hitting the second breakpoint (restore point), then all we have to do is restore the state, rewind, and fuzz again:



InMemoryFuzzer.py: How to

Before you use the fuzzer, you should already know the following:

- Which process to fuzz
- The function entry address(s) (aka your snapshot points)
- The restore point(s) (typically a RETN address)
- Which function argument(s) to fuzz

First thing, open the application you want to fuzz again. And if needed, change how many times you want to fuzz a routine by editing the "maxFuzzCount" global variable in the source code. Please note that InMemoryFuzzer.py has two modes for fuzzing: **Single routine**, or **multiple**.

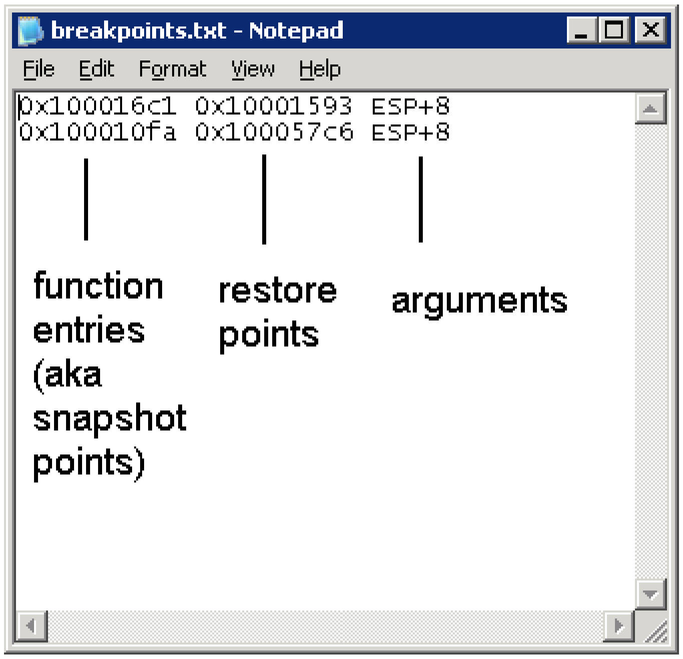
Single routine mode allows the user to put every required information (function entry, restore point, argument) in one line:

```
C:>C:\python25\python.exe InMemoryFuzzer.py <Snapshot point> <Restore point> <Argument>
```

So if we were to reuse the same example in the "How it works" section, we would be feeding the fuzzer with the following:

```
C:>C:\python25\python.exe InMemoryFuzzer.py  0x1001BEEF  0x1001BFEA ESP+4
```

Multiple-Routine mode, which is my favorite mode, does not have to called from command line. All you must do is prepare breakpoints.txt, which contains information such as the snapshot point/restore point/argument with the same format: <snapshot point> <restore point> <argument>. Example:

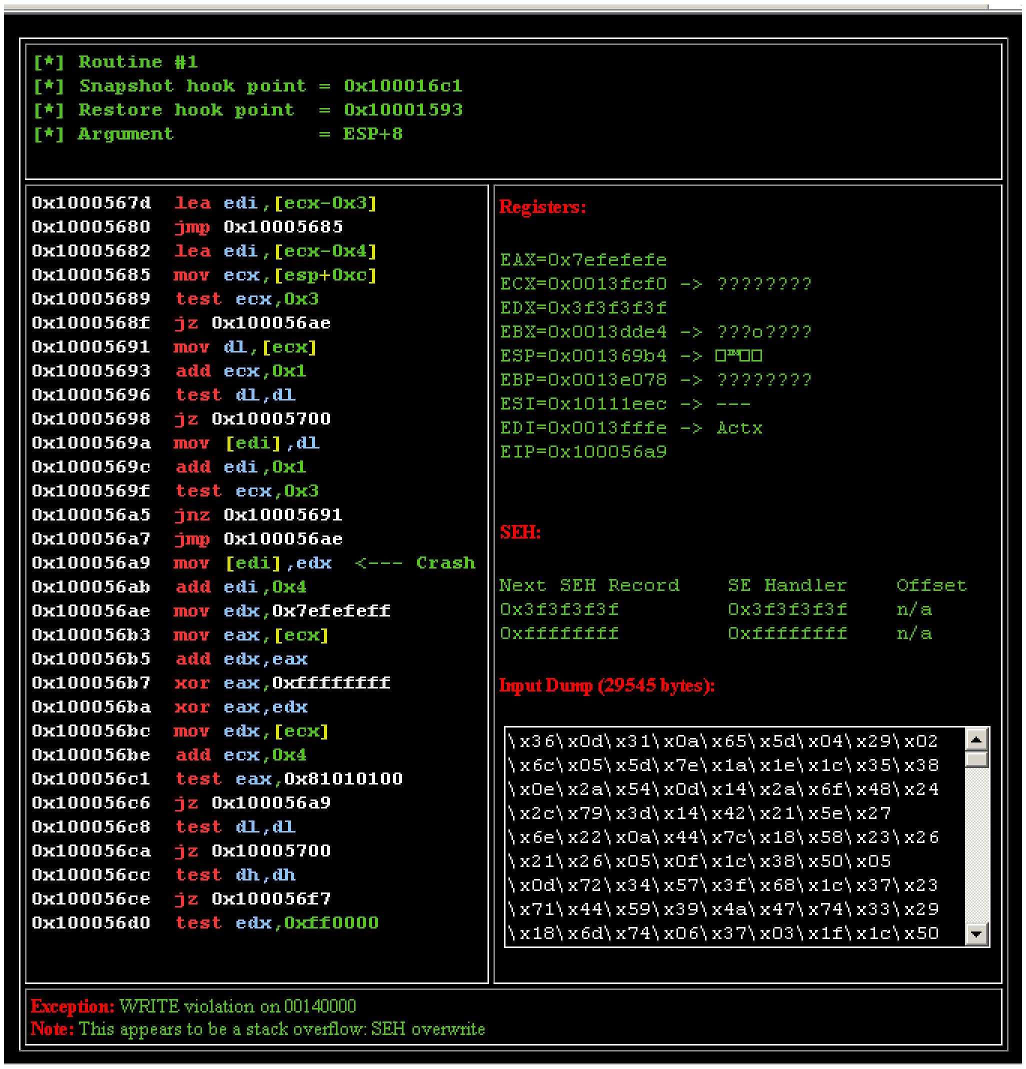

breakpoints.txt - Notepad
File Edit Format View Help
function entries (aka snapshot points)
restore points
arguments

```
0x100016c1 0x10001593 ESP+8
0x100010fa 0x100057c6 ESP+8
```

Once you have breakpoints.txt ready, double click on InMemoryFuzzer.py, you'll be asked which process to attach, trigger the vulnerable routine by feeding some user input again (does not have to be the same pattern as you did for Tracer.py) and then it'll start fuzzing once the execution flow hits our first breakpoint.

Tip:
If your fuzzer hangs during fuzzing, it is probably due to not having a good restore point, or certain input gets out of the code path as you predicted. In that case, pick another restore point, or trigger the fuzzer again.

When the fuzzer is complete, there should be a newly created folder named "**crashbin**" under the same directory as the fuzzer. Crash Bin is a place where InMemoryFuzzer.py stores all the crashes (html files), and the inputs that caused them. Here's an example of a crash dump:



```
[*] Routine #1
[*] Snapshot hook point = 0x100016c1
[*] Restore hook point  = 0x10001593
[*] Argument            = ESP+8
```

```
0x1000567d  lea edi,[ecx-0x3]
0x10005680  jmp 0x10005685
0x10005682  lea edi,[ecx-0x4]
0x10005685  mov ecx,[esp+0xc]
0x10005689  test ecx,0x3
0x1000568f  jz 0x100056ae
0x10005691  mov dl,[ecx]
0x10005693  add ecx,0x1
0x10005696  test dl,dl
0x10005698  jz 0x10005700
0x1000569a  mov [edi],dl
0x1000569c  add edi,0x1
0x1000569f  test ecx,0x3
0x100056a5  jnz 0x10005691
0x100056a7  jmp 0x100056ae
0x100056a9  mov [edi],edx  <--- Crash
0x100056ab  add edi,0x4
0x100056ae  mov edx,0x7efefeff
0x100056b3  mov eax,[ecx]
0x100056b5  add edx,eax
0x100056b7  xor eax,0xffffffff
0x100056ba  xor eax,edx
0x100056bc  mov edx,[ecx]
0x100056be  add ecx,0x4
0x100056c1  test eax,0x81010100
0x100056c6  jz 0x100056a9
0x100056c8  test dl,dl
0x100056ca  jz 0x10005700
0x100056cc  test dh,dh
0x100056ce  jz 0x100056f7
0x100056d0  test edx,0xff0000
```

Registers:

```
EAX=0x7efefefe
ECX=0x0013fcf0 -> ????????
EDX=0x3f3f3f3f
EBX=0x0013dde4 -> ???o????
ESP=0x001369b4 -> □™□□
EBP=0x0013e078 -> ????????
ESI=0x10111eec -> ---
EDI=0x0013fffe -> Actx
EIP=0x100056a9
```

SEH:

```
Next SEH Record     SE Handler     Offset
0x3f3f3f3f          0x3f3f3f3f     n/a
0xffffffff          0xffffffff     n/a
```

Input Dump (29545 bytes):

```
\x36\x0d\x31\x0a\x65\x5d\x04\x29\x02
\x6c\x05\x5d\x7e\x1a\x1e\x1c\x35\x38
\x0e\x2a\x54\x0d\x14\x2a\x6f\x48\x24
\x2c\x79\x3d\x14\x42\x21\x5e\x27
\x6e\x22\x0a\x44\x7c\x18\x58\x23\x26
\x21\x26\x05\x0f\x1c\x38\x50\x05
\x0d\x72\x34\x57\x3f\x68\x1c\x37\x23
\x71\x44\x59\x39\x4a\x47\x74\x33\x29
\x18\x6d\x74\x06\x37\x03\x1f\x1c\x50
```

Exception: WRITE violation on 00140000
Note: This appears to be a stack overflow: SEH overwrite

Each crash dump contains information including:

- Function entry (snapshot point) address
- Argument
- Argument length to crash the application
- Registers (and what data they're pointing to)
- Disassembled instruction
- SEH chains and offsets (if found)
- Input that caused the crash

After an exception is found, the rest leaves for the user to analyze. This is where IDA Pro, or Immunity Debugger becomes handy again.

Who is sinn3r

sinn3r works for Digital Defense Inc. A proud member of Corelan Security and dev of Offensive Security Exploit Database.

Contact: http://twitter.com/_sinn3r