PaiMei - Reverse Engineering Framework

RECON2006

Pedram Amini
pamini@tippingpoint.com

June 16, 2006

Mandatory Narcissistic Slide

- Launched OpenRCE.org one year ago, to date
  - I'm curious, how many of you were here last year?
- Currently employed by TippingPoint
- I manage the Security Research Team (TSRT)
- Small group put together about 6 months ago
  - We are looking to expand
- You will be hearing more from us in the coming months
- Thanks in advance
  - Cody Pierce
  - Cameron Hotchkies
  - Peter Silberman
  - Ero Carrera
  - Beta testers

Talk Outline

- PaiMei overview
    - Motivations behind creation
    - Breakdown of components

Talk Outline

- PaiMei overview
  - Motivations behind creation
  - Breakdown of components
- Command line scripts
  - Intro to and demos of various scripts built on Paimei

Talk Outline

- PaiMei overview
    - Motivations behind creation
    - Breakdown of components
- Command line scripts
    - Intro to and demos of various scripts built on Paimei
- Console (GUI) and tools
    - Intro to and demos of various GUI tools built on PaiMei

Talk Outline

- PaiMei overview
  - Motivations behind creation
  - Breakdown of components
- Command line scripts
  - Intro to and demos of various scripts built on Paimei
- Console (GUI) and tools
  - Intro to and demos of various GUI tools built on PaiMei
- In-house tools, bugs and ideas
  - Overview of some in-house tools not being released
  - Ideas for interested tool developers
  - Needs for future development

The Name

- Named after the Kill Bill 2 character
- Pai Mei actually means white eyebrow
  - But that has nothing to do with the tool

The Sweet Doll

- I haven't decided how to give this out yet
- Or even if I'm willing to part with it for that matter
- Someone in this audience could soon be the proud owner of this bad boy

TISINC99.COM
WE GOT THE VALUE YOU WANT!!!
VOLUME 2
KILL BILL VOL.2
PAI MEI
Features: Cane, Swords, Sword Rack, and Base

Really, What is it?

- It's a win32 reverse engineering framework
- Written entirely in Python
- Think of PaieMei as an RE swiss army knife
- Already proven effective for a number of tasks
    - Fuzzer assistance
    - Code coverage tracking
    - Data flow tracking
    - A beta tester used it to solve the T2'06 RE challenge

My hopes and dreams

That with community support and contributions, PaiMei can do for RE dev what Metasploit does for exploit dev

Motivation: Rapid Development

- Avoid the learning / re-learning curve of various SDKs
- Develop in a higher level language
  - Easy management of arbitrary data structures
  - Less code
  - Less debugging of the actual tool
- Build data representation into the framework, as opposed to an after-thought
  - Of course, coming from me, this translates into graphing

Motivation: Homogenous Environment

- Making tools and languages talk to one another is tedious
  - IDA vs. OllyDbg vs. MySQL
  - C/C++ vs. Python
- Centralized tool creation vs. the old school:
  - Launch debugger
  - Run plug-in
  - Save output to disk
  - Parse output through Perl into IDC
  - Import into IDA

Core Components

PyDbg

A pure Python win32 debugger abstraction class

pGRAPH

An abstraction library for representing graphs as a collection of nodes, edges and clusters

PIDA

A binary abstraction library, built on top of pGRAPH, for representing binaries as a collection of functions, basic blocks and instructions

Extended Components

Utilities

A set of abstraction classes for accomplishing various repetitive tasks

Console

A pluggable WxPython GUI for quickly and efficiently rolling out your own sexy RE tools

Scripts

Individual scripts built on the framework

PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

Example API

```
enumerate_processes()
enumerate_modules()
enumerate_threads()
attach()
load()
suspend_thread()
resume_thread()
```

PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

Example API

```
bp_set_hw()
bp_set()
bp_set_mem()
bp_del_hw()
bp_del()
bp_del_mem()
bp_is_ours_mem()
```

PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

Example API

```
read()
write()
virtual_alloc()
virtual_query()
smart_dereference()
```

PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

Example API

```
process_snapshot()
process_restore()
```

PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

Example API

```
stack_unwind()
seh_unwind()
```

PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

Example API

`set_callback()`

PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

Example API

```
disasm()
disasm_around()
```

PyDbg

Exposes all the expected functionality and then some ...

- Process, module, and thread enumeration
- Hardware, software and memory breakpoints
- Memory read/write/alloc and smart dereferencing
- Memory snapshots and restores
- Stack and SEH unwinding
- Exception and event handling
- Disassembly (libdasm)
- Utility functions

Example API

```
flip_endian()
flip_endian_dword()
func_resolve()
hex_dump()
to_binary()
to_decimal()
```

PyDbg: Example

Abstracted interface allows for painless development

```
from pydbg import *
from pydbg.defines import *

def handler_breakpoint (pydbg):
    # ignore the first windows driven breakpoint.
    if pydbg.first_breakpoint:
        return DBG_CONTINUE

    print "ws2_32.recv() called from thread %d @%08x" % \
        pydbg.dbg.dwThreadId,
        pydbg.exception_address)

    return DBG_CONTINUE

dbg = pydbg()

# register a breakpoint handler function.
dbg.set_callback(EXCEPTION_BREAKPOINT, handler_breakpoint)
dbg.attach(XXXXX)

recv = dbg.func_resolve("ws2_32", "recv")
dbg.bp_set(recv)

pydbg.run()
```

PyDbg: Random Idea Implementation

The problem

I want to solve the F-Secure T2'06 challenge ... but I'm lazy.

1. Open the binary in IDA
2. Locate password read and process exit
3. Set breakpoints on both
4. The first time a password is read, snapshot
5. When the exit is reached, restore
6. Read the buffer address off the stack
7. Change the password
8. Continue

pGRAPH

Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

pGRAPH

Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

Example API

```
add_node()
add_edge()
del_node()
del_edge()
```

pGRAPH

Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

Example API

```
find_node()
find_edge()
edges_from()
edges_to()
```

pGRAPH

Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

Example API

```
graph_cat()
graph_sub()
graph_up()
graph_down()
graph_intersect()
graph_proximity()
```

pGRAPH

Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

Example API

```
render_graph_graphviz()
render_graph_gml()
render_graph_udraw()
```

pGRAPH

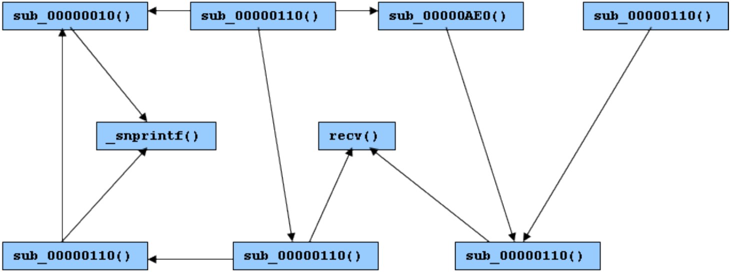
Exposes much of the expected functionality:

- Node and edge management
- Node and edge searching
- Graph manipulation
- Graph rendering

Why do we need this library?

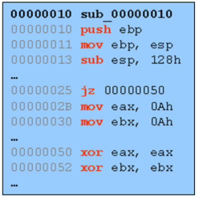
Graph Representation: Module

- Disassembled binaries can be represented as graphs
  - Functions represented as nodes
  - Intra-modular calls represented as edges
- AKA call graph



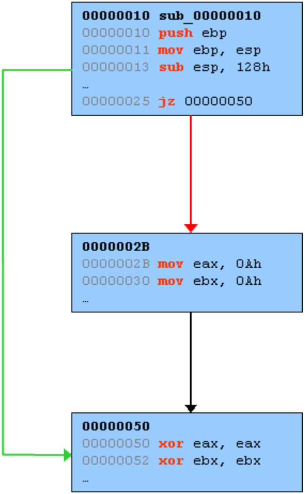
Graph Representation: Function

- Functions can also be represented as graphs
    - Basic blocks represented as nodes
    - Branches represented as edges



```
00000010 sub_00000010
00000010 push ebp
00000011 mov ebp, esp
00000013 sub esp, 128h
…
00000025 jz 00000050
0000002B mov eax, 0Ah
00000030 mov ebx, 0Ah
…
00000050 xor eax, eax
00000052 xor ebx, ebx
…
```

- AKA control flow graph or CFG



```
00000010 sub_00000010
00000010 push ebp
00000011 mov ebp, esp
00000013 sub esp, 128h
…
00000025 jz 00000050
```

```
0000002B
0000002B mov eax, 0Ah
00000030 mov ebx, 0Ah
…
```

```
00000050
00000050 xor eax, eax
00000052 xor ebx, ebx
…
```

PIDA

- Extends from pGRAPH to represent binaries as a graph of graphs
- PIDA files are propogated by an IDA Python script pida_dump.py
  - This is important, I will show it to you in a second
- The database is serialized to a zlib compressed .pida file
- PIDA enumerates basic blocks and discovers RPC routines
- The .pida file can later be loaded independent of IDA
- All the aforementioned graph functionality is available for (ab)use
- Quick demo

PIDA: Contrived Example

Again, abstracted interface allows for painless development

```
import pida import *

module = pida.load("some_file.pida")

# render a function graph in uDraw format for the entire module.
fh = open("graphs/functions.udg", "w+")
fh.write(module.render_graph_udraw())
fh.close()

# step through each function in the module:
for function in module.functions.values():
    # if we found the function we are interested in:
    if function.name == "some_function":
        # step through each basic block in the function.
        for bb in function.basic_blocks.values():
            print "\t%08x - %08x" % (bb.ea_start, bb.ea_end)
            # print each instruction in each basic block.
            for ins in bb.instructions.values():
                print "\t\t%s" % ins.disasm

        # render a GML graph of this function.
        fh = open("graphs/function.gml", "w+")
        fh.write(function.render_graph_gml())
        fh.close()
```

PIDA: Contrived Example

...Continued

```
# if we found the second function we are interested in.
if function.ea_start == 0xdeadbeef:

    # render a uDraw format proximity graph.
    fh = open("graphs/proximity.udg", "w+")

    # look 3 levels up and 2 levels down.
    prox_graph = module.graph_proximity(function.id, 3, 2)
    fh.write(prox_graph.render_graph_udraw())
    fh.close()
```

Together, PIDA and PyDbg offer a powerful combination for building a variety of tools. Consider for example the ease of re-creating Process Stalker on top of this platform.

PIDA: Real World Example

Locate all functions within a binary that open a file and display the execution path from the entry point to the call of interest...

```
# for each function in the module
for function in module.functions.values():
    # create a downgraph from the current routine and locate the calls to [Open|Create]File[A|W]
    downgraph = module.graph_down(function.ea_start, -1)
    matches   = [node for node in downgraph.nodes.values() if re.match(".*(create|open)file.*", \
                node.name, re.I)]
    upgraph   = pgraph.graph()

    # for each matching node create a temporary upgraph and add it to the parent upgraph.
    for node in matches:
        tmp_graph = module.graph_up(node.ea_start, -1)
        upgraph.graph_cat(tmp_graph)

    # write the intersection of the down graph from the current function and the upgraph from
    # the discovered interested nodes to disk in gml format.
    downgraph.graph_intersect(upgraph)

    if len(downgraph.nodes):
        fh = open("%s.gml" % function.name, "w+")
        fh.write(downgraph.render_graph_gml())
        fh.close()
```

Utilities

- Classes for further abstracting frequently repeated functionality:
    - Code Coverage
    - Crash Binning
    - Process Stalker
    - uDraw Connector

Utility: Code Coverage

- Simple container for storing code coverage data
- Supports persistant storage to MySQL or serialized file
- You can use this class to keep track of where you have been
- Examples:
    - Process Stalker
    - Individual fuzzer test case tracking

Utility: Crash Binning

- Simple container for categorizing and storing "crashes"
- Stored crashes are organized in bins by exception address
- The in-house version of this class goes one step further by categorizing on path as well (stack unwind)
- The `crash_synopsis()` routine generates detailed crash reports:
    - Exception address, type and violation address
    - Offending thread ID and context
    - Disassembly around the exception address
    - Stack and SEH unwind information
- This class is extremely useful for fuzzer monitoring
    - *ex:* 250 crashes vs. 248 crashes at x and 2 crashes at y
- *Note to Pedram*: Mention the Excel file format exploit "fuzzer"

Utility: Process Stalker

- Abstracted interface to Process Stalking style code coverage
- Currently only being used by the pstalker GUI module
- A command line interface can be easily built
- The class handles all the basics:
  - Re-basing and setting breakpoints in the main module
  - Re-basing and setting breakpoints in loaded libraries
  - Recording, with or without context data, hit breakpoints
  - Monitoring for access violations
  - Exporting (through the code coverage class) to MySQL

Utility: uDraw(Graph) Connector

Python interface to the uDraw(Graph) API. Much of the uDraw API currently remains unwrapped. *Note to Pedram*: Mention how badass uDraw is.

- Draw graphs
- Move the graph
- Modify the graph
- Register callbacks

Utility: uDraw(Graph) Connector

Python interface to the uDraw(Graph) API. Much of the uDraw API currently remains unwrapped. *Note to Pedram*: Mention how badass uDraw is.

- Draw graphs
- Move the graph
- Modify the graph
- Register callbacks

Example API

```
graph_new()
graph_update()
```

Utility: uDraw(Graph) Connector

Python interface to the uDraw(Graph) API. Much of the uDraw API currently remains unwrapped. *Note to Pedram*: Mention how badass uDraw is.

- Draw graphs
- Move the graph
- Modify the graph
- Register callbacks

Example API

```
focus_node()
layout_improve_all()
scale()
open_survey_view()
```

Utility: uDraw(Graph) Connector

Python interface to the uDraw(Graph) API. Much of the uDraw API currently remains unwrapped. *Note to Pedram*: Mention how badass uDraw is.

- Draw graphs
- Move the graph
- Modify the graph
- Register callbacks

Example API

```
change_element_color()
window_background()
window_status()
window_title()
```

Utility: uDraw(Graph) Connector

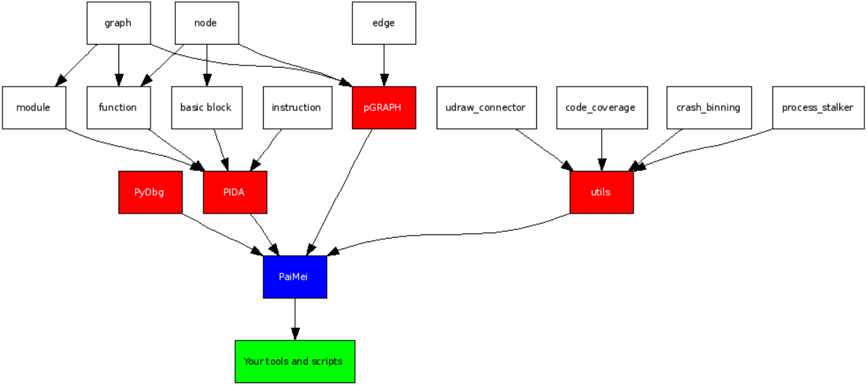
Python interface to the uDraw(Graph) API. Much of the uDraw API currently remains unwrapped. *Note to Pedram*: Mention how badass uDraw is.

- Draw graphs
- Move the graph
- Modify the graph
- Register callbacks

Example API

```
set_command_handler()
```

How it All Ties Together



DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

```
procedure("pedram", 25)
```

DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions

```
procedure("pedram", 25)
```

DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions
2. Reverse the argument list

```
procedure("pedram", 25)
```

DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions
2. Reverse the argument list
3. PUSH numeric arguments directly

```
procedure("pedram", 25)
```

```
PUSH 20
```

DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions
2. Reverse the argument list
3. `PUSH` numeric arguments directly
4. Allocate space for string arguments and `PUSH` address

```
procedure("pedram", 25)
```

```
PUSH 20
PUSH 0x12345678
```

```
0x12345678: "pedram"
```

DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions
2. Reverse the argument list
3. PUSH numeric arguments directly
4. Allocate space for string arguments and PUSH address
5. Write the CALL instruction

```
procedure("pedram", 25)
```

```
PUSH 20
PUSH 0x12345678
CALL procedure
```

0x12345678: *"pedram"*

DPC: Debuggee Procedure Call

Allows you to call arbitrary functions in your target. Implemented using a simple process:

1. Allocate space for new instructions
2. Reverse the argument list
3. `PUSH` numeric arguments directly
4. Allocate space for string arguments and `PUSH` address
5. Write the `CALL` instruction
6. Write an `INT 3` to regain control

```
procedure("pedram", 25)
```

```
PUSH 20
PUSH 0x12345678
CALL procedure
INT 3
```

`0x12345678:` *"pedram"*

DPC: Usage

- Once attached you are given a command prompt
- Any Python statement is valid
- `dbg` references current PyDbg instance
- Convenience wrappers exist for memory manipulaton
    - `alloc()`, `free()`, `free_all()`, `show_all()`
- Assigned variables are not persistant!
    - Use `glob` for that
    - `print glob` to display what you have assigned
- `dpc(procedure, *args, **kwargs)`
    - kwargs are for fast call support
- Took me less than 30 minutes to write the 1st version of this tool

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ
25-29	29	6

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ
25-29	29	6
30-31	31	2

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ
25-29	29	6
30-31	31	2
32-37	37	6

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ
25-29	29	6
30-31	31	2
32-37	37	6
38-41	41	4

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ
25-29	29	6
30-31	31	2
32-37	37	6
38-41	41	4
42-43	43	2

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ
25-29	29	6
30-31	31	2
32-37	37	6
38-41	41	4
42-43	43	2
44-47	47	4

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ
25-29	29	6
30-31	31	2
32-37	37	6
38-41	41	4
42-43	43	2
44-47	47	4
48-53	53	6

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ
25-29	29	6
30-31	31	2
32-37	37	6
38-41	41	4
42-43	43	2
44-47	47	4
48-53	53	6
54-59	59	6

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ
25-29	29	6
30-31	31	2
32-37	37	6
38-41	41	4
42-43	43	2
44-47	47	4
48-53	53	6
54-59	59	6
60-61	61	2

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ
25-29	29	6
30-31	31	2
32-37	37	6
38-41	41	4
42-43	43	2
44-47	47	4
48-53	53	6
54-59	59	6
60-61	61	2
62-67	67	6

DPC: Example One

Taking shortcuts

- The following routine would have taken a good effort to reverse
- Using DPC however the functionality is quickly evident
- Call out the answer if you know it

Input Range	Return	Δ
25-29	29	6
30-31	31	2
32-37	37	6
38-41	41	4
42-43	43	2
44-47	47	4
48-53	53	6
54-59	59	6
60-61	61	2
62-67	67	6
68-71	71	4

DPC: Example Two

Here's another one...

Arg 1	Arg 2	Arg 3	Return

DPC: Example Two

Here's another one...

Arg 1	Arg 2	Arg 3	Return
paimei	eyebrow	25	0x00000001

DPC: Example Two

Here's another one...

Arg 1	Arg 2	Arg 3	Return
paimei	eyebrow	25	0x00000001
paimei	apple	50	0x00000001

DPC: Example Two

Here's another one...

Arg 1	Arg 2	Arg 3	Return
paimei	eyebrow	25	0x00000001
paimei	apple	50	0x00000001
paimei	pear	69	0xFFFFFFFF

DPC: Example Two

Here's another one...

Arg 1	Arg 2	Arg 3	Return
paimei	eyebrow	25	0x00000001
paimei	apple	50	0x00000001
paimei	pear	69	0xFFFFFFFF
pai	paimei	666	0xFFFFFFFF

DPC: Example Two

Here's another one...

Arg 1	Arg 2	Arg 3	Return
paimei	eyebrow	25	0x00000001
paimei	apple	50	0x00000001
paimei	pear	69	0xFFFFFFFF
pai	paimei	666	0xFFFFFFFF
paimei	paimei	31337	0x00000000

DPC: Example Two

Here's another one...

Arg 1	Arg 2	Arg 3	Return
paimei	eyebrow	25	0x00000001
paimei	apple	50	0x00000001
paimei	pear	69	0xFFFFFFFF
pai	paimei	666	0xFFFFFFFF
paimei	paimei	31337	0x00000000
pai	paimei	3	0x00000000

DPC: (Quick) Live Demo

OllyDbg Connector

- PyDbg is designed for mostly non-interactive functionality
- This two-part tool adds live graphing functionality to OllyDbg
- Part 1: Receiver
  - Socket server for OllyDbg
  - Receives module name, base address and offset from plug-in
  - Socket client to uDraw(Graph)
  - Loads specified PIDA file and generates graph
- Part 2: Connector
  - Registers hotkeys for transmitting location to receiver
  - , Step into and xmit current location
  - . Step over and xmit current location
  - / Xmit current location

OllyDbg Connector: Live Demo

Proc Peek

- This two-part tool was designed for discovering *low hanging fruit* vulnerabilities
  - Which, believe it or not, is quite effective
- The first half of the tool is a static reconnaissance phase
  - *proc_peek_recon.py*
- The second half of the tool is a run-time analysis phase
  - *proc_peek.py*

General philosophy

With minimal setup, generate a list of locations that can be easily monitored and *checked off*. This approach is great for 1st phase auditing and can be handed off to an intern.

Proc Peek: proc_peek_recon.py

- IDA Python script
- Looks for *interesting* locations, or peek points
    - Inline `memcpy()` and `strcpy() routines`
    - Calls to API that accept format string tokens
        - Ignoring ones that do not contain `%s`
    - Calls to potentially *dangerous* API such as `strcat()`, `strcpy()`, etc...
- Discovered peek points are written to a file
- I'll show you this now

Proc Peek: proc_peek.py

- PyDbg based script (a bit dated)
- Attach to the target process
- Set breakpoints on each peek point
- When a breakpoint is hit:
    - Present the user with relevant information
    - Prompt for action: *ignore*, *continue*, *make notes*
- Supports automated keyword searching (Hoglund: *Boron tagging*)
- Also features Winsock recv() tracking (more on this later)
- I don't have a good demo for this, so we'll move on

Overview

- Some complex tools are not suitable for the command line
- The PaiMei console provides a base for new GUI modules
- Development for the framework is well documented (I think)
- Allows you to focus your effors on the tool

General layout

- Modules are independent of one another
  - Though you can push / pull data between them
- Each module represented by a notebook icon
- Entire right pane is controlled by the module
- Left status bar displays console wide messages
- Right status bar is owned by the current module
- *Connections* menu establishes connectivity to MySQL and uDraw
- *Advanced* menu exposes log window clearing and CLI
- The CLI (Command Line Interface) is a full Python interpreter and allows you to interact with any portion of the console.
  - Explicitly documented module member variables are listed on the right-hand side of the CLI

PAIMEIdocs

- HTML documentation browser
- Use the control bar at the top to load general or developer specific documentation
- Not all that exciting

PAIMEIexplore

- The *hello world* of the console
- The in-house version has a bit more functionality
- To use:
  - Load a PIDA file
  - Double click the PIDA file
  - Browse through the explorer tree
  - Select a function to display disassembly
  - Connect to uDraw
  - Graph a function through the right-click context menu

Overview

- File fuzzing and exception monitoring tool built on PaiMei
- Developed by Cody Pierce
- Loads a target file
- Generates mutations based at specified offset / range, variable length and byte values
    - More advanced features include, additive mutations
- Supports mid-session pause and resume
- Features predictable completion time and run-time statistics
- In-house experimental features:
    - Auto file discovery
    - Auto handler discovery
    - Auto fuzz
    - ie: Give it a laptop and go

Live Demo

Overview

- A binary diffing tool built on PaiMei
- Being developed by Peter Silberman
- Still an early beta and not currently distributed
- Heuristic based diffing engine (like Sabre BinDiff)
- The goal of the module is to allow the user to deeply control the diffing algorithm
- Customized algorithms can be saved for later use
- This will likely lead to job specific sets:
  - Malware analysis
  - Generic patch diffing
  - Microsoft patch diffing
  - Etc...

Supported Heuristics

Some of these were gleaned from the Sabre Security white papers:

- API calls
- Argument and variable sizes
- Constants
- Control flow
- CRC
- Name
- NECI (graph heuristics)
- Recursive calls
- Size
- Small Prime Product (SPP)
- "Smart" MD5
- Stack frame
- String references

Live Demo

Overview

- Code coverage recording tool
- This is the "next generation" of Process Stalker
- All metadata is stored to MySQL
- Three step approach:
    - Setup data sources
    - Capture code coverage data
    - Explore captured data
- Filtering support allows you to pinpoint interesting code locations

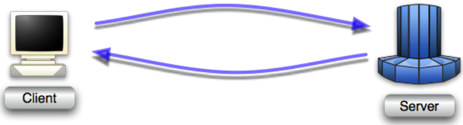
Live Demo

PAIMEIproxyfuzz

- Developed by Cody Pierce
- Currently in an experimental phase
- Simple concept for inline client/server fuzzing

PAIMEIproxyfuzz

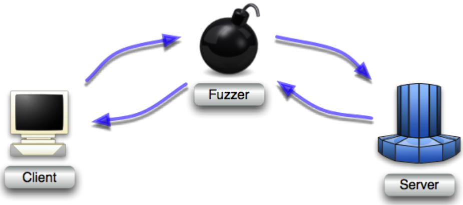
- Developed by Cody Pierce
- Currently in an experimental phase
- Simple concept for inline client/server fuzzing



- Typical client / server communication
- Blue edge represents legit data

PAIMEIproxyfuzz

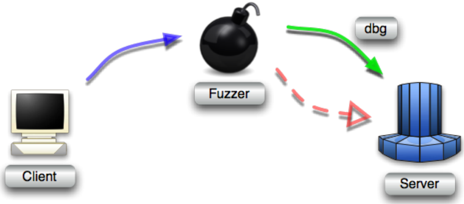
- Developed by Cody Pierce
- Currently in an experimental phase
- Simple concept for inline client/server fuzzing



- Proxy becomes server to client and client to server
- Purely pass thru at this point

PAIMEIproxyfuzz

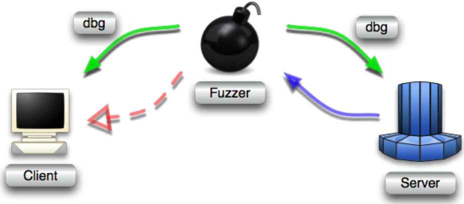
- Developed by Cody Pierce
- Currently in an experimental phase
- Simple concept for inline client/server fuzzing



- Potentially mutate client request prior to pass thru
- Attach PyDbg to receiving process (exception monitoring)

PAIMEIproxyfuzz

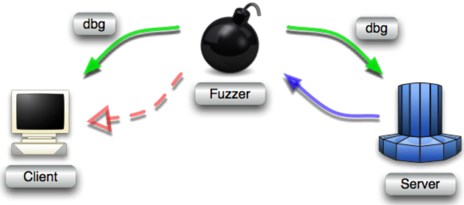
- Developed by Cody Pierce
- Currently in an experimental phase
- Simple concept for inline client/server fuzzing



- Potentially mutate server response prior to pass thru
- Attach PyDbg to receiving process (exception monitoring)

PAIMEIproxyfuzz

- Developed by Cody Pierce
- Currently in an experimental phase
- Simple concept for inline client/server fuzzing



- Adn yes, this has found bugs already
- In enterprise backup software you probably use today

PAIMEIsocketstalker

- Use breakpoints to "hook" `recv()` and recv_from()
  - `recv(SOCKET s, char *buf, int len, int flags);`
- Grab the buffer address and receive length arguments
  - `address = dbg.get_arg(2)`
  - `length = dbg.get_arg(3)`
- If and only if the buffer is not on the stack (more on this later)
- Set a memory breakpoint on the buffer range
  - `if not dbg.is_address_on_stack(address):`
  - ` dbg.bp_set_mem(buffer_address, length)`
- The memory breakpoint handler takes care of the rest

Memory Breakpoint Handling

- `memory_breakpoint_hit` boolean flag indicates direct hits
- Offending instruction address, target address and violation type
  - `dbg.exception_address`
  - `dbg.write_violation`
  - `dbg.violation_address`
- End result: Know which instructions touched what bytes of data
  - ie: Ghetto, yet functional data flow tracking

Memory Breakpoint Handling

- `memory_breakpoint_hit` boolean flag indicates direct hits
- Offending instruction address, target address and violation type
  - `dbg.exception_address`
  - `dbg.write_violation`
  - `dbg.violation_address`
- End result: Know which instructions touched what bytes of data
  - ie: Ghetto, yet functional data flow tracking

Limitations

Smallest granularity for memory breakpoints is page size (4k). This is fine for the heap, but miserable for the stack.

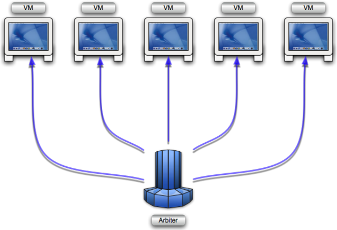
PAIMEIfilestalker

- Similar concept to socket stalker
- More API hooks are necessary:
  - `CreateFileA/W()`: Regex on file name argument
  - MapViewOfFile/Ex(): Regex on `GetMappedFileNameA()`
  - ReadFile/Ex(): Track read buffers
- The rest of the logic is same as before
- With file tracking, we have a solution for tracking stack buffers...

Parallel and Serial HW Breakpoint Abuse

Stack Buffer			

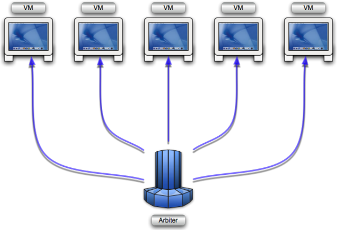
Parallel and Serial HW Breakpoint Abuse



Stack Buffer			
vm-1	vm-2	vm-3	vm-4
vm-5			

- Using an arbitration script
- Divide the target buffer among the available systems

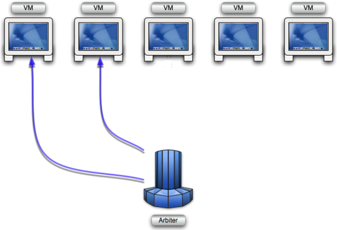
Parallel and Serial HW Breakpoint Abuse



Stack Buffer			
	vm-1	vm-2	vm-3
vm-4	vm-5		

- As the entire buffer range was not exhausted
- Restart the process with the same target file
- This is possible because the re-processing of a file is deterministic
- Alternatively: memory snapshot / restore and VMWare revert

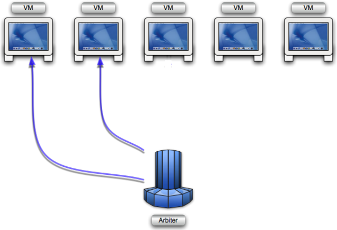
Parallel and Serial HW Breakpoint Abuse



Stack Buffer			
		vm-1	vm-2

- Repeat as necessary

Parallel and Serial HW Breakpoint Abuse



Stack Buffer			

- *Note to Pedram*: Show example output

My Attempt to Inspire

- I hope this framework picks up some traction
- To aid that along I am going to share some random ideas for development

Malware Profiler

- I will never get around to this, so someone else do it
- Post unpacking / PIDA conversion, static analysis tool
- Step through the call chains within the binary
  - Mark common sequences with a high level label
  - Automatically extract information such as mutex name, startup keys, etc..
- Can help narrow analysis areas, ie:
  - Glean what you can through live analysis
  - Automatically tag and command statically recognized code sequences
  - What you are left with will be the more interesting sections
- The tool should be driven by XML configuration files (next slide)

Malware Profiler: Continued

Theorized example XML

```
<classification name="SMTP Engine">
    <API name="htons">
        <argument index=1>25</argument>
    </API>
</classification>
<classification name="Address Harvesting">
    <API name="FindFirstFile()"></API>
    <API name="FindNextFile()"></API>
    <API name="MapViewOfFile()"></API>
    <string match="regex">
        [^@]+@[^\.]+\.com
    </string>
</classification>
<classification name="Startup Entry">
    <API name="RegCreateKeyEx">
        <argument index=1>
            HKEY_LOCAL_MACHINE
        </argument
        <argument index=2>
            <string match="regex">\run|\runonce</string>
        </argument>
    </API>
</classification>
```

PyDbg Symbol Support

- Add the necessary Windows API to parse symbols
- Automatically provide symbolic names throughout the output when available

More Advanced Explorer Interface

- The addition of some basic navigation features could be useful
- Some features similar to IDA, such as:
    - Comment support
    - Cross reference jumping
    - Searching
    - Etc...

Memory Snapshot Management Class

- A generic class for managing memory snapshots from PyDbg would be nice
- Similar to crash binning or code coverage
- Desired features include:
    - Persistant storage
    - Enumeration
    - Search
    - Diff support
- The diff feature could come into play for example in DPC
    - List all changes made by the last procedure I called

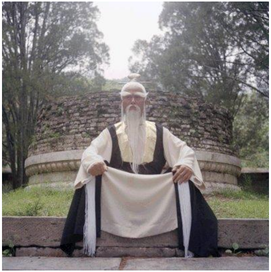
A Real Installer

- This will likely be a key factor in spreading adoption of PaiMei
- The full installation of PaiMei has number of prerequisites
- My *__install_requirements.py* script is ok, but far from optimal
- It would be nice if someone with better installer skills created one
    - Nullsoft NSIS perhaps?

Bugs and enhancements

- While it is stable, the framework is constantly maturing
- One major current design issue:
  - PIDA files can consume a lot of memory
- The solution I have for this in my head:
  - Do not load the entire contents of the file
  - Instead, poll the file on demand
- Another major issue is IDAs misrepresentation
  - ie: Alex's talk, but where we have no symbols
  - Ero Carrera of Sabre is doing some work in this arena

Questions?



Total Slide Count

62