Process Stalking

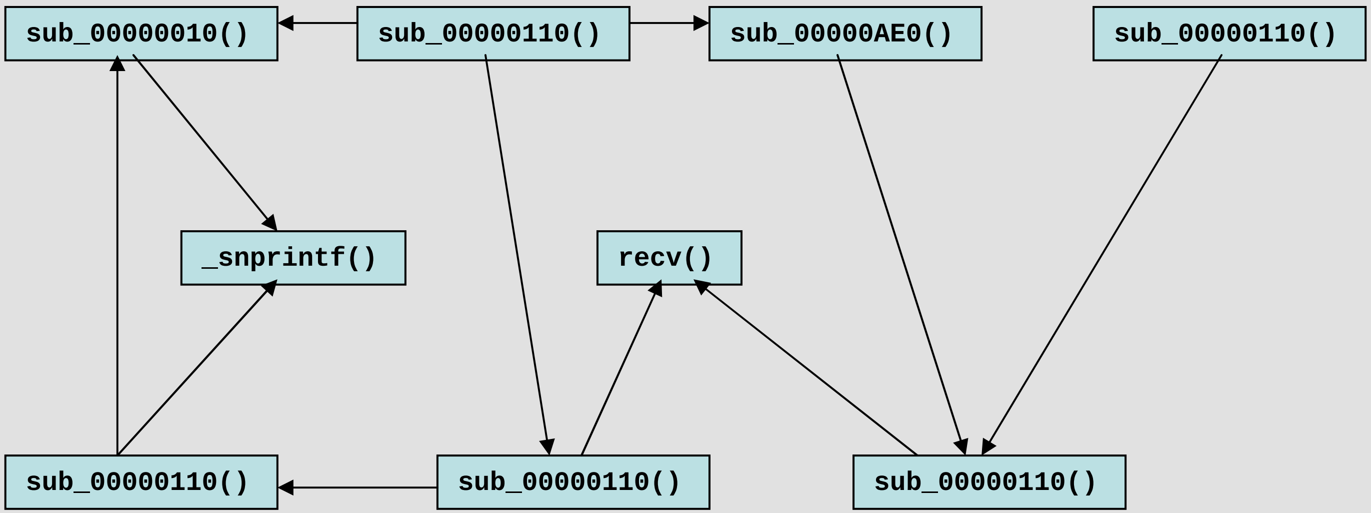
Run-Time Visual Reverse Engineering

Pedram Amini – pamini@tippingpoint.com

```
0100: mov ax, 13h      0108: mov ds, bx     010E: inc ax        0113: inc si             011A: mov si, cx             01010100010010010101000001010001010
0103: int 10h          010A: xor cx, cx     010F: add al, ah    0114: cmp si, 0FA00h     011C: jmp short loc_10C      00101001011010101101000100111010100
0105: mov bx, 0A000h   010C: mov al, [si]   0111: mov [si], al  0118: jnz short loc_10C  01010110100010101010111001010101101010101010100101110101010
0111110001011010101110010100101111001010100001000000000101001010101111001100001100101010100101110011111110000110101010000000111110100101111001010100001
0100110001010000101111000101010010111100101010001100101010001000111011110010100101111001010100010010100101010101010000000101010100000111110101010101000
```

Call Graphs

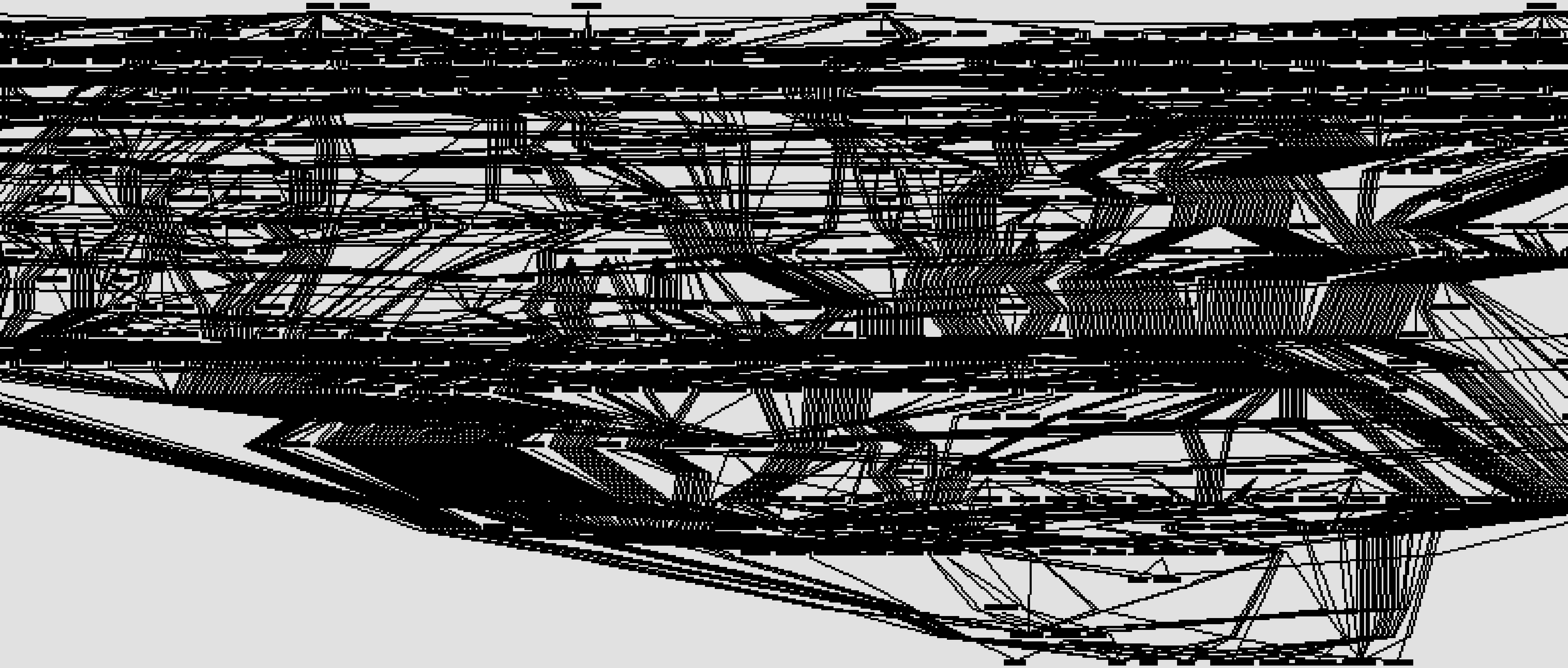
- Disassembled binaries can be visualized as graphs
  - Functions = nodes
  - Calls = edges
- IDA supports this type of visualization
- Useful for viewing the relationships between functions



- However…

Call Graphs

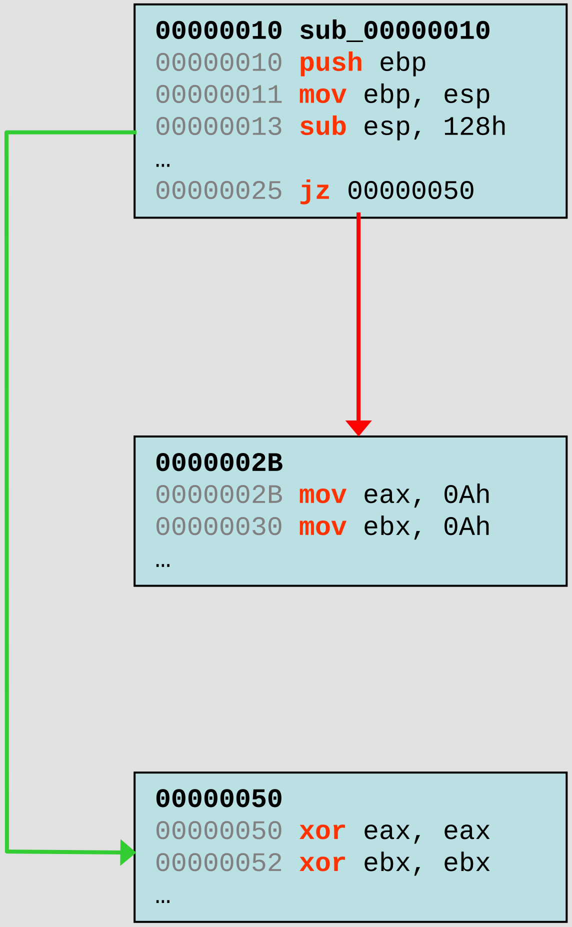
- In most real-world scenarios, function call graphs can be unmanageable and down right frightening:



Control Flow Graphs (CFGs)

- Functions can also be visualized as graphs
  - Basic blocks = nodes
  - Branches = 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
…
```



```
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
…
```

- IDA also supports this type of visualization
- Useful for easy viewing of execution paths
- pGRAPH

RE Analysis Challenges

- Input tracing
  - What code handles our inputs?
- Code coverage
  - How we can we determine where our fuzzer has gone?
  - How can we get our fuzzer deeper into the process?
- Complexity
  - How can we digest/understand mass volumes of machine code?
- Filtering
  - How can we filter uninteresting trace data? (Example: GUI handling code)
- Trace speed
  - How can we increase the speed of our tracing?

Process Stalker Overview

- Requirements
  - IDA Pro (commercial)
  - Python (free)
  - Oreas GDE Community Edition (free)
    - Quick demo in a second

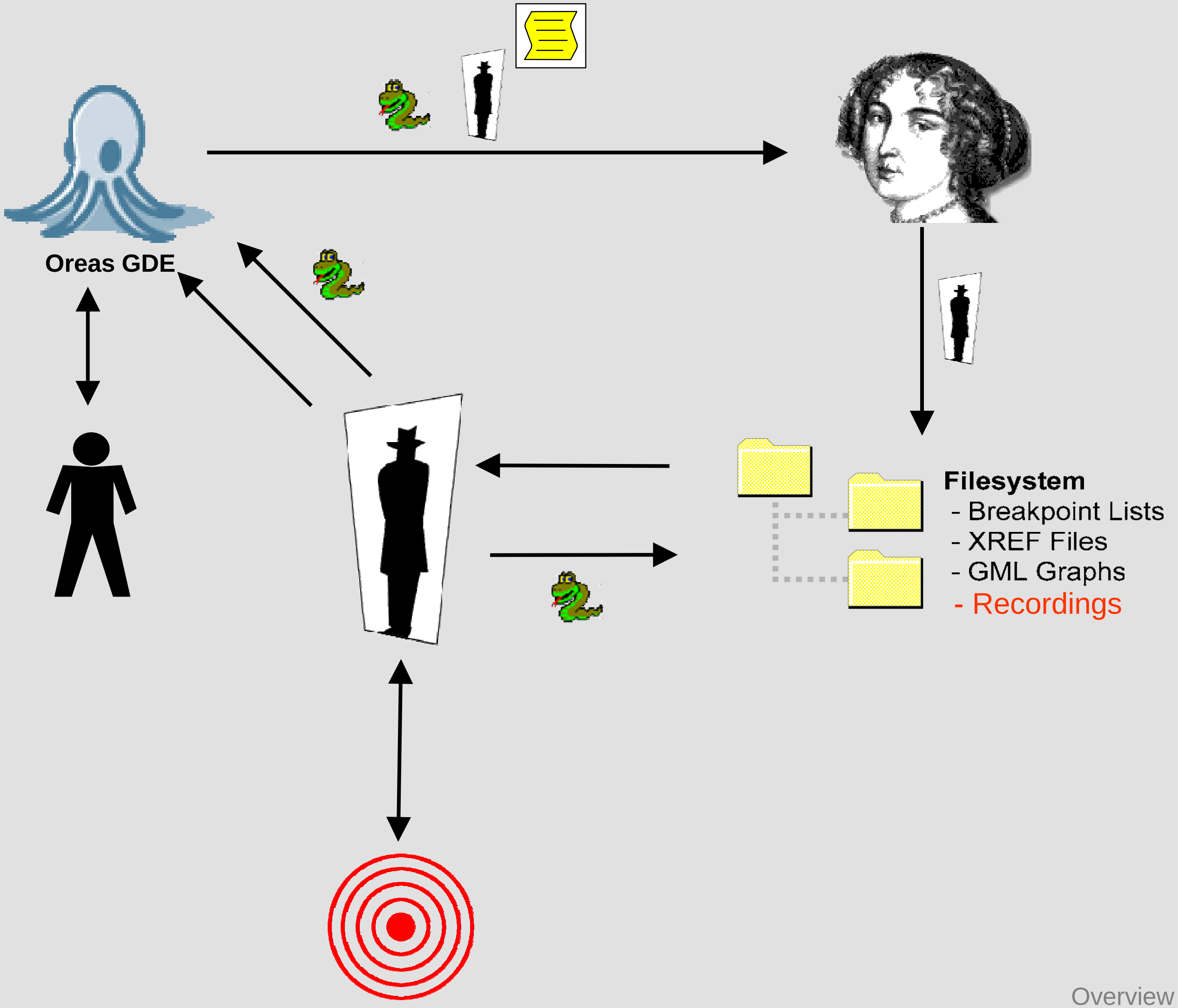
- Components
  - IDA plug-in
  - Standalone tracer
  - Python scripts

- Development
  - C/C++
  - Python + custom API
  - Function Analyzer / Dumbug

- Related work
  - Sabre Security, BinNavi
  - HBGary, Inspector
  - SISecure? (Rootkit.com screenshot)

Data Flow Diagram

- Load binary in IDA
- Export to FS
- Stalk process
- Record
- Process results
- View in GDE
- Instrument graphs
- View in GDE again
- Make edits
- Mark locations
- Export back to IDA



Oreas GDE

Filesystem
- Breakpoint Lists
- XREF Files
- GML Graphs
- Recordings

Process Stalker IDA Plug-in Internals

- Built on top of Function Analyzer

- Analysis routine is applied to each identified function

- Breakpoint entries are generated for every node:
  - ndmpsrvr.dll:0002b1b0:0002b29c
  - Module, function offset, node offset

- Cross reference entries are generated for every call:
  - 0002cbd0:0002cc34:0002bb20
  - Function offset, node offset, called function offset

- Customized .GML graph’s are generated for each function:
  - ndmpsrvr.dll-010a1af0.gml
  - ndmpsrvr.dll-010a1b20.gml

Process Stalker Tracer Internals

- Built on top of Dumbug

- Attach to or load a target process

- On DLL load events
  - Determine module base address
  - Add loaded module to linked list
  - Automatically import available breakpoints

- On breakpoint events
  - If recording, write entry to file:
    - 0008c29d:000005cc:IMComms.dll:10001000:0000d25d
    - GetTickCount(), thread ID, module, module base, breakpoint offset
  - Optionally raise breakpoint restore flag and SINGLE_STEP
  - Optionally apply and record register inspection (next slide)

Process Stalker Tracer Register Inspection

- Newer, unique and very useful feature
- Adds “smart” data about register contents to hit nodes
- Decreases performance of course
  - But not enough to outweigh the benefits
- Overview
  - On breakpoint event
  - For each register
    - Dereference as memory address
    - Ignore non-writeable addresses
    - If address is within stack range, mark stack, otherwise mark heap
    - Check for Unicode string
    - Check for ANSI string
    - Grab hex bytes (32)
    - Record

Process Stalker Script Internals

- Written in Python

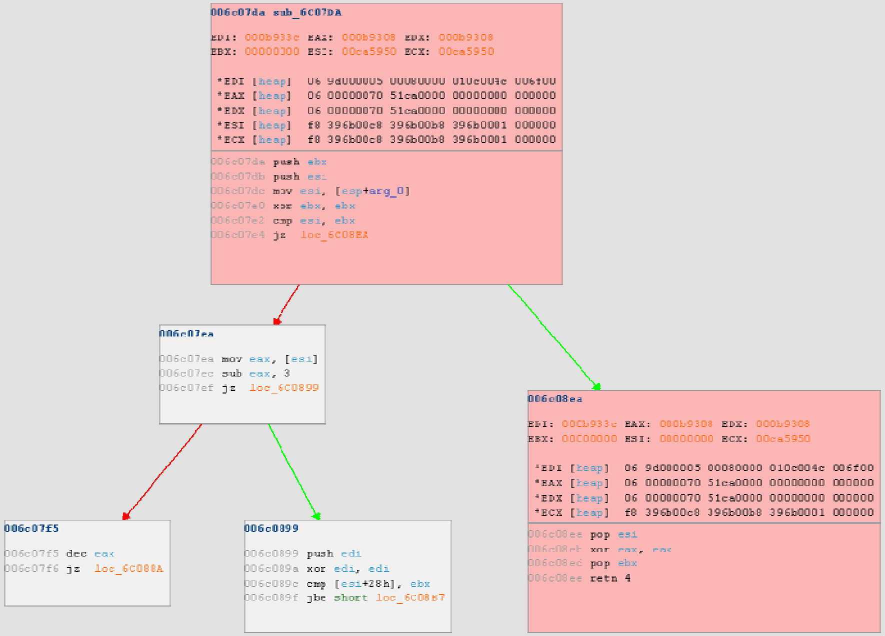
- Process Stalker API: gml, ps_parsers
  - GML: Can parse and manipulate generated .GML files
  - PS_PARSERS: Can parse and manipulate breakpoint lists, recordings, cross-reference lists and register metadata files
  - Fully documented

- Various functionality already implemented:
  - Recording -> list -> breakpoint filter
  - Graph concatenation with optional cross referencing
  - Recursive graph visualization
  - Run trace “folding” for loop visualization (more on this later)
  - And more…

Now the pretty slides…

Visual Run-Time Tracing

- Immediately see which nodes handle your input
- View graphs with different layout algorithms
- View relevant register data



```
006c07da sub_6C07DA

EDI: 000b933c EAX: 000b9308 EDX: 000b9308
EBX: 00000000 ESI: 00ca5950 ECX: 00ca5950

 *EDI [heap]  06 9d000005 00080000 010c004c 006f00
 *EAX [heap]  06 00000070 51ca0000 00000000 000000
 *EDX [heap]  06 00000070 51ca0000 00000000 000000
 *ESI [heap]  f8 396b00c8 396b00b8 396b0001 000000
 *ECX [heap]  f8 396b00c8 396b00b8 396b0001 000000
006c07da push ebx
006c07db push esi
006c07dc mov esi, [esp+arg_0]
006c07e0 xor ebx, ebx
006c07e2 cmp esi, ebx
006c07e4 jz  loc_6C08EA
```

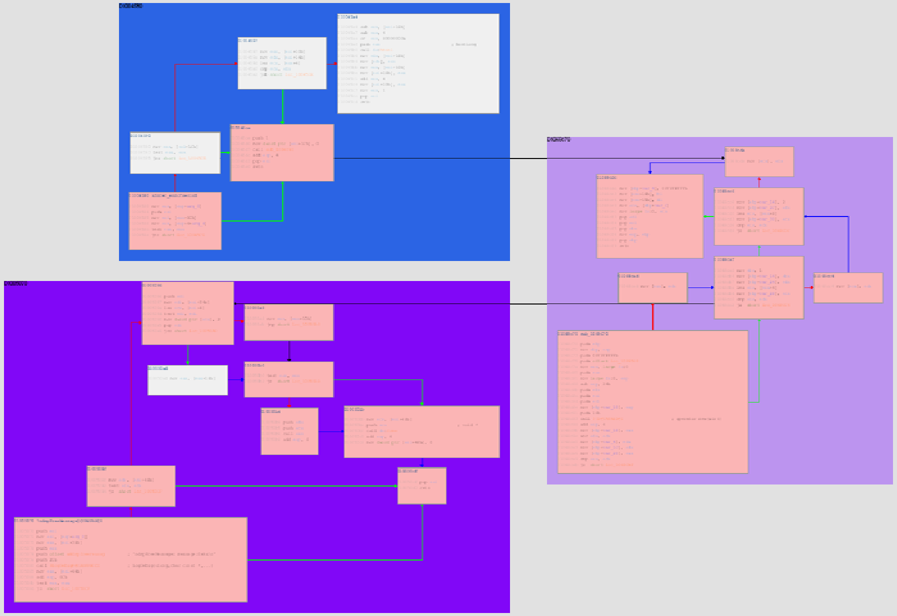
```
006c07ea

006c07ea mov eax, [esi]
006c07ec sub eax, 3
006c07ef jz  loc_6C0899
```

```
006c07f5

006c07f5 dec eax
006c07f6 jz  loc_6C088A
```

```
006c0899

006c0899 push edi
006c089a xor edi, edi
006c089c cmp [esi+28h], ebx
006c089f jbe short loc_6C08B7
```

```
006c08ea

EDI: 000b933c EAX: 000b9308 EDX: 000b9308
EBX: 00000000 ESI: 00000000 ECX: 00ca5950

 *EDI [heap]  06 9d000005 00080000 010c004c 006f00
 *EAX [heap]  06 00000070 51ca0000 00000000 000000
 *EDX [heap]  06 00000070 51ca0000 00000000 000000
 *ECX [heap]  f8 396b00c8 396b00b8 396b0001 000000
006c08ea pop esi
006c08eb xor eax, eax
006c08ed pop ebx
006c08ee retn 4
```

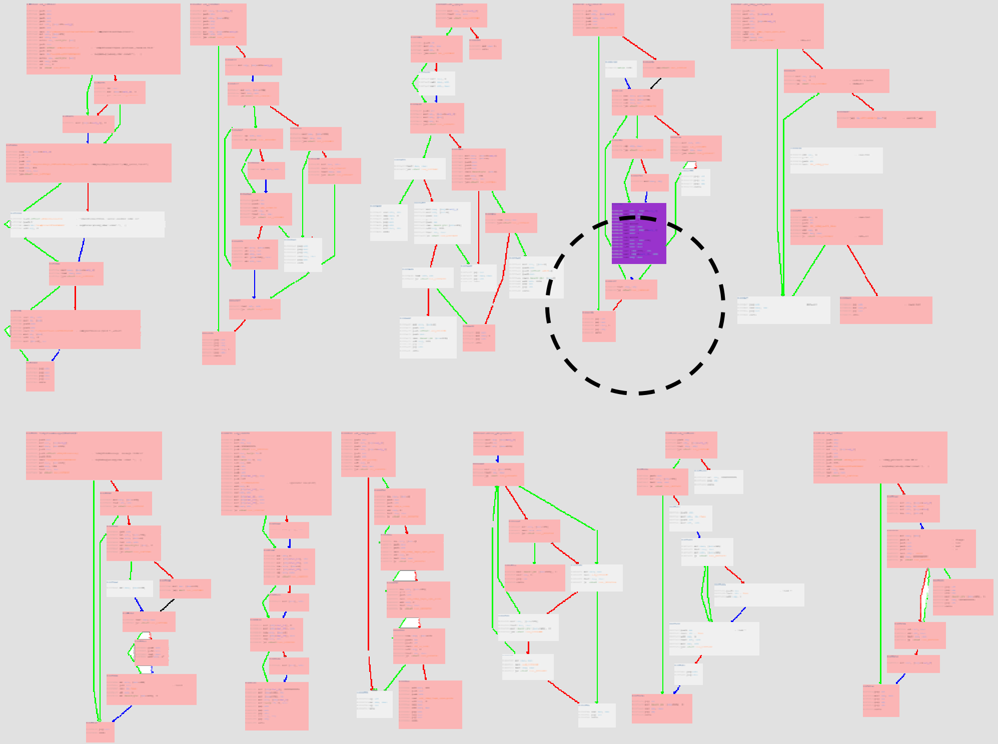
Hierarchical layout

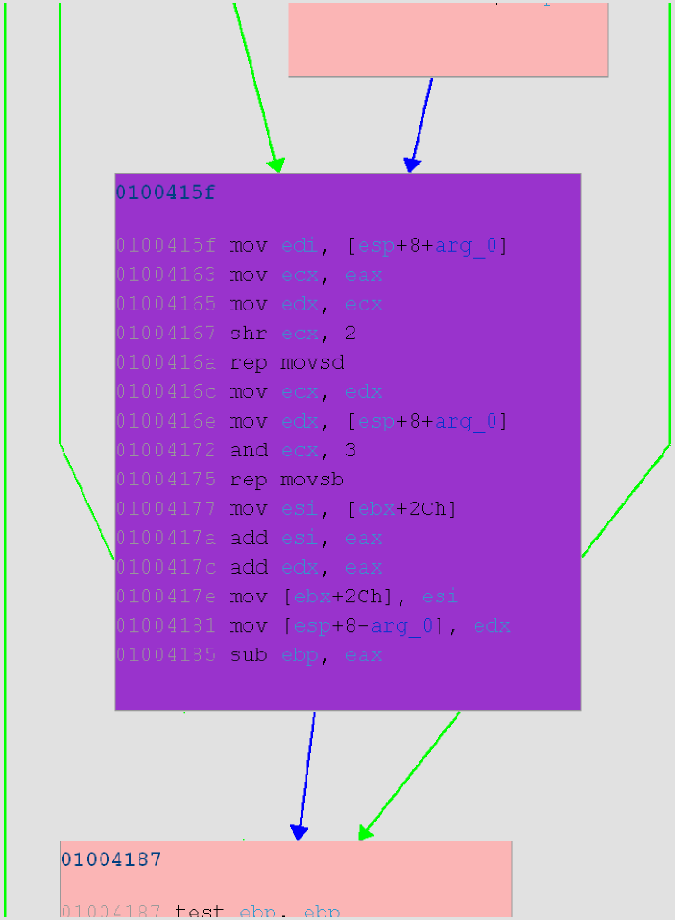


Cluster orthogonal layout

Automated Highlighting

- Potentially interesting nodes are automatically highlighted
- ex: reps, *str*, *wcs*, *alloc*, *mem*



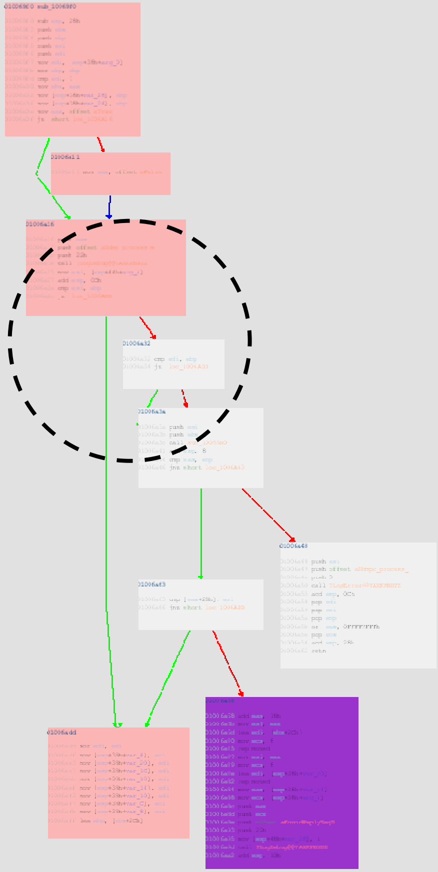


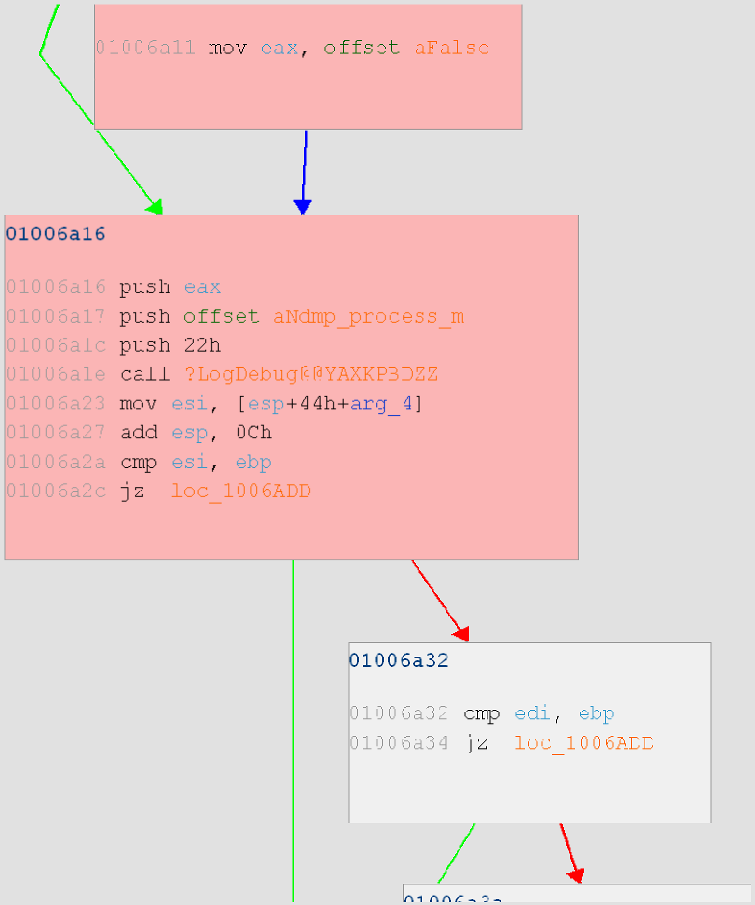
```
0100415f

0100415f mov edi, [esp+8+arg_0]
01004163 mov ecx, eax
01004165 mov edx, ecx
01004167 shr ecx, 2
0100416a rep movsd
0100416c mov ecx, edx
0100416e mov edx, [esp+8+arg_0]
01004172 and ecx, 3
01004175 rep movsb
01004177 mov esi, [ebx+2Ch]
0100417a add esi, eax
0100417c add edx, eax
0100417e mov [ebx+2Ch], esi
01004181 mov [esp+8-arg_0], edx
01004185 sub ebp, eax
```

```
01004187

01004187 test ebp, ebp
```

Alternative Paths

- Easily view and examine branch conditions
- Determine changes required to get fuzzer "deeper" into process state





```
01006a11 mov eax, offset aFalse
```

```
01006a16

01006a16 push eax
01006a17 push offset aNdmp_process_m
01006a1c push 22h
01006a1e call ?LogDebug@@YAXKPBDZZ
01006a23 mov esi, [esp+44h+arg_4]
01006a27 add esp, 0Ch
01006a2a cmp esi, ebp
01006a2c jz  loc_1006ADD
```

```
01006a32

01006a32 cmp edi, ebp
01006a34 jz  loc_1006ADD
```

Folding for Loop Visualization

- Algorithms exist for detecting logical loops statically
- ps_view_recording_trace applies a basic sequence folding routine to cluster repeat sequences
- Using a sliding window the input sequence is traversed. The longest possible discovered sequences are added to a cluster
- Viewing with cluster orthogonal generates a hideous graph. However, the logical loops are easy to spot and analyze
- Quick demo

Speed

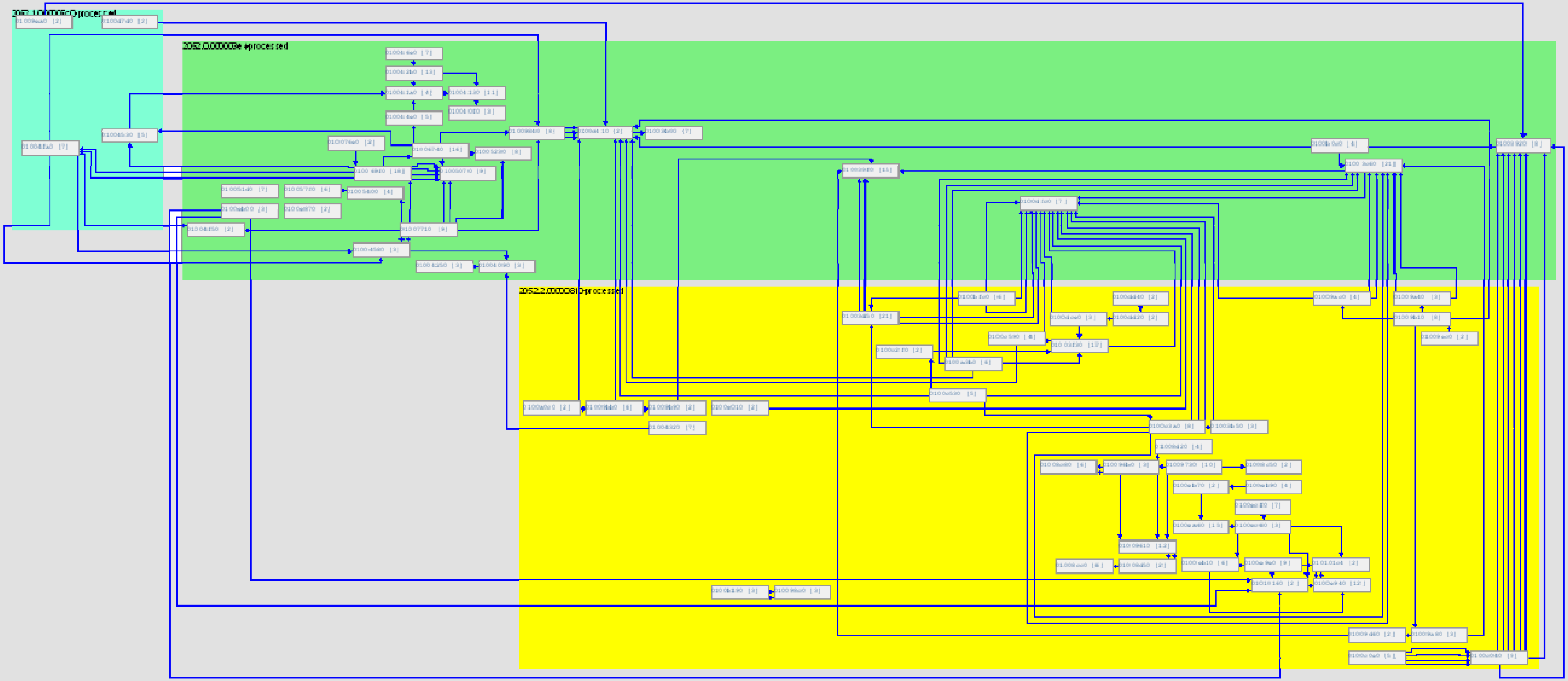
- Much faster than single-step tracing

- Two modes of operation
  - Breakpoint restore
  - One shot

- Breakpoint filtering can further improve performance
  - Functions only
  - Potentially interesting modules only

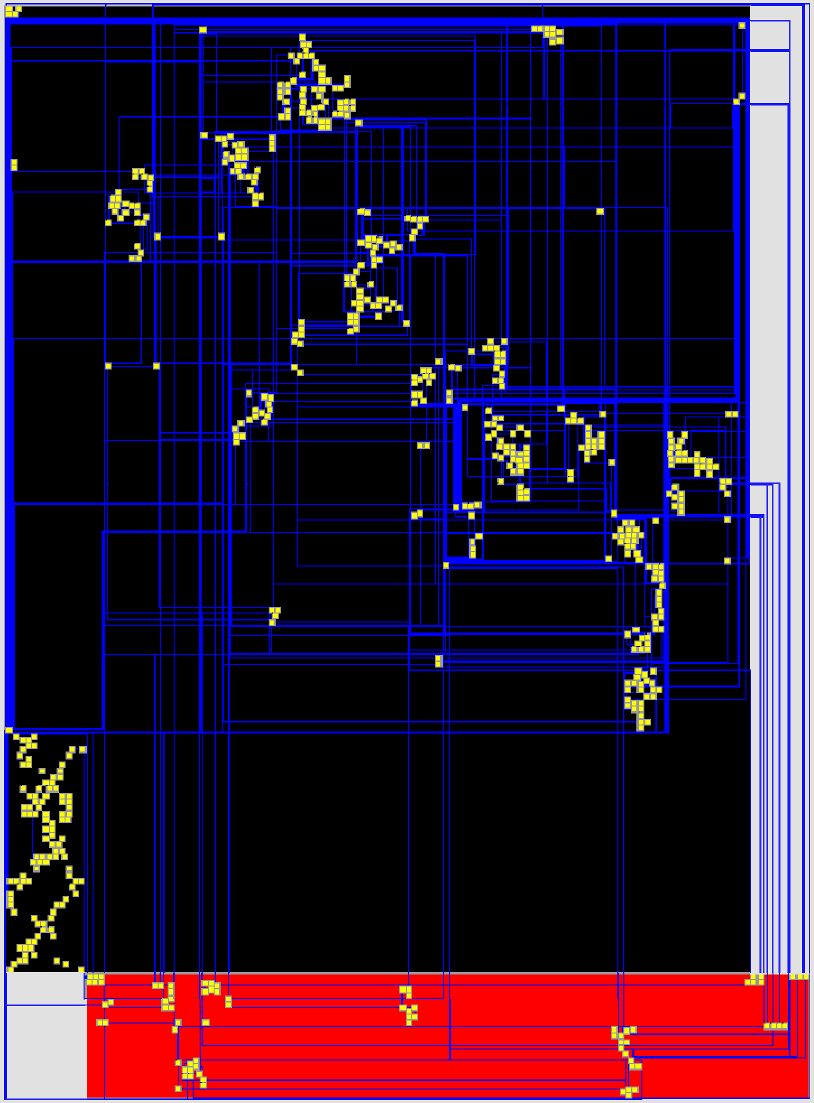
State Mapping

- ex: Authenticated vs. non-authenticated code
- ex: What our fuzzer has reached vs. what our fuzzer can reach



Filtering

- Recordings can be joined and/or diffed
- Example: GUI handling code can be recorded and diffed out
- MS05-030: MSOE.DLL
  - Black: GUI functions
  - Red: Non-GUI functions
- The graph on the right was generated using state mapping



Recording Statistics

- Node hit counts
- Node transition times

```
$ ps_view_recording_stats 2284.0.000003d8-processed

function block hit counts for module irc.dll

         46011500    5   46014e81    1   46012510    4
         4600b010    2   460179e0    1   4600ae70    4
         4601559e    1   46006820    4   46006630   24
         ...

function transition times (milliseconds) for module irc.dll

         4600f560   40   460067e0    0   4600f560    0
         46006820    0   46006630    0   4601559e    0
         46006630   21   4600f560   60   460067e0   10
         4600f560    0   46001690    0   4600f560    0
         ...
```

Demonstration

Command Line Arguments

```
$ process_stalker
process stalker
pedram amini <pedram.amini@gmail.com>
compiled on Jun 14 2005

usage:
  process_stalker <-a pid | -l filename | -la filename args>

options:
  [-b bp list]   specify the breakpoint list for the main module.
  [-r recorder]  enter a recorder (0-9) from trace initiation.
  [--one-time]   disable breakpoint restoration.
  [--no-regs]    disable register enumeration / dereferencing.
```

More Commands

```
$ ps_process_recording gui_shit

$ cat gui_shit.* > gui_shit.processed

$ wc -l gui_shit.processed
  4455 gui_shit.processed

$ time ps_bp_filter msoe.dll.bpl msoe.dll.nogui \
  `ps_recording_to_list gui_shit.processed msoe.dll` out
  real    0m28.367s

$ wc -l msoe.dll.bpl msoe.dll.nogui
  58165 msoe.dll.bpl
  50560 msoe.dll.nogui

$ time ps_view_recording_funcs 844.1.processed > hitgraph.gml
  real    0m7.446s

$ time ps_graph_highlight -nodes hit hitgraph.gml > hitgraph_hl.gml
  real    0m5.795s

$ time ps_add_register_metadata 844-regs.1 hitgraph_hl.gml > with_regs.gml
  real 0m7.977s
```

In Development

- Still working on this stuff:
  - Argument dereferencing
    - With automatic detection of ASCII and Unicode strings
  - Smarter highlighting
  - PDB parsing for when you have source code (hit lines)

- Other ideas:
  - Arbitrary data structure visualization
  - Data flow visualization

- Potential design changes:
  - Remove dependency on IDA
  - Switch from debugger to emulation instrumentation (BOCHS)

www.OpenRCE.org

Open Reverse Code Engineering Community Website