BINARY INSTRUMENTATION FOR SECURITY PROFESSIONALS

GAL DISKIN / INTEL

black hat®
BRIEFINGS & TRAINING
USA + 2011
EMBEDDING SECURITY

Please Remember to Complete Your Feedback Form

blackhat®
BRIEFINGS & TRAINING

USA + 2011
EMBEDDING SECURITY

LEGAL DISCLAIMER

ALL CODE IN THIS PRESENTATION IS COVERED BY THE FOLLOWING:

WHO AM I

» Currently @ Intel
  - Security researcher
  - Evaluation team leader

» Formerly a member of the binary instrumentation team @ Intel

» Before that a private consultant

» Always a hacker

» …

Online presence: www.diskin.org, @gal_diskin, LinkedIn, E-mail (yeah, even FB & G+)

ABOUT THIS WORKSHOP

» Intro to DBI and its information security usages

» How does DBI work – Intro to a DBI engine (Pin)

» InfoSec DBI tools

INSTRUMENTATION

» Source / Compiler Instrumentation

» Static Binary Instrumentation

» Dynamic Binary Instrumentation

What is it good for?

DBI USAGES

WHAT NON-SECURITY PEOPLE USE DBI FOR

» Simulation / Emulation

» Performance analysis

» Correctness checking

» Memory debugging

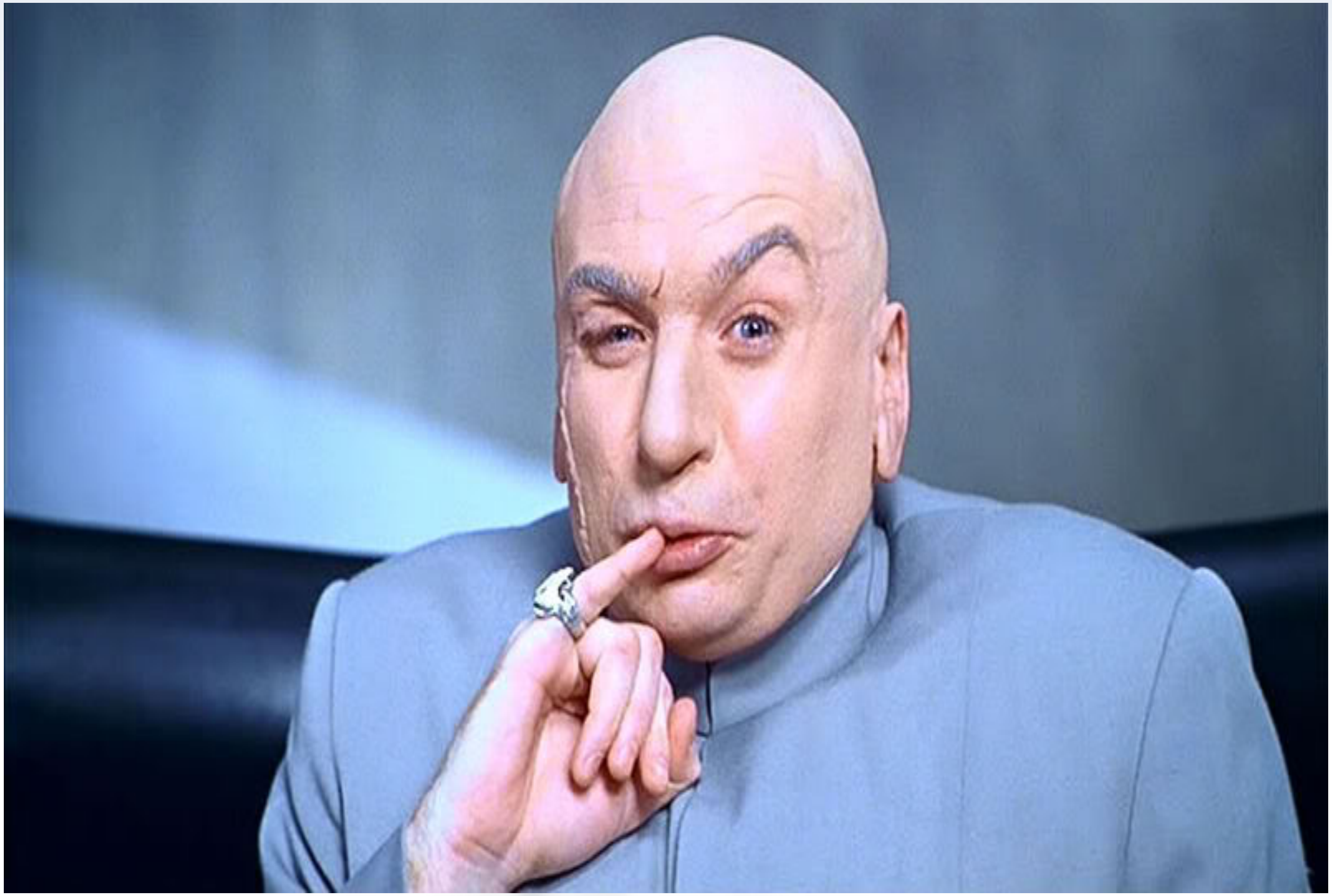
» Parallel optimization

» Call graphs

» Collecting code metrics

» Automated debugging

WHAT DO WE WANT TO USE IT FOR?



GETTING A JOB

» Ad is © Rapid7/jduck

- Developing exploits using the Metasploit Framework
- Reverse engineering compiled applications
- SMT/SAT solvers
- Various run-time analysis techniques
- Dynamic Binary Instrumentation/Translation
- Fuzz-testing
- Programming in other assembly languages, such as ARM, PPC, SPARC, MIPS
- Embedded device research and exploitation

Exploit Engineer Wanted: Get Paid for Open Source

Posted by Rapid7 Staff on Jan 26, 2011 8:10:00 AM

Originally Posted by jduck

After the incredible success of the Metasploit Express and Metasploit Pro product launches last year, we are happy to announce a new position on the Rapid7 Metasploit team. Effective immediately, we are seeking a self-driven Exploit Engineer to join the team of full-time Metasploit developers.

Job duties include researching vulnerabilities and writing exploit code in the form of Metasploit modules (Ruby). Exploit modules will be released to the public under the BSD open source license.

The ideal candidate will primarily work from home, but will meet with team members approximately once a week in Austin, TX. However, exceptions may be made for the perfect candidate. **Candidates must have the right to work in the United States.**

Benefits include:

- Competitive salary and bonus plan
- Health care and medical benefits
- Paid to contribute to an open-source project
- Exploits publicly released under BSD license

A candidate must have a solid understanding of:

- Common vulnerability classes
- State-of-the-art exploitation techniques
- Programming in Ruby, C, C++, and x86 assembly
- Common networking protocols (TCP/IP and related protocols)
- Network and system administration of a lab environment
- Using debuggers and disassemblers (WinDbg, IDA Pro)
- Binary patch diffing (BinDiff or otherwise)
- Common operating system implementations (Windows, Linux, etc)

In addition to the requirements, we prefer candidates who have experience:

- Developing exploits using the Metasploit Framework
- Reverse engineering compiled applications
- SMT/SAT solvers
- Various run-time analysis techniques
- Dynamic Binary Instrumentation/Translation
- Fuzz-testing
- Programming in other assembly languages, such as ARM, PPC, SPARC, MIPS
- Embedded device research and exploitation

All interested parties should email their resumes to jobs[at]metasploit.com.

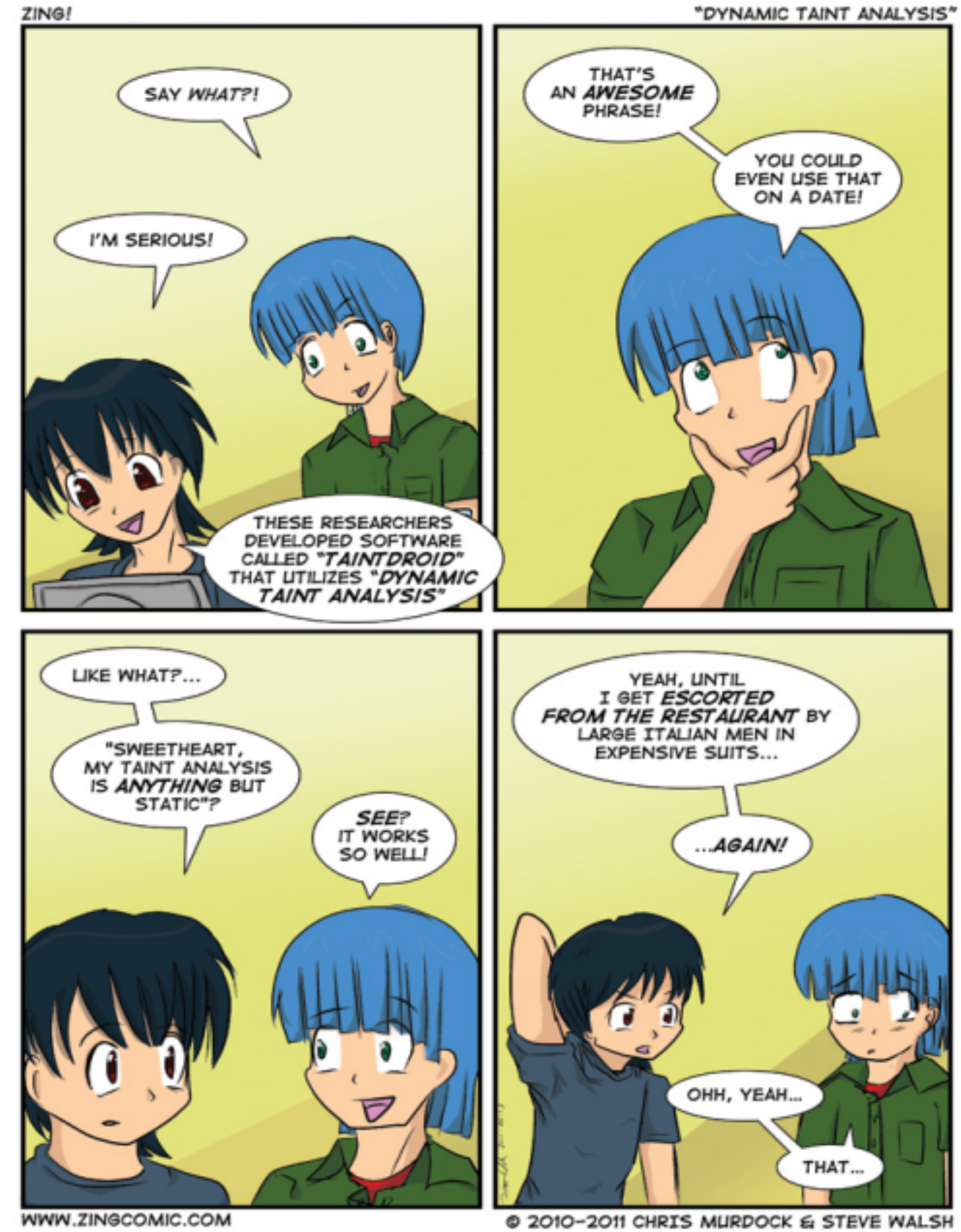
TAINT ANALYSIS

» Following tainted data flow through programs

» Transitive property

$$X \in T(Y) \wedge Z \in T(X) \rightarrow Z \in T(Y)$$

$$(x<y) \wedge (z<x) \rightarrow (z<y)$$



TAINT (DATA FLOW) ANALYSIS

» Data flow analysis
  - Vulnerability research
  - Privacy
» Malware analysis
» Unknown vulnerability detection
» Test case generation
» …

TAINT (DATA FLOW) ANALYSIS

» Edgar Barbosa in H2HC 2009

» Flayer

» Some programming languages have a taint mode

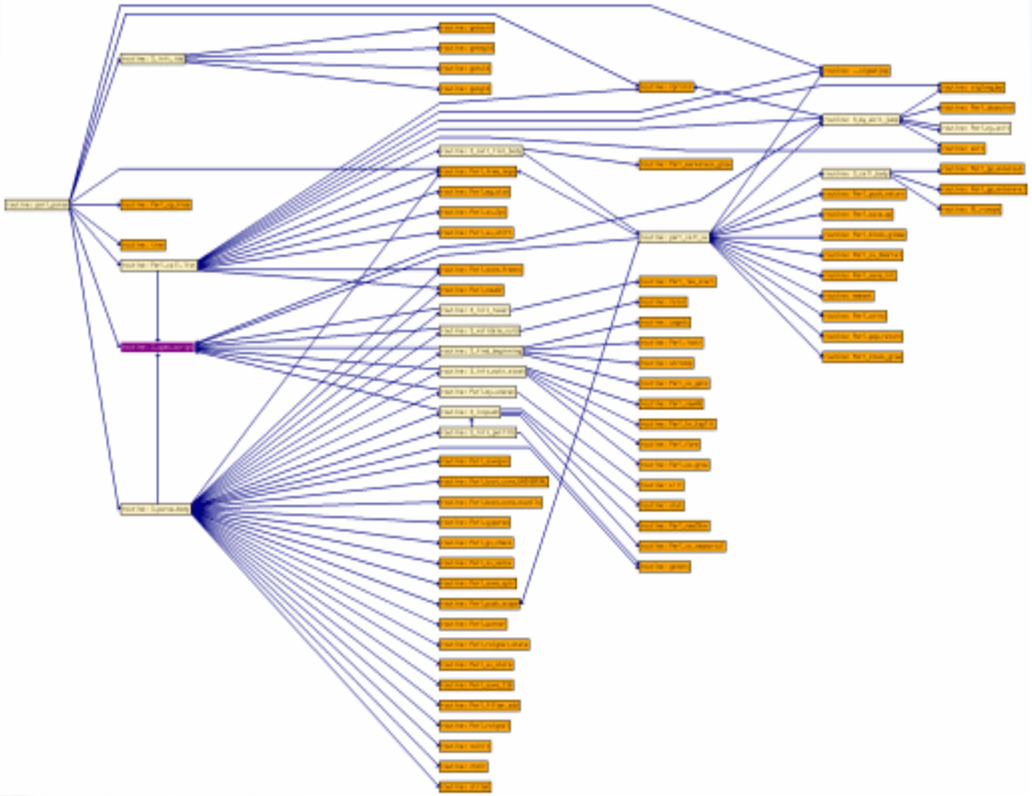
CONTROL FLOW ANALYSIS

» Call graphs

» Code coverage

» Examples:

- [Pincov]



PRIVACY MONITORING

» Relies on taint analysis

- Source = personal information
- Sink = external destination

» Examples:

- [Taintdroid]
- [Privacy Scope]

KNOWN VULNERABILITY DETECTION

» Detect exploitable **condition**

- Double free
- Race condition
- Dangling pointer
- Memory leak

UNKNOWN VULNERABILITY DETECTION

» Detect exploit **behavior**

- Overwriting a return address
- Corruption of meta-data
  - E.g. Heap descriptors
- Execution of user data
- Overwrite of function pointers

VULNERABILITY DETECTION

» Examples:

- Intel ® Parallel Studio
- Determina

FUZZING / SECURITY TEST CASE GENERATION

» Feedback driven fuzzing
  - Code coverage driven
    - Corpus distillation
  - Data coverage driven
    - Haven't seen it in the wild
  - Constraints
  - Evolutionary fuzzing
» Checkpointing
» In-memory fuzzing
» Event / Fault injection

FUZZING / SECURITY TEST CASE GENERATION

» Examples:

- Tavis Ormandy @ HITB'09
- Microsoft SAGE

AUTOMATED EXPLOIT DEVELOPMENT

» Known exploit techniques

» SAT/SMT

AUTOMATED VACCINATIONS

» Detecting attacks

» Introducing diversity

» Adaptive self-regenerative systems

» Examples:

- Sweeper
- GENESIS

PRE-PATCHING OF VULNERABILITIES

» Modify vulnerable binary code

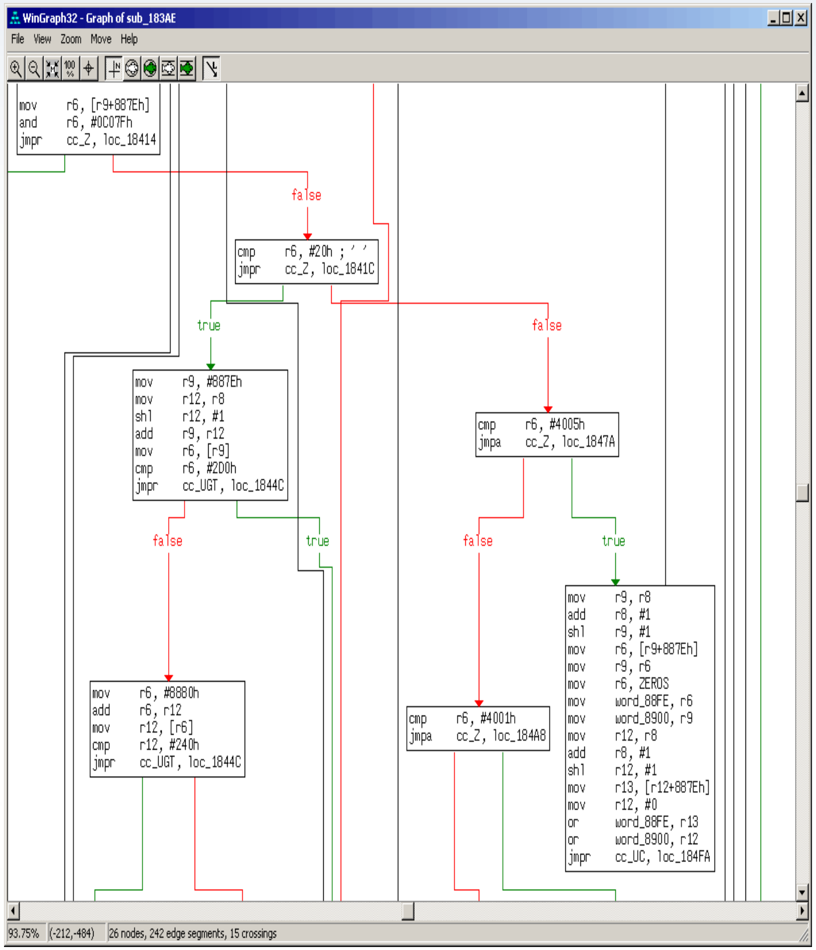
» Insert additional checks

» Example:

- Determina LiveShield

REVERSING

- » De-obfuscation / unpacking
- » Frequency analysis
- » SMC analysis
- » Automated lookup for behavior / functions
- » Differential analysis / equivalence analysis



REVERSING

» Examples:

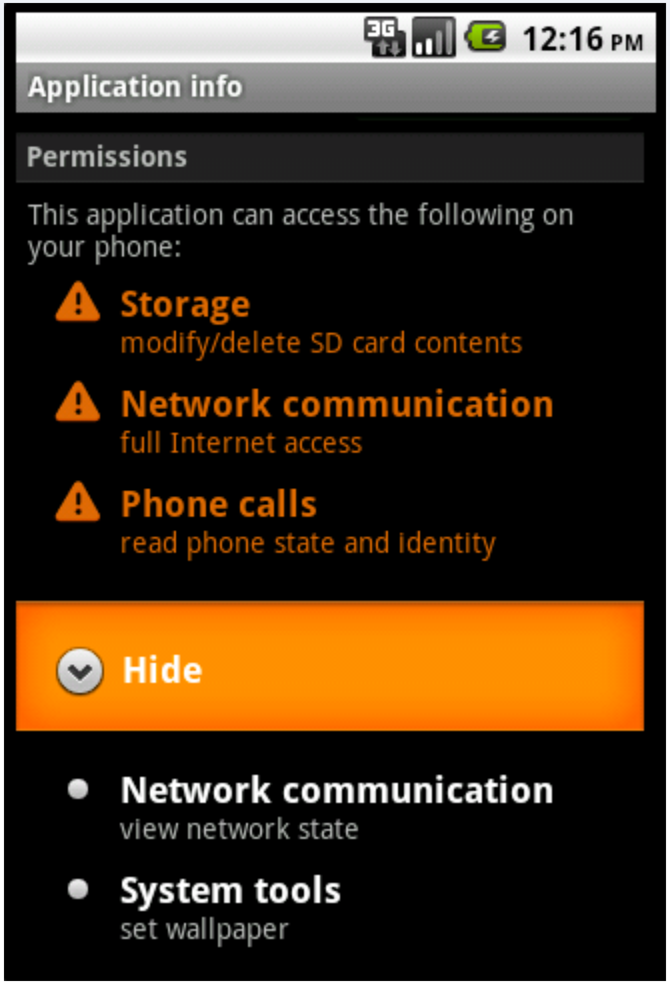
- Covert debugging / Danny Quist & Valsmith @ BlackHat USA 2007
- Black Box Auditing Adobe Shockwave - Aaron Portnoy & Logan Brown
- tartetatintools
- Automated detection of cryptographic primitives

TRANSPARENT DEBUGGING

- Hiding from anti-debug techniques
- Anti-instrumentation
- Anti-anti instrumentation

BEHAVIOR BASED SECURITY

- Creating legit behavior profiles and allowing programs to run as long as they don't violate those
- Alternatively, looking for backdoor / Trojan behavior
- Examples:
  - [HTH – Hunting Trojan Horses]



OTHER USAGES

- Vulnerability classification
- Anti-virus technologies
- Forcing security practices
  - Adding stack cookies
  - Forcing ASLR
- Sandboxing
- Forensics

SECTION SUMMARY

- Data & Control flow analysis
- Privacy
- Vulnerability detection
- Fuzzing
- Automated exploitation
- Reverse engineering & Transparent debugging
- Behavior based security
- Pre-patching

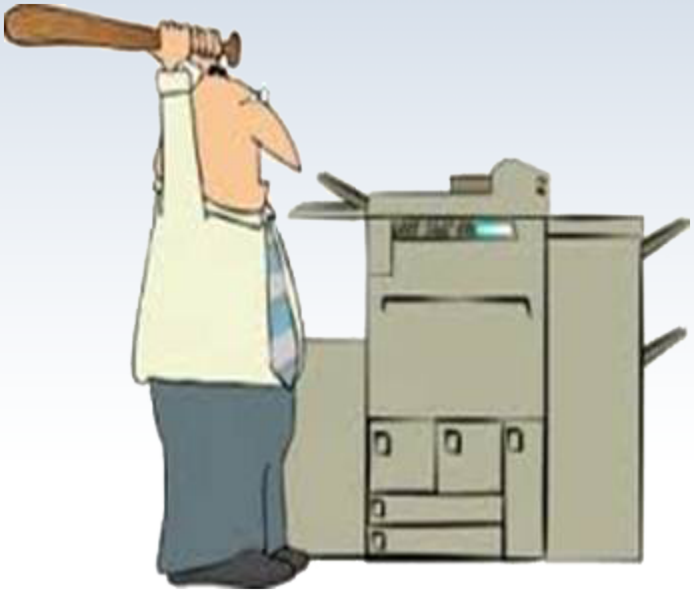
I told you DBI is wonderful - what's next?

INTRO TO A DBI ENGINE AND HOW IT WORKS

BINARY INSTRUMENTATION ENGINES

- Pin
- DynamoRio
- Valgrind
- ERESI
- Many more…

PIN & PINTOOLS



- Pin – the instrumentation **engine**
  - JIT for x86
- PinTool – the instrumentation **program**
- PinTools register **hooks** on events in the program
  - **Instrumentation routines** – called **only** on the **first** time something happens
  - **Analysis routines** – called **every** time this object is reached
  - **Callbacks** – called **whenever** a certain **event** happens

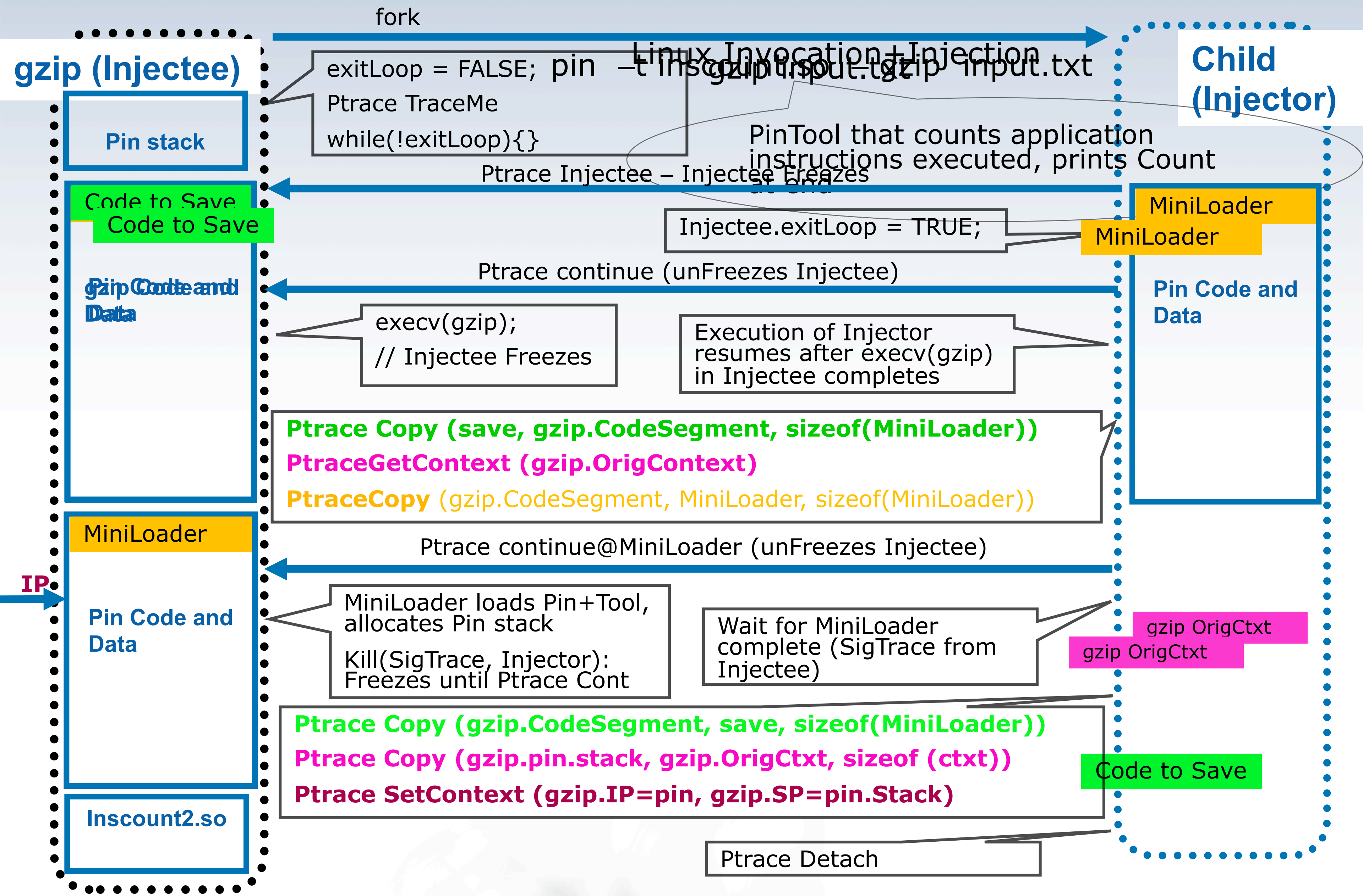
WHERE TO FIND INFO ABOUT PIN

- Website: www.pintool.org
- Mailing list @ Yahoo groups: Pinheads

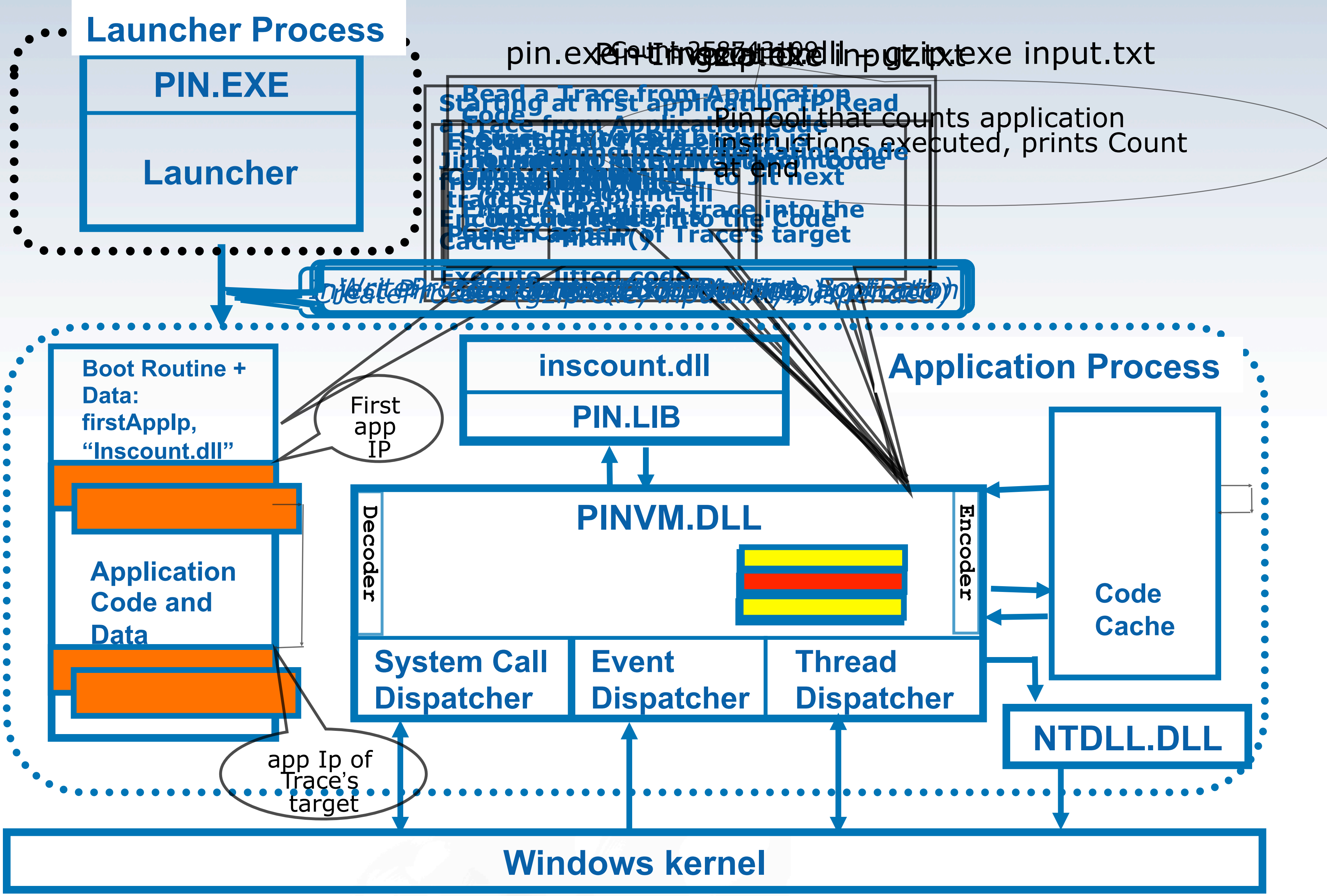
A PROGRAM'S BUILDING BLOCKS

- Instruction
- Basic Block
- Trace or Super-block

PIN INJECTION


fork
gzip (Injectee)
Child (Injector)
Linux Invocation+Injection
pin –t inscount.so – gzip input.txt
gzip input.txt
exitLoop = FALSE;
Ptrace TraceMe
while(!exitLoop){}
Pin stack
PinTool that counts application instructions executed, prints Count at end
Ptrace Injectee – Injectee Freezes
Code to Save
Code to Save
MiniLoader
MiniLoader
Injectee.exitLoop = TRUE;
Ptrace continue (unFreezes Injectee)
gzip Code and Data
Pin Code and Data
Pin Code and Data
execv(gzip);
// Injectee Freezes
Execution of Injector resumes after execv(gzip) in Injectee completes
Ptrace Copy (save, gzip.CodeSegment, sizeof(MiniLoader))
PtraceGetContext (gzip.OrigContext)
PtraceCopy (gzip.CodeSegment, MiniLoader, sizeof(MiniLoader))
MiniLoader
Ptrace continue@MiniLoader (unFreezes Injectee)
IP
Pin Code and Data
MiniLoader loads Pin+Tool, allocates Pin stack
Kill(SigTrace, Injector): Freezes until Ptrace Cont
Wait for MiniLoader complete (SigTrace from Injectee)
gzip OrigCtxt
gzip OrigCtxt
Ptrace Copy (gzip.CodeSegment, save, sizeof(MiniLoader))
Ptrace Copy (gzip.pin.stack, gzip.OrigCtxt, sizeof (ctxt))
Ptrace SetContext (gzip.IP=pin, gzip.SP=pin.Stack)
Code to Save
Inscount2.so
Ptrace Detach

PIN EXECUTION



Launcher Process
PIN.EXE
Launcher
PinTool that counts application instructions executed, prints Count at end
Boot Routine + Data: firstAppIp, “Inscount.dll”
First app IP
Application Code and Data
app Ip of Trace’s target
inscount.dll
PIN.LIB
Application Process
PINVM.DLL
Decoder
Encoder
Code Cache
System Call Dispatcher
Event Dispatcher
Thread Dispatcher
NTDLL.DLL
Windows kernel

SECTION SUMMARY

» There are many DBI engines

» We're focusing on Pin in this workshop

» We've seen how Pin injection into a process works

» We've seen how it behaves during execution

How do you program a DBI engine?

INTRO TO PINTOOLS

PINTOOL 101: INSTRUCTION COUNTING

```
#include "pin.h"

UINT64 icount = 0;

void docount() { icount++; }

void Instruction(INS ins, void *v)
{
   INS_InsertCall(ins, IPOINT_BEFORE,
   (AFUNPTR)docount, IARG_END);
}

void Fini(INT32 code, void *v)
{ std::cerr << "Count " << icount << endl; }

int main(int argc, char * argv[])   {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();  // Never returns
   return 0;
}
```

Execution time routine

Jitting time routine

```
switch to pin stack
save registers
call docount
restore regs & stack
inc icount
```

- `sub    $0xff, %edx`

```
inc icount
```

- `cmp    %esi, %edx`

```
save eflags
inc icount
restore eflags
```

- `jle    <L1>`

```
inc icount
```

- `mov    0x1, %edi`

PIN COMMAND LINE

- Pin -pin_switch1 … -t pintool.so -tool_switch1 … -- program program_arg1 …

- Pin provides PinTools with a way to parse the command line using the KNOB class

HOOKS

» The heart of Pin's approach to instrumentation

» Analysis and Instrumentation

» Can be placed on various events / objects, e.g:

- Instructions
- Context switch
- Thread creation
- Much more…

INSTRUMENTATION AND ANALYSIS

» Instrumentation

- Usually defined in the tool “main”
- Once per object
- Heavy lifting

» Analysis

- Usually defined in instrumentation routine
- Every time the object is accessed
- As light as possible

GRANULARITY

» INS – Instruction

» BBL – Basic Block

» TRACE – Trace

» RTN – Routine

» SEC – Section

» IMG – Binary image

OTHER INSTRUMENTABLE OBJECTS

» Threads

» Processes

» Exceptions and context changes

» Syscalls

» …

INSTRUCTION COUNTING: TAKE 2

```
#include "pin.H"

UINT64 icount = 0;

void PIN_FAST_ANALYSIS_CALL docount(INT32 c) { icount += c; }

void Trace(TRACE trace, void *v){// Pin Callback
  for(BBL bbl = TRACE_BblHead(trace);
      BBL_Valid(bbl);
      bbl = BBL_Next(bbl))
      BBL_InsertCall(bbl, IPOINT_ANYWHERE,
                     (AFUNPTR)docount, IARG_FAST_ANALYSIS_CALL,
                     IARG_UINT32, BBL_NumIns(bbl),
                     IARG_END);
}

void Fini(INT32 code, void *v) {// Pin Callback
  fprintf(stderr, "Count %lld\n", icount);
}

int main(int argc, char * argv[]) {
   PIN_Init(argc, argv);
   TRACE_AddInstrumentFunction(Trace, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();
   return 0;
}
```

INSTRUMENTATION POINTS

» IPOINT_BEFORE
  • Before an instruction or routine
» IPOINT_AFTER
  • Fall through path of an instruction
  • Return path of a routine
» IPOINT_ANYWHERE
  • Anywhere inside a trace or a BBL
» IPOINT_TAKEN_BRANCH
  • The taken edge of branch

INLINING

Inlinable

```
int docount0(int i) {
    x[i]++
    return x[i];
}
```

Not-inlinable

```
int docount1(int i) {
    if (i == 1000)
        x[i]++;
    return x[i];
}
```

Not-inlinable

```
int docount2(int i) {
    x[i]++;
    printf("%d", i);
    return x[i];
}
```

Not-inlinable

```
void docount3() {
    for(i=0;i<100;i++)
        x[i]++;
}
```

INLINING

» –log_inline records inlining decisions in pin.log

```
Analysis function (0x2a9651854c) from mytool.cpp:53  INLINED
Analysis function (0x2a9651858a) from mytool.cpp:178 NOT INLINED
    The last instruction of the first BBL fetched is not a ret
instruction
```

» The disassembly of an un-inlined analysis function

```
0x0000002a9651858a push rbp
0x0000002a9651858b mov rbp, rsp
0x0000002a9651858e mov rax, qword ptr [rip+0x3ce2b3]
0x0000002a96518595 inc dword ptr [rax]
0x0000002a96518597 mov rax, qword ptr [rip+0x3ce2aa]
0x0000002a9651859e cmp dword ptr [rax], 0xf4240
0x0000002a965185a4 jnz 0x11
```

» The function could not be inlined because it contains a control-flow changing instruction (other than ret)

CONDITIONAL INSTRUMENTATION

- *XXX*_InsertIfCall
- *XXX*_InsertThenCall

LIVENESS ANALYSIS

» Not all registers are used by each program

» Pin takes control of "dead" registers

- Used for both Pin and tools

» Pin transparently reassigns registers

HOW TRANSLATED CODE LOOKS?

```
    APP IP
 2 0x77ec4600 cmp    rax, rdx
22 0x77ec4603 jz     0x77f1eac9
40 0x77ec4609 movzx ecx, [rax+0x2]
37 0x77ec460d call   0x77ef7870
```

Application Trace
How many BBLs in this trace?

Compiler generated code for docount Inlined by Pin
r14 allocated by Pin

r15 allocated by Pin

Points to per-thread spill area

```
20 0x001de0000 mov r14, 0xc5267d40    //inscount2.docount
58 0x001de000a add  [r14], 0x2        //inscount2.docount
 2 0x001de0015 cmp rax, rdx
 9 0x001de0018 jz 0x1deffa0 (PIN-VM)  //patched in future
52 0x001de001e mov r14, 0xc5267d40    //inscount2.docount
29 0x001de0028 mov  [r15+0x60], rax
57 0x001de002c lahf                         save
37 0x001de002e seto al                      status
50 0x001de0031 mov  [r15+0xd8], ax          flags
30 0x001de0039 mov rax,  [r15+0x60]
12 0x001de003d add  [r14], 0x2         //inscount2.docount
40 0x001de0048 movzx edi, [rax+0x2]  //ecx alloced to edi
22 0x001de004c push 0x77ec4612    //push retaddr
61 0x001de0051 nop
17 0x001de0052 jmp 0x1deffd0 (PIN-VM)//patched in future
```

(

SECTION SUMMARY

» The “Hello (DBI) World” is instruction counting

» There are various levels of granularity we can instrument as well as various points we can instrument in

» Instrumentation routines are called once, analysis routines are called every time

» Performance is better when working at higher granularity, when your heavy work is done in instrumentation routines and when your code is inline-able or you use conditional instrumentation

When we can't start the process ourselves

ATTACHING AND DETACHING

ATTACHING TO A RUNNING PROCESS

- Simply add "-pid <PID#>" command line option instead of giving a program at the end of command line
  - pin –pid 12345 –t MyTool.so
- Related APIs:
  - PIN_IsAttaching
  - IMG_AddInstrumentFunction
  - PIN_AddApplicationStartFunction

DETACHING

- Pin can also detach from the application
- Related APIs:
  - PIN_Detach
  - PIN_AddDetachFunction

We don't want to concentrate on instructions all the time.

SYMBOLS, FUNCTIONS & PROBES

SYMBOLS

- Function symbols
- Debug symbols

- Stripped executables

- Init APIs:
  - PIN_InitSymbols
  - PIN_InitSymbolsAlt

SYMBOL API

- SYM_Next
- SYM_Prev
- SYM_Name
- SYM_Invalid
- SYM_Valid
- SYM_Dynamic
- SYM_IFunc
- SYM_Value
- SYM_Index
- SYM_Address
- PIN_UndecorateSymbolName

BACK TO THE SOURCE LINE

- PIN_GetSourceLocation (

```
ADDRINT   address,
INT32 *   column,
INT32 *   line,
string *  fileName )
```

FUNCTION REPLACEMENT

» RTN_Replace

- Replace app function with tool function

» RTN_ReplaceSignature

- Replace function and modify its signature

» PIN_CallApplicationFunction

- Call the application function and JIT it

PROBE MODE

» JIT Mode
  - Code translated and translation is executed
  - Flexible, slower, robust, common

» Probe Mode
  - Original code is executed with "probes"
  - Faster, less flexible, less robust

PROBE SIZE

Copy of entry point with original bytes:

```
0x50000004:   push   %ebp
0x50000005:   mov    %esp,%ebp
0x50000007:   push   %edi
0x50000008:   push   %esi
0x50000009:   jmp    0x400113d9
```

Original function entry point:

Entry point overwritten with probe:

```
0x400113d4:  push   %ebp
0x400113d4:  jmp    0x41481064
0x400113d5:  mov    %esp,%ebp
0x400113d7:  push   %edi
0x400113d8:  push   %esi
0x400113d9:  push   %ebx
```

```
0x41481064:   push   %ebp       // tool wrapper func
::::::::::::::::::::
          :   call 0x50000004  // call original func
```

OUT OF MEMORY FAULT INJECTION

- The following example will show how to use probe mode to randomly inject out of memory errors into programs

```
#include "pin.H"
#include <time.h>
#include <iostream>
// Injected failure “frequency”
#define FAIL_FREQ 100
typedef VOID * ( *FP_MALLOC )( size_t );

// This is the malloc replacement routine.
VOID * NewMalloc( FP_MALLOC orgFuncptr, UINT32 arg0 )
{
        if ( (rand() % FAIL_FREQ) == 1 )
        {
                return NULL; //force fault
        }
        return orgFuncptr( arg0 ); //call real malloc and return value
}
```

```
// Pin calls this function every time a new img is loaded.
// It is best to do probe replacement when the image is loaded,
// because only one thread knows about the image at this time.
VOID ImageLoad( IMG img, VOID *v )
{
   // See if malloc() is present in the image.  If so, replace it.
   RTN rtn = RTN_FindByName( img, "malloc" );

   if (RTN_Valid(rtn))
   {
      // Define a function prototype of the orig func
      PROTO proto_malloc = PROTO_Allocate( PIN_PARG(void *),
                           CALLINGSTD_DEFAULT, "malloc",
                           PIN_PARG(int), PIN_PARG_END() );

      // Replace the application routine with the replacement function.
      RTN_ReplaceSignatureProbed(rtn, AFUNPTR(NewMalloc),
                                 IARG_PROTOTYPE, proto_malloc,
                                 IARG_ORIG_FUNCPTR,
                                 IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                                 IARG_END);

      // Free the function prototype.
      PROTO_Free( proto_malloc );
   }
}
```

```
int main( INT32 argc, CHAR *argv[] )
{
   // Initialize sumbols
   PIN_InitSymbols();

   // Initialize Pin
   PIN_Init(argc, argv);

   // Initialize RNG
   srand( time(NULL) );

   // Register ImageLoad to be called when an image is loaded
   IMG_AddInstrumentFunction( ImageLoad, 0 );

   // Start the program in probe mode, never returns
   PIN_StartProgramProbed();

    return 0;
}
```

TOOL WRITER RESPONSIBILITIES

» No control flow into the instruction space where probe is placed
  - 6 bytes on IA-32, 7 bytes on Intel64, 1 bundle on IA64
  - Branch into “replaced” instructions will fail
  - Probes at function entry point only

» Thread safety for insertion and deletion of probes
  - During image load callback is safe
  - Only loading thread has a handle to the image

» Replacement function has “same” behavior as original

SECTION SUMMARY

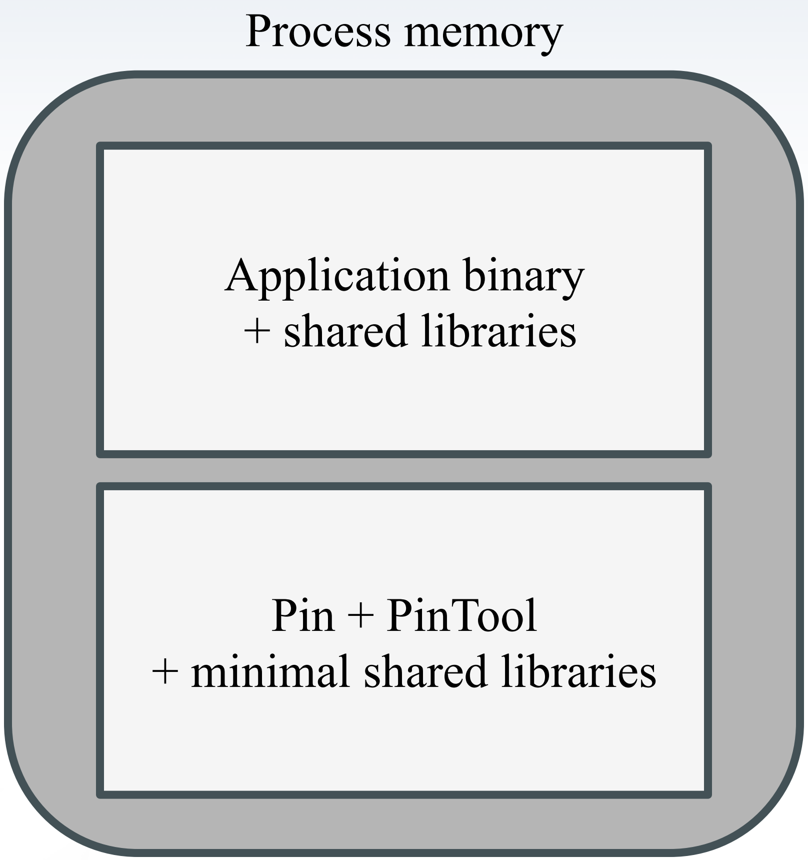
- Pin supports function symbols and has limited support for debug symbols
- Pin supports function replacement
- Probe mode allows you to place probes on functions. It is much faster but less robust and less flexible
- Certain considerations apply when writing tools
- We saw how simple it is to write a pintool to simulate out of memory situations

Some stuff to think about when writing your tools

ISOLATION, RECURSION AND PERFORMANCE

ISOLATION

» How do we isolate two programs loaded in the same process sharing the same virtual memory?



ISOLATION/WINDOWS

» Pin Tools are compiled to use the static CRT

» Pin on Windows does not separate DLLs loaded by the tool from the application DLLs - it uses the same system loader.

- The tool should not load any DLL that can be shared with the application.
- The tool should avoid static links to any common DLL, except for those listed in PIN_COMMON_LIBS (see source\tools\ms.flags file).

ISOLATION/WINDOWS

- Pin on Windows guarantees safe usage of C/C++ run-time services in Pin tools, including indirect calls to Windows API through C run-time library.
  - Any other use of Windows API in Pin tool is not guaranteed to be safe
- Pin uses some base types that conflict with Windows types. If you use "windows.h", you may see compilation errors. So do:

  namespace WINDOWS { #include <windows.h> }

ISOLATION/LINUX

» Pin is injected in to address space and has its own copy of the dynamic loader and runtime libraries (GLIBC, etc).

» Pin uses a small library of CRT for direct calls to system calls.

» The process has a single signals table (shared among all threads), pin manages an internal signal table and emulate all the system calls related to signals.

ISOLATION/LINUX

» pthread functions cannot be called from an analysis or replacement routine

» Pintools on Linux need to take care when calling standard C or C++ library routines from analysis or replacement functions

- Because the C and C++ libraries linked into Pintools are **not** thread-safe

RECURSION IN TOOLS

» Tool function calls an instrumented app function that then calls back to the tool function…

» When does it happen?

- Bad isolation
- Probe mode
- PIN_CallApplicationFunction

TOOL PERFORMANCE

- Analysis vs. Instrumentation
- Inlining
- If-Then instrumentation
- Predicated calls
- Number of args to analysis routines
- Buffering (see next)

BUFFERING

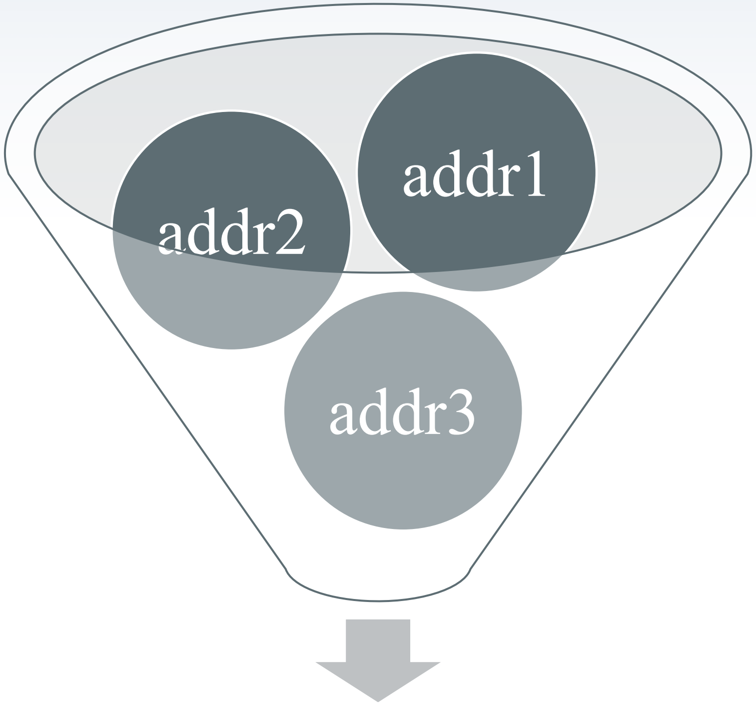
» Managing a (per-thread) buffer is a necessity for a large class of Pin tools

» Pin Buffering API abstracts away the need for a Pin tool to manage (per-thread) buffers

- PIN_DefineTraceBuffer
- INS_InsertFillBuffer

» Tool defines BufferFull function that is called automatically when the buffer becomes full

MEMBUFFER_SIMPLE



Dump buffer contents

```
KNOB<UINT32> KnobNumPagesInBuffer(KNOB_MODE_WRITEONCE,"pintool","num_pages_in_buffer",
                        "256", "number of pages in buffer");

// Struct of memory reference written to the buffer
struct MEMREF {
   ADDRINT appIP;
   ADDRINT memAddr; };
// The buffer ID returned by the one call to PIN_DefineTraceBuffer
BUFFER_ID bufId;
TLS_KEY appThreadRepresentitiveKey;
int main(int argc, char * argv[]) {
    PIN_Init(argc,argv) ;
   // Pin TLS slot for holding the object that represents an application thread
    appThreadRepresentitiveKey = PIN_CreateThreadDataKey(0);
   // Define the buffer that will be used – buffer is allocated to each thread when the thread starts running
    bufId = PIN_DefineTraceBuffer(sizeof(struct MEMREF), KnobNumPagesInBuffer,
                            BufferFull, 0);  // BufferFull is the tool function will be called when buffer is full
INS_AddInstrumentFunction(Instruction, 0);  // The Instruction function will use the Pin Buffering
                                            // API to insert the instrumentation code that writes
                                            // the MEMREF of a memory accessing INS into the buffer
   PIN_AddThreadStartFunction(ThreadStart, 0);
   PIN_AddThreadFiniFunction(ThreadFini, 0);
   PIN_AddFiniFunction(Fini, 0);

   PIN_StartProgram();
}
```

```
/* Pin generates code to call this function when a buffer fills up, and exceutes a callback to this
 * function when the thread exits. Pin will NOT inline this function
 * @param[in] id                    buffer handle
 * @param[in] tid                   id of owning thread
 * @param[in] ctxt                  application context
 * @param[in] buf                   actual pointer to buffer
 * @param[in] numElements           number of records
 * @param[in] v                     callback value
 * @return  A pointer to the buffer to resume filling. */

VOID * BufferFull(BUFFER_ID id, THREADID tid, const CONTEXT *ctxt, VOID *buf,
                  UINT64 numElements, VOID *v)
{
   // retrieve the APP_THREAD_REPRESENTITVE* of this thread from the Pin TLS
   APP_THREAD_REPRESENTITVE * appThreadRepresentitive =
      static_cast<APP_THREAD_REPRESENTITVE*>( PIN_GetThreadData(
                                        appThreadRepresentitiveKey, tid ) );

   appThreadRepresentitive->ProcessBuffer(buf, numElements);

   return buf;
}
```

```
VOID Instruction (INS ins, VOID *v)
{
  UINT32 numMemOperands = INS_MemoryOperandCount(ins);

  // Iterate over each memory operand of the instruction.
  for (UINT32 memOp = 0; memOp < numMemOperands ; memOp++)
  {  // Add the instrumentation code to write the appIP and memAddr
     // of this memory operand into the buffer
     // Pin will inline the code that writes to the buffer
     INS_InsertFillBuffer(ins,  IPOINT_BEFORE,  bufId,
                          IARG_INST_PTR,  offsetof(struct MEMREF, appIP),
                          IARG_MEMORYOP_EA,  memOp,
                          offsetof(struct MEMREF,  memAddr),
                          IARG_END);
  }
}
```

INSTRUMENTATION ORDER

- What happens when multiple analysis routines are attached to the same instruction?
- IARG_CALL_ORDER

XED – X86 ENCODER DECODER

- XED is the decoder and encoder Pin uses
- XED documentation also available in www.pintool.org
- Can be used as a stand-alone tool as well
  - As a library
  - As a command line tool

SECTION SUMMARY

- There are many things to consider when writing PinTools
- It is important to be careful of isolation and recursion problems with your tools
- Tool performance is affected by many things
- Using the buffering API can help tool performance
- We saw a simple buffering tool to record memory accesses
- We can control the instrumentation order when placing multiple analysis calls on the same object

Simple, yet powerful

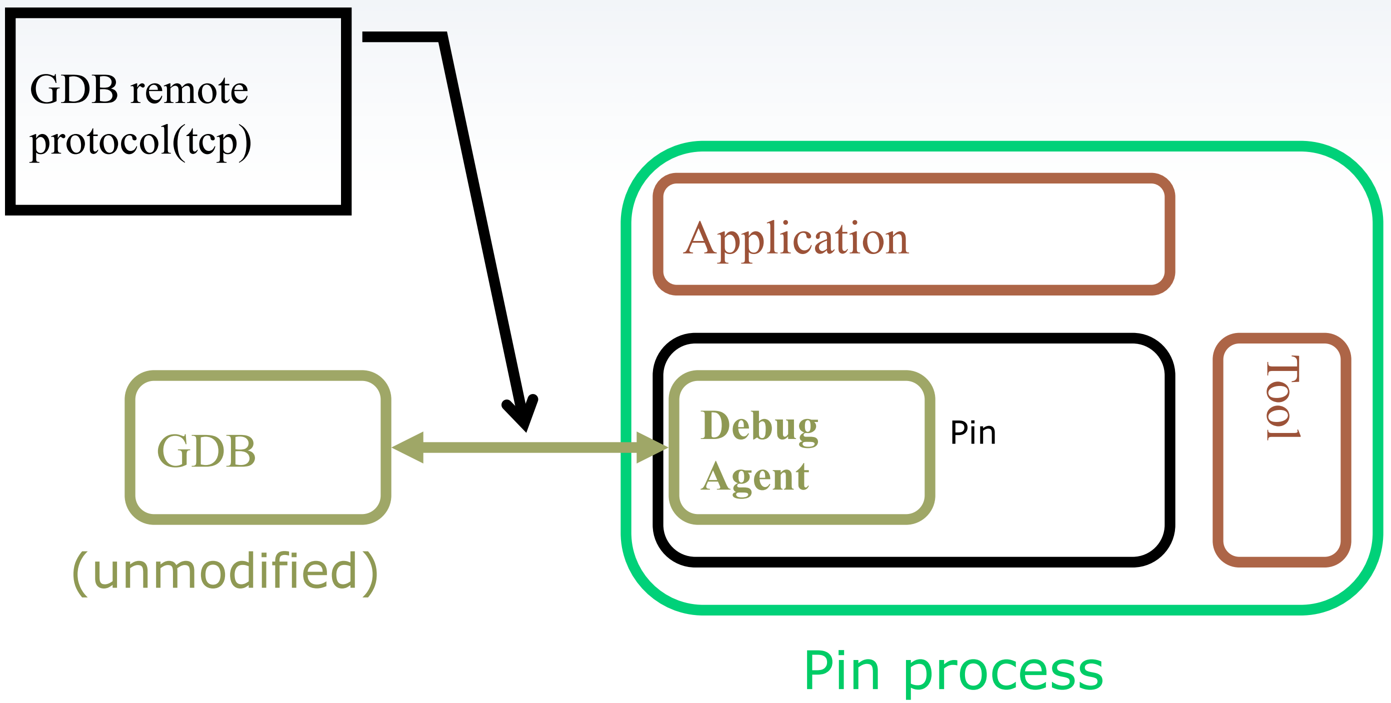
TRANSPARENT DEBUGGING & EXTENDING THE DEBUGGER

TRANSPARENT DEBUGGING

» Transparent debugging
  - "-appdebug" on Linux
  - Use vsdbg.bat on Windows

» Vsdbg actually instruments MSVS to get all required functionality

PIN DEBUGGER INTERFACE



EXTENDING THE DEBUGGER

- PIN_AddDebugInterpreter
- PIN_RemoveDebugInterpreter
- PIN_ApplicationBreakpoint
- PIN_SetDebugMode
- PIN_GetDebugStatus
- PIN_GetDebugConnectionInfo
- PIN_GetDebuggerType
- PIN_WaitForDebuggerToConnect

The real deal

SECURITY PINTOOLS

MORE TAINT ANALYSIS

- What can be tainted?
  - Memory
  - Register
- Can the flags register be tainted?
- Can the PC be tainted?

MORE TAINT ANALYSIS

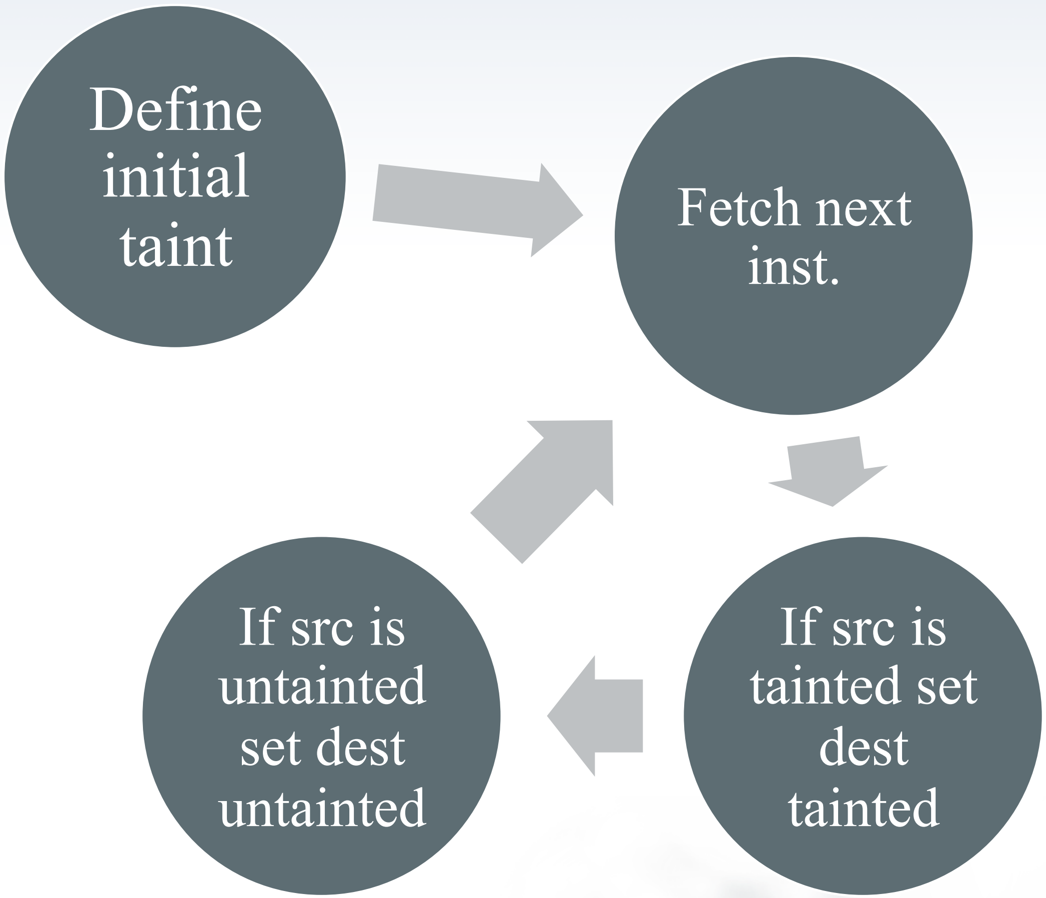
» For each instruction
  - Identify source and destination operands
    - Explicit, Implicit
  - If SRC is tainted then set DEST tainted
  - If SRC isn't tainted then set DEST not tainted

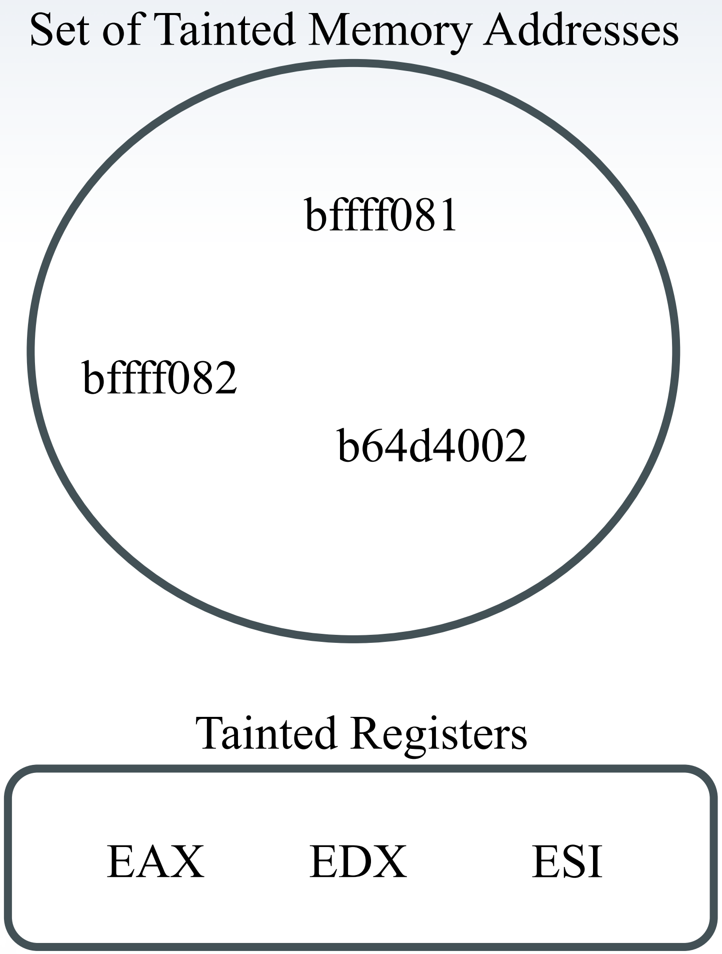
» Sounds simple, right?

MORE TAINT ANALYSIS

- Implicit operands
- Partial register taint
- Math instructions
- Logical instructions
- Exchange instructions

A SIMPLE TAINT ANALYZER





```
#include "pin.H"
#include <iostream>
#include <fstream>
#include <set>
#include <string.h>
#include "xed-iclass-enum.h"

set<ADDRINT> TaintedAddrs;   // tainted memory addresses
bool TaintedRegs[REG_LAST];  // tainted registers
std::ofstream out;           // output file

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE,  "pintool",
    "o", "taint.out", "specify file name for the output file");

/*!
 *  Print out help message.
 */
INT32 Usage()
{
   cerr << "This tool follows the taint defined by the first argument to " << endl <<
   "the instrumented program command line and outputs details to a file" << endl ;

    cerr << KNOB_BASE::StringKnobSummary() << endl;

    return -1;
}
```

```
VOID DumpTaint()  {
   out <<  "====================================" << endl;
   out <<  "Tainted Memory: " << endl;
   set<ADDRINT>::iterator it;
   for ( it=TaintedAddrs.begin() ; it != TaintedAddrs.end(); it++ )
   {
      out << " " << *it;
   }
   out << endl << "***" << endl << "Tainted Regs:" << endl;

   for (int i=0; i < REG_LAST; i++)   {
      if (TaintedRegs[i])   {
           out << REG_StringShort((REG)i);
      }   }
   out <<  "====================================" << endl;
}
```

```
// This function marks the contents of argv[1] as tainted
VOID MainAddTaint(unsigned int argc, char *argv[])  {
           if (argc != 2)  return;

           int n = strlen(argv[1]);
           ADDRINT taint = (ADDRINT)argv[1];

           for (int i = 0; i < n; i++) TaintedAddrs.insert(taint + i);

           DumpTaint();
}
```

```
// This function represents the case of a register copied to memory
void RegTaintMem(ADDRINT reg_r, ADDRINT mem_w)  {
   out << REG_StringShort((REG)reg_r) << " --> " << mem_w << endl;

   if (TaintedRegs[reg_r])   {
      TaintedAddrs.insert(mem_w);
   }
   else //reg not tainted --> mem not tainted
   {
       if (TaintedAddrs.count(mem_w))  { // if mem is already not tainted nothing to do
           TaintedAddrs.erase(TaintedAddrs.find(mem_w));
       }
   }
}
```

```
// this function represents the case of a memory copied to register
void MemTaintReg(ADDRINT mem_r, ADDRINT reg_w, ADDRINT inst_addr)  {
        out << mem_r << " --> " << REG_StringShort((REG)reg_w) << endl;

        if (TaintedAddrs.count(mem_r)) //count is either 0 or 1 for set
        {
                TaintedRegs[reg_w] = true;
        }
        else //mem is clean -> reg is cleaned
        {
                TaintedRegs[reg_w] = false;
        }
}
```

```
// this function represents the case of a reg copied to another reg
void RegTaintReg(ADDRINT reg_r, ADDRINT reg_w)
{
        out << REG_StringShort((REG)reg_r) << " --> " <<
               REG_StringShort((REG)reg_w) << endl;

        TaintedRegs[reg_w] = TaintedRegs[reg_r];
}

// this function represents the case of an immediate copied to a register
void ImmedCleanReg(ADDRINT reg_w)
{
        out << "const --> " << REG_StringShort((REG)reg_w) << endl;

        TaintedRegs[reg_w] = false;
}

// this function represents the case of an immediate copied to memory
void ImmedCleanMem(ADDRINT mem_w)
{
        out << "const --> " << mem_w << endl;

        if (TaintedAddrs.count(mem_w)) //if mem is not tainted nothing to do
        {
                TaintedAddrs.erase(TaintedAddrs.find(mem_w));
        }
}
```

HELPERS

```
// True if the instruction has an immediate operand
// meant to be called only from instrumentation routines
bool INS_has_immed(INS ins);

// returns the full name of the first register operand written
REG INS_get_write_reg(INS ins);

// returns the full name of the first register operand read
REG INS_get_read_reg(INS ins)
```

```
/*!
 * This function checks for each instruction if it does a mov that can potentially
 * transfer taint and if true adds the approriate analysis routine to check
 * and propogate taint at run-time if needed
 * This function is called every time a new trace is encountered.
 */
VOID Trace(TRACE trace, VOID *v)  {
  for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))  {
    for (INS ins = BBL_InsHead(bbl); INS_Valid(ins); ins = INS_Next(ins))   {
      if ( (INS_Opcode(ins) >= XED_ICLASS_MOV) &&
           (INS_Opcode(ins) <= XED_ICLASS_MOVZX) )   {
        if (INS_has_immed(ins))   {
          if (INS_IsMemoryWrite(ins))  {  //immed -> mem
              INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)ImmedCleanMem,
                                              IARG_MEMORYOP_EA, 0,
                                              IARG_END);
          }
          else    //immed -> reg
          {
              REG insreg = INS_get_write_reg(ins);
              INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)ImmedCleanReg,
                                              IARG_ADDRINT, (ADDRINT)insreg,
                                              IARG_END);
           }
        } // end of if INS has immed
        else if (INS_IsMemoryRead(ins)) //mem -> reg
```

```
    else if (INS_IsMemoryRead(ins))  { //mem -> reg
      //in this case we call MemTaintReg to copy the taint if relevant
                REG insreg = INS_get_write_reg(ins);
                INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)MemTaintReg,
                        IARG_MEMORYOP_EA, 0,
                        IARG_ADDRINT, (ADDRINT)insreg, IARG_INST_PTR,
                        IARG_END);
      }
      else if (INS_IsMemoryWrite(ins)) { //reg -> mem
      //in this case we call RegTaintMem to copy the taint if relevant
                REG insreg = INS_get_read_reg(ins);
                INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)RegTaintMem,
                        IARG_ADDRINT, (ADDRINT)insreg,
                        IARG_MEMORYOP_EA, 0,
                        IARG_END);
      }
      else if (INS_RegR(ins, 0) != REG_INVALID())   { //reg -> reg
      //in this case we call RegTaintReg
                REG Rreg = INS_get_read_reg(ins);
                REG Wreg = INS_get_write_reg(ins);
                INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)RegTaintReg,
                        IARG_ADDRINT, (ADDRINT)Rreg,
                        IARG_ADDRINT, (ADDRINT)Wreg,
                        IARG_END);
      }
      else { out << "serious error?!\n" << endl; }
    } // IF opcode is a MOV
  }  // For INS
 }  // For BBL
} // VOID Trace
```

```
/*!
 * Routine instrumentation, called for every routine loaded
 * this function adds a call to MainAddTaint on the main function
 */
VOID Routine(RTN rtn, VOID *v)
{
            RTN_Open(rtn);

            if (RTN_Name(rtn) == "main") //if this is the main function
            {
               RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)MainAddTaint,
                  IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                  IARG_FUNCARG_ENTRYPOINT_VALUE, 1,
                  IARG_END);
            }

            RTN_Close(rtn);
}


/*!
 * Print out the taint analysis results.
 * This function is called when the application exits.
 */
VOID Fini(INT32 code, VOID *v)
{
            DumpTaint();
            out.close();
}
```

```
int main(int argc, char *argv[])
{
   // Initialize PIN
   PIN_InitSymbols();

   if( PIN_Init(argc,argv) )
   {
      return Usage();
   }

   // Register function to be called to instrument traces
   TRACE_AddInstrumentFunction(Trace, 0);
   RTN_AddInstrumentFunction(Routine, 0);

   // Register function to be called when the application exits
   PIN_AddFiniFunction(Fini, 0);

  // init output file
   string fileName = KnobOutputFile.Value();
   out.open(fileName.c_str());

   // Start the program, never returns
   PIN_StartProgram();

   return 0;
}
```

TAINT VISUALIZATION

- Do we need to visualize registers?
- How to visualize memory?
- Is the PC important?

RETURN ADDRESS PROTECTION

- Detecting return address overwrites for functions in a certain binary
- Before function: save the expected return address
- After function: check that the return address was not modified

```
#include <stdio.h>
#include "pin.H"
#include <stack>

typedef struct
{
   ADDRINT address;
   ADDRINT value;
} pAddr;

stack<pAddr> protect; //addresses to protect

FILE * logfile; //log file

// called at end of process
VOID Fini(INT32 code, VOID *v)
{
   fclose(logfile);
}

// Save address to protect on entry to function
VOID RtnEntry(ADDRINT esp, ADDRINT addr)
{
   pAddr tmp;
   tmp.address = esp;
   tmp.value = *((ADDRINT *)esp);
   protect.push(tmp);
}
```

```
// check if return address was overwritten
VOID RtnExit(ADDRINT esp, ADDRINT addr) {
   pAddr orig = protect.top();
   ADDRINT cur_val = (*((ADDRINT *)orig.address));
   if (orig.value != cur_val)   {
      fprintf(logfile, "Overwrite at: %x old value: %x, new value: %x\n",
                        orig.address, orig.value, cur_val );
   }
   protect.pop();
}

//Called for every RTN, add calls to RtnEntry and RtnExit
VOID Routine(RTN rtn, VOID *v)   {
   RTN_Open(rtn);
   SEC sec = RTN_Sec(rtn);
   IMG img = SEC_Img(sec);

   if ( IMG_IsMainExecutable(img) && (SEC_Name(sec) == ".text") )
   {
      RTN_InsertCall(rtn, IPOINT_BEFORE,(AFUNPTR)RtnEntry, IARG_REG_VALUE,
                              REG_ESP, IARG_INST_PTR, IARG_END);
      RTN_InsertCall(rtn, IPOINT_AFTER ,(AFUNPTR)RtnExit , IARG_REG_VALUE,
                              REG_ESP, IARG_INST_PTR, IARG_END);
   }
   RTN_Close(rtn);
}
```

```
// Help message
INT32 Usage()
{
    PIN_ERROR( "This Pintool logs function return addresses in main module and reports modifications\n"
              + KNOB_BASE::StringKnobSummary() + "\n");
    return -1;
}


// Tool main function - initialize and set instrumentation callbacks
int main(int argc, char *argv[])
{
    // initialize Pin + symbol processing
    PIN_InitSymbols();
    if (PIN_Init(argc, argv)) return Usage();

    // open logfile
    logfile = fopen("protection.out", "w");

    // set callbacks
    RTN_AddInstrumentFunction(Routine, 0);
    PIN_AddFiniFunction(Fini, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}
```

AUTOMATED EXPLOITATION

- This program is the bastard son of the previous two examples
- It relies on the ability to find the source of the taint to connect the taint to the input
- This PinTool creates a log we can use to exploit the program

```
// This functions marks the contents of argv[1] as tainted
VOID MainAddTaint(unsigned int argc, char *argv[])
{
   if (argc != 2)
   {
      return;
   }

   int n = strlen(argv[1]);
   ADDRINT taint = (ADDRINT)argv[1];
   for (int i = 0; i < n; i++)
   {
      TaintedAddrs[taint + i] = i+1;
   }
}

// This function represents the case of a register copied to memory
void RegTaintMem(ADDRINT reg_r, ADDRINT mem_w)
{
   if (TaintedRegs[reg_r])
   {
      TaintedAddrs[mem_w] = TaintedRegs[reg_r];
   }
   else //reg not tainted --> mem not tainted
   {
      if (TaintedAddrs.count(mem_w)) // if mem is already not tainted nothing to do
      {
         TaintedAddrs.erase(mem_w);
      }
   }
}
```

```
VOID RtnExit(ADDRINT esp, ADDRINT addr)
{

/*
 *   SNIPPED...
 */

ADDRINT cur_val = (*((ADDRINT *)orig.address));
if (orig.value != cur_val)
   {
      out << "Overwrite at: " << orig.address << " old value: " << orig.value
                                  << " new value: " << cur_val << endl;
      for (int i=0; i<4; i++)
      {
           out << "Source of taint at: " << (orig.address + i) << " is: "
                      << TaintedAddrs[orig.address+i] << endl;
      }

      out << "Dumping taint" << endl;
      DumpTaint();
   }

   protect.pop();
}
```

FROM LOG TO EXPLOIT

- Simple processing of the log file gives us the following:
  - The indices in the input string of the values that overwrote the return pointer
  - All memory addresses that are tainted at the time of use
- With a bit of effort we can find a way to encode wisely and take advantage of all tainted memory
  - But for sake of example I use the biggest consecutive buffer available
- We can mark areas we don’t want to be modified like protocol headers

Please Remember to Complete Your Feedback Form

What about process context manipulations?

BONUS 1: SIGNALS, EXCEPTIONS AND SYSTEM CALLS

CONTEXTS

- Physical vs. Pin
- Constant vs. modifiable
  - IARG_CONST_CONTEXT
  - IARG_CONTEXT

CONTEXT* OBJECTS

» CONTEXT* can NOT be dereferenced. It is a handle to be passed to Pin API functions

» CONTEXT* is passed by default to a number of Pin Callback functions: e.g.

- PIN_AddThreadStartFunction
- PIN_DefineTraceBuffer
- PIN_AddContextChangeFunction

CONTEXT* OBJECTS

» Pin API has getters and setters for:
  - GP registers within the context
  - FP registers within the context

» Contexts can be passed to analysis routines:
  - IARG_(CONST)_CONTEXT
  - The analysis routine will NOT be inlined
  - The passing of the CONTEXT* is time consuming
  - Passing IARG_CONST_CONTEXT is ~4X faster than IARG_CONTEXT

CONTEXT* ...

» Changes made to the contents of a CONTEXT*
  - IARG_CONTEXT
    - Changes made will be visible in subsequent Pin API calls made from within the nesting of the analysis function
    - Changes made will NOT be visible in the application context after return from the analysis function
  - Passed to Pin Callbacks
    - Changes made will be visible in both of above

SIGNALS AND EXCEPTIONS

- Exceptions
  - Pin / tool exceptions
  - Monitoring application exceptions
  - Injecting exceptions to the application
- Signals
  - Application
  - Tool
  - Injection

EXCEPTIONS

» Catch Exceptions that occur in Pin Tool code

- Global exception handler
  - PIN_AddInternalExceptionHandler
- Guard code section with exception handler
  - PIN_TryStart
  - PIN_TryEnd

EXCEPTIONS

» Inject exceptions to the application
  - Set the exception address
    - PIN_SetExceptionAddress
  - Raise the exception in the application
    - PIN_RaiseException

MONITORING APPLICATION EXCEPTIONS AND SIGNALS

» PIN_AddContextChangeFunction

- Can monitor and change that application state at application exceptions

SIGNALS

» Establish an interceptor function for signals delivered to the application
  - Tools should never call sigaction() directly to handle signals.
  - Function is called whenever the application receives the requested signal, regardless of whether the application has a handler for that signal.
  - Function can then decide whether the signal should be forwarded to the application

SIGNALS

» A tool can take over ownership of a signal in order to:
  - Use the signal as an asynchronous communication mechanism to the outside world.
  - "squash" certain signals that the application generates.
    - A tool that forces speculative execution in the application may want to intercept and squash exceptions generated in the speculative code.

» A tool can set only one "intercept" handler for a particular signal, so a new handler overwrites any previous handler for the same signal. To disable a handler, pass a NULL function pointer.

SIGNALS

- PIN_InterceptSignal
  - Allows tool to intercept a signal
  - If the handler returns FALSE the signal is not passed to the application
- PIN_UnblockSignal
  - Prevents application from blocking a signal

SYSTEM CALLS

» Instrumenting system calls is easy, until now we had:

- INS_IsSyscall

» But we also have dedicated APIs:

- PIN_AddSyscallEntryFunction
- PIN_AddSyscallExitFunction

SYSTEM CALLS

» Getters and setters:
  - PIN_GetSyscallNumber
  - PIN_GetSyscallArgument
  - PIN_SetSyscallNumber
  - PIN_SetSyscallArgument
  - PIN_GetSyscallReturn
  - PIN_GetSyscallErrno

SECTION SUMMARY

- CONTEXT objects give you access to the entire context the application sees but at a big performance cost
- Pin provides a way to register callbacks for exceptions, signals and context switches in general
- Pin provides an way to register callbacks on system calls

Because we live in a parallel universe

BONUS 2: PROCESSES AND THREADS

MULTI THREADING

» Application threads execute JITted code including instrumentation code (inlined and not inlined)
  - Pin does not introduce serialization
  - Instrumentation code can use Pin and/or OS synchronization constructs
  - The JITting itself (VM) is serialized

» Pin provides APIs for thread local storage.

» Pin callbacks are serialized

INSTRUCTION COUNTING: TAKE 3 - MT

```
#include "pin.H"
INT32 numThreads = 0;
const INT32 MaxNumThreads = 10000;
struct THREAD_DATA
{
    UINT64 _count;
    UINT8 _pad[56];  // guess why?  } icount[MaxNumThreads];
// Analysis routine
VOID PIN_FAST_ANALYSIS_CALL docount(ADDRINT c, THREADID tid) { icount[tid]._count += c; }
// Pin Callback
VOID ThreadStart(THREADID threadid, CONTEXT *ctxt, INT32 flags, VOID *v) {numThreads
  ++;}

VOID Trace(TRACE trace, VOID *v) { // Jitting time routine: Pin Callback
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
        BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)docount, IARG_FAST_ANALYSIS_CALL,
                       IARG_UINT32, BBL_NumIns(bbl), IARG_THREAD_ID, IARG_END);    }

VOID Fini(INT32 code, VOID *v){// Pin Callback
    for (INT32 t=0; t<numThreads; t++)
        printf ("Count[of thread#%d]= %d\n",t,icount[t]._count);   }

int main(int argc, char * argv[])    {
    PIN_Init(argc, argv);
    for (INT32 t=0; t<MaxNumThreads; t++) {icount[t]._count = 0;}
    PIN_AddThreadStartFunction(ThreadStart, 0);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram(); return 0;    }
```

THREADING CALLBACKS

» PIN_AddThreadStartFunction

» PIN_AddThreadFiniFunction

THREADING API

- PIN_ThreadId
- PIN_ThreadUid
- PIN_GetParentTid
- PIN_WaitForThreadTermination
- PIN_CreateThreadDataKey
- PIN_DeleteThreadDataKey
- PIN_Yield
- PIN_ExitThread
- PIN_SetThreadData
- PIN_GetThreadData
- PIN_Sleep

TOOL THREADS

» You can create tool threads
  - Handle buffers
  - Parallelize data processing

TOOL THREAD API

» PIN_SpawnInternalThread

» PIN_IsApplicationThread

» PIN_ExitThread

INSTRUMENTING A PROCESS TREE

» Fork

» Execv

» Windows

PROCESS CALLBACKS

- PIN_AddFollowChildProcessFunction
- PIN_AddForkFunction
- PIN_AddFiniFunction
- PIN_AddApplicationStartFunction

PRCESS API

- PIN_IsProcessExiting
- PIN_GetPid
- PIN_ExitProcess
- PIN_ExitApplication

SECTION SUMMARY

- » Pin has various APIs and callbacks to handle multi threading
- » Pin supports instrumenting entire process trees using "–follow_execv"
- » You can get callbacks on fork and execv in Linux

Because most of us don’t write perfect code

BONUS 3: PRACTICAL ISSUES WITH TOOLS

LOGGING

» Macros:
  - LOG
  - PIN_ERROR

» Command line options:
  - -unique_logfile

» Logfiles:
  - pin.log
  - pintool.log

OUTPUT FROM YOUR TOOL

» It is very important to consider how to send output from your tool

» Writing to stdout / stderr might not always work even in non-GUI applications

» Best method is to write to a file

- Use C++ functions carefully

DEBUGGING YOUR PINTOOLS

» Windows vs. Linux

» Make options:

- SOTOOL=0
  - Make tool build as an executable instead of DLL
- DEBUG=1
  - Enable debugging

» Execute Pin with -pause_tool option

- Pin will pause and print how to attach a debugger

Where to look for information?

BIBLIOGRAPHY AND REFERENCES

BIBLIOGRAPHY & REFERENCES

» This is a list some relevant material. No specific logical order was applied to the list. The list is in no way complete nor aims to be.

» Dino Dai Zvoi publications on DBT and security

» Shellcode analysis using DBI / Daniel Radu & Bruce Dang (Caro 2011)

» Black Box Auditing Adobe Shockwave / Black Box Auditing Adobe Shockwave

» Making Software Dumber / Tavis Ormandy

BIBLIOGRAPHY & REFERENCES

» Taint Analysis / Edgar Barbosa

» ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks / Lucas Davi, Ahmad-Reza Sadeghi, Marcel Winandy

» Hybrid Analysis of Executables to Detect Security Vulnerabilities

» Tripux: Reverse-Engineering Of Malware Packers For Dummies / Joan Calvet

» Tripux @ Google code

» devilheart: Analysis of the spread of taint of MS-Word

BIBLIOGRAPHY & REFERENCES

» PIN home page

» PIN mailing list @Yahoo (PinHeads)

» Pin online documentation

» DynamoRIO mailing list

» DynamoRIO homepage

» Valgrind homepage

» ERESI project

» Secure Execution Via Program Shepherding / Vladimir Kiriansky, Derek Bruening, Saman Amarasinghe

BIBLIOGRAPHY & REFERENCES

» Pincov – a code coverage module for PIN

» P-debugger – a multi thread debugging tool based on PIN

» Tartetatintools - a bunch of experimental pintools for malware analysis

» PrivacyScope

» TaintDroid

» Dynamic Binary Instrumentation for Deobfuscation and Unpacking / Jean-Yves Marion, Daniel Reynaud

BIBLIOGRAPHY & REFERENCES

» Automated Identication of Cryptographic Primitives in Binary Programs / Felix Grobert, Carsten Willems and Thorsten Holz

» Covert Debugging: Circumventing Software Armoring Techniques / Danny Quist, Valsmith

» Using feedback to improve black box fuzz testing of SAT solvers

» All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution / Edward J. Schwartz, Thanassis Avgerinos, David Brumley

» Automated SW debugging using PIN

BIBLIOGRAPHY & REFERENCES

» Determina website (no real information)

» Determina blog

» Sweeper: A Lightweight End-to-End System for Defending Against Fast Worms / James Newsome, David Brumley, et. el.

» Hunting Trojan Horses / Micha Moffie and David Kaeli

» Helios: A Fast, Portable and Transparent Instruction Tracer / Stefan Bühlmann and Endre Bangerter

» secuBT: Hacking the Hackers with User-Space Virtualization / Mathias Payer

BIBLIOGRAPHY & REFERENCES

» Understanding Swizzor's Obfuscation / Joan Calvet and Pierre-Marc Bureau

» GENESIS: A FRAMEWORK FOR ACHIEVING SOFTWARE COMPONENT DIVERSITY

» A PinTool implementing datacollider algorithm from MS

» Rootkit detection via Kernel Code Tunneling / Mihai Chiriac

» Dytan: A Generic Dynamic Taint Analysis Framework / James Clause, Wanchun Li, and Alessandro Orso