HACKING USING DYNAMIC BINARY INSTRUMENTATION

GAL DISKIN / INTEL

(@GAL_DISKIN)

HITB 2012

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+)

CREDITS

- Tevi Devor of the Pin development team for parts of his Pin tutorial that were adapted used as a base for the Pin tutorial part of this presentation
- Dmitriy "D1g1" Evdokimov (@evdokimovds) from DSecRG for reviewing the presentation and providing constructive criticism

ABOUT THIS WORKSHOP

- How does DBI work – Intro to a DBI engine (Pin)
- The InfoSec usages of DBI
- InfoSec DBI tools

WHAT IS INSTRUMENTATION

- (Binary) instrumentation is the capability to observe, monitor and modify a (binary) program behavior

in·stru·men·ta·tion *noun*
\ˌin(t)-strə-mən-ˈtā-shən, -ˌmen-\

Definition of INSTRUMENTATION

1 : the arrangement or composition of music for instruments especially for a band or orchestra

2 : the use or application of instruments (as for observation, measurement, or control)

3 : instruments for a particular purpose; *also* : a selection or arrangement of instruments

See instrumentation defined for English-language learners »

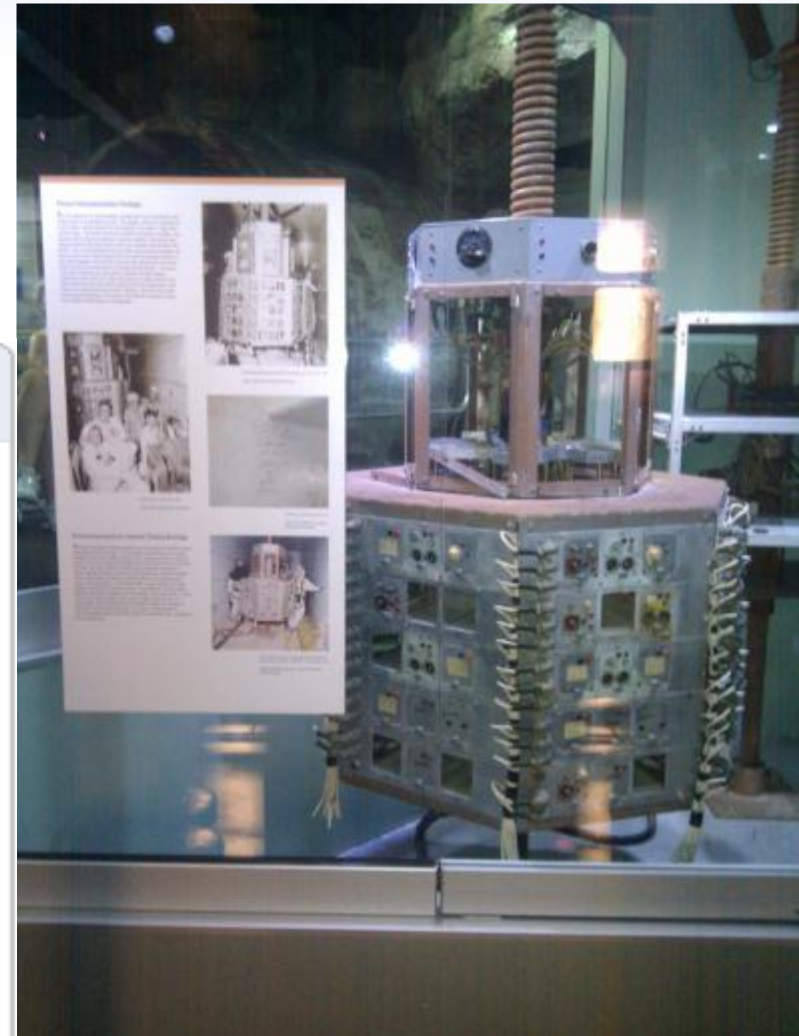
See instrumentation defined for kids »

Examples of INSTRUMENTATION

- There was a problem with the airplane's *instrumentation*.

First Known Use of INSTRUMENTATION

1845



INSTRUMENTATION TYPES

» Source / Compiler Instrumentation

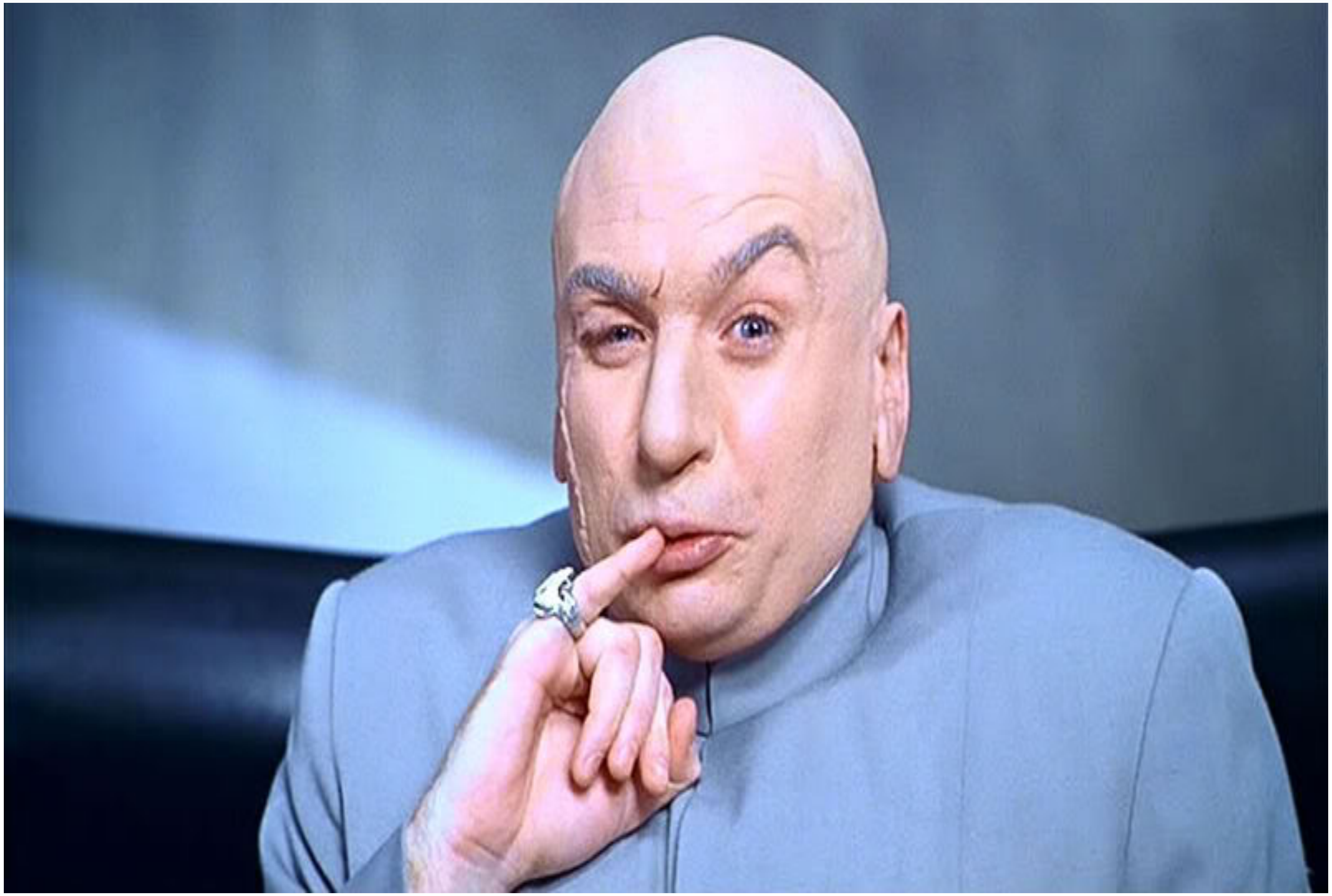
» Static Binary Instrumentation

» Dynamic Binary Instrumentation

HOW NON-SECURITY PEOPLE USE DBI

- 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.

SECURITY APPLICATIONS OF INSTRUMENTATION

- *Data flow analysis*
- *Control flow analysis*

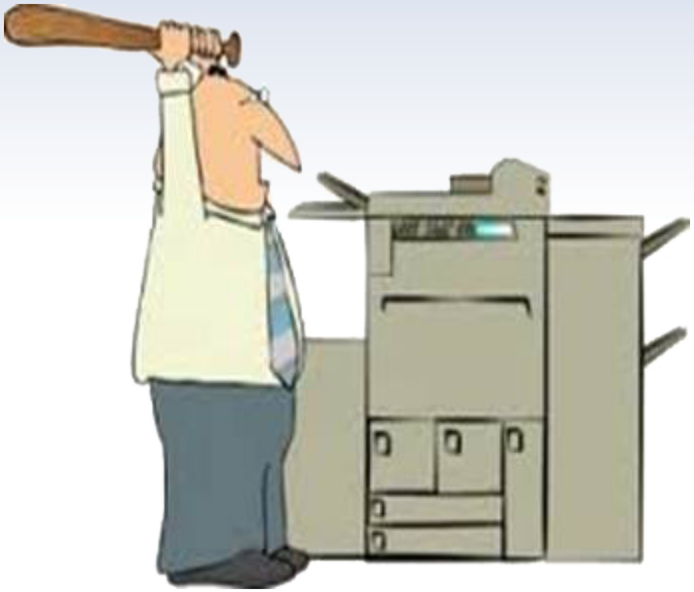
- Fuzzing
- Vulnerability detection
- Program visualization
- Taint analysis
- Reverse engineering
- Transparent debugging

- Privacy
- Vulnerability classification
- Behavior based security
- Anti-virus / Anti-malware technologies
- Forcing security practices
- Sandboxing
- Automated exploitation
- Pre-patching
- Forensics

BINARY INSTRUMENTATION ENGINES

- Pin
- DynamoRio
- Valgrind
- DynInst
- 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

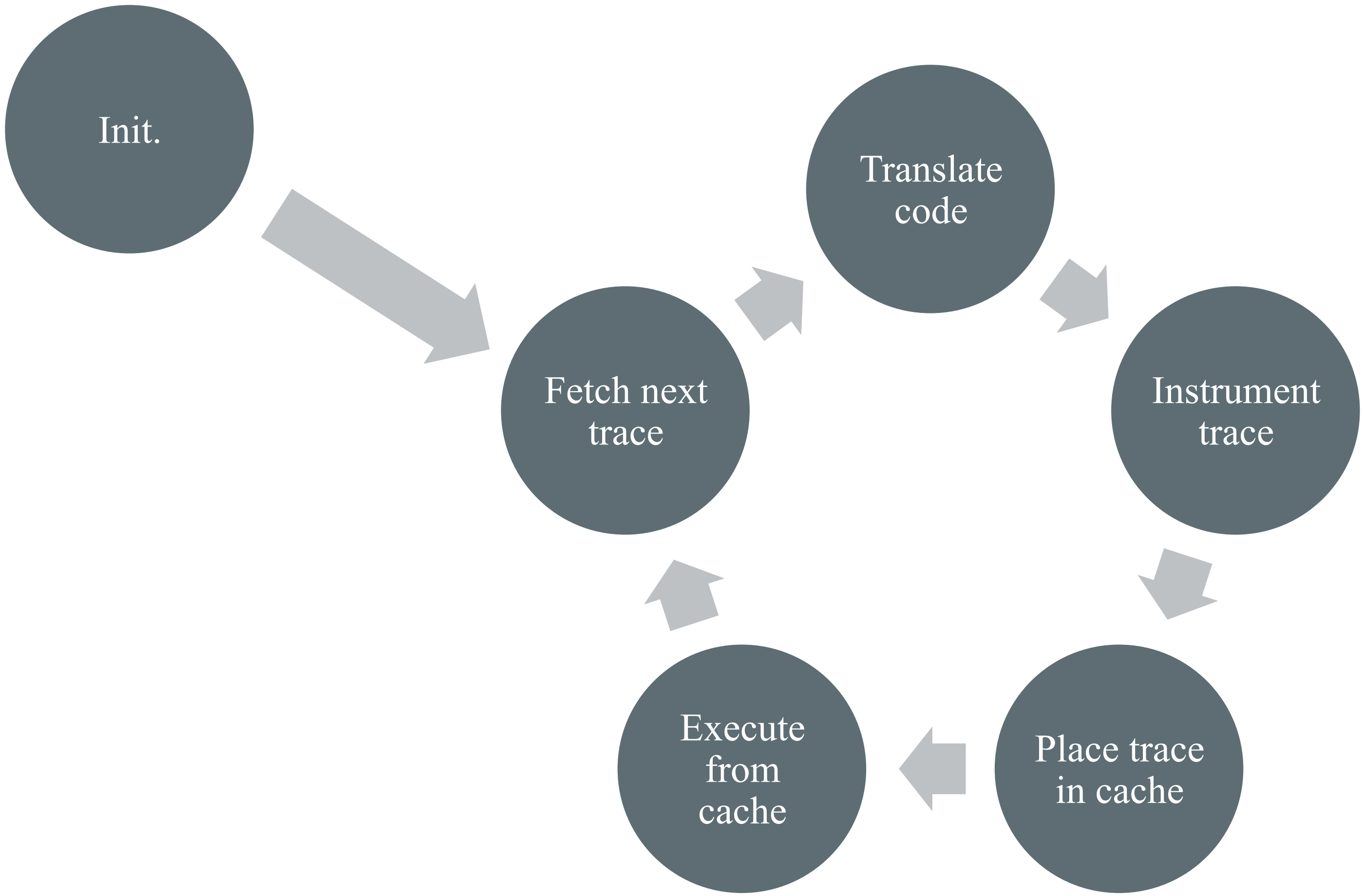
A PROGRAM'S BUILDING BLOCKS

» Instruction

» Basic Block

» Trace (sometimes called Super-block)

PIN EXECUTION



PINTOOL 101: INSTRUCTION COUNTING

```
#include "pin.h"

UINT64 icount = 0;

void docount() { icount++; }
```

Execution time routine

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

Jitting time routine

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

```
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;
}
```

```
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_options] -t pintool.dll [pintool_options] – app_name.exe [app_args]

- 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

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

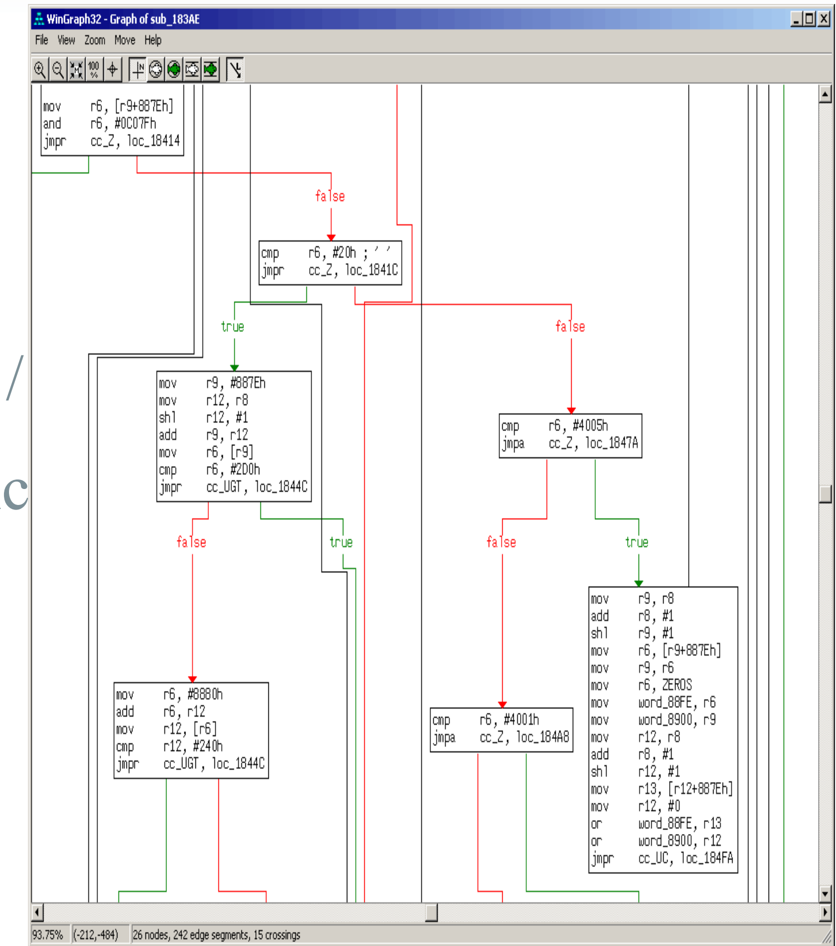
Points to per-thread spill area

```
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
```

(

REVERSING

- De-obfuscation / unpacking
- Frequency analysis
- SMC analysis
- Automated lookup for behavior /
- Differential analysis / equivalenc
- Data structure restoration



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

COVERT DEBUGGING

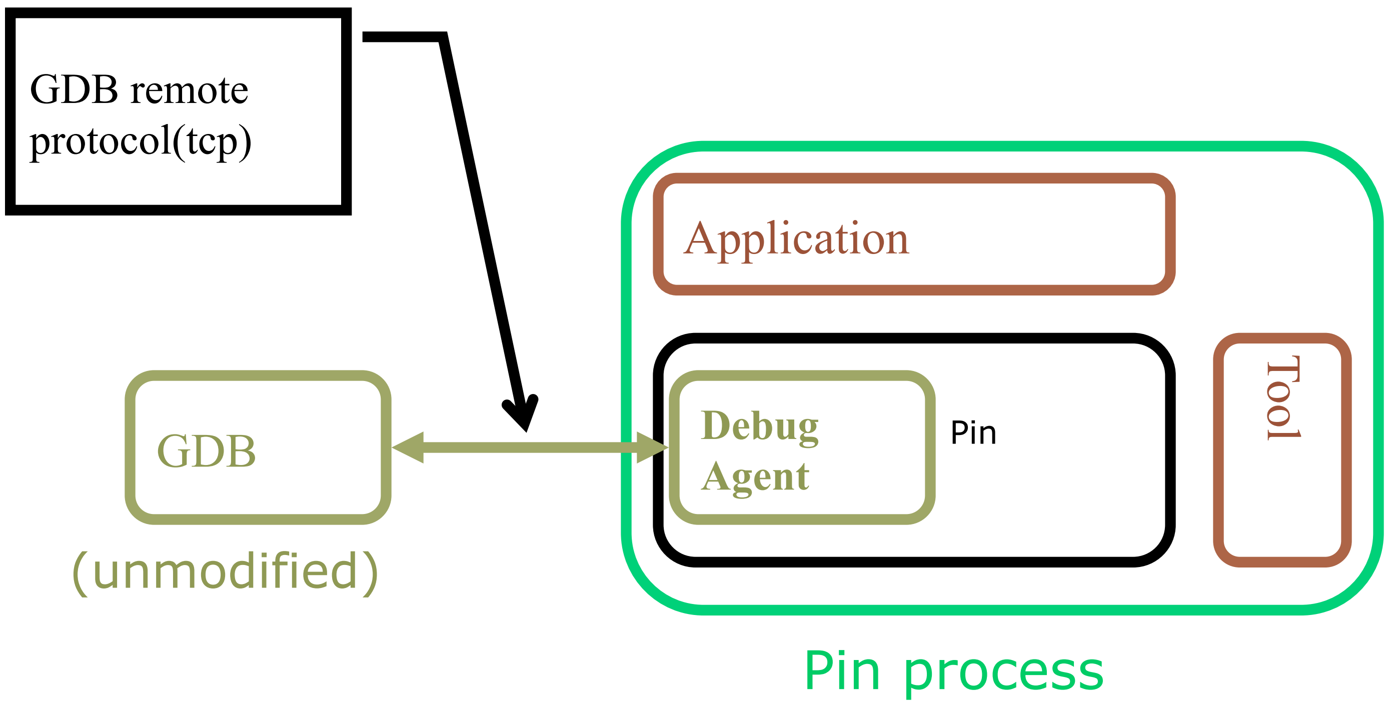
» Hiding from anti-debug techniques
» Anti-instrumentation
» Anti-anti instrumentation

TRANSPARENT DEBUGGING

- PIN's terminology for "covert deugging"
- Transparent debugging
  - "-appdebug" on Linux

- Experimental Windows support exists and might go mainline soon (look for vsdbg.bat in the Pin kit)

PIN DEBUGGER INTERFACE



COVERT DEBUGGING DEMO

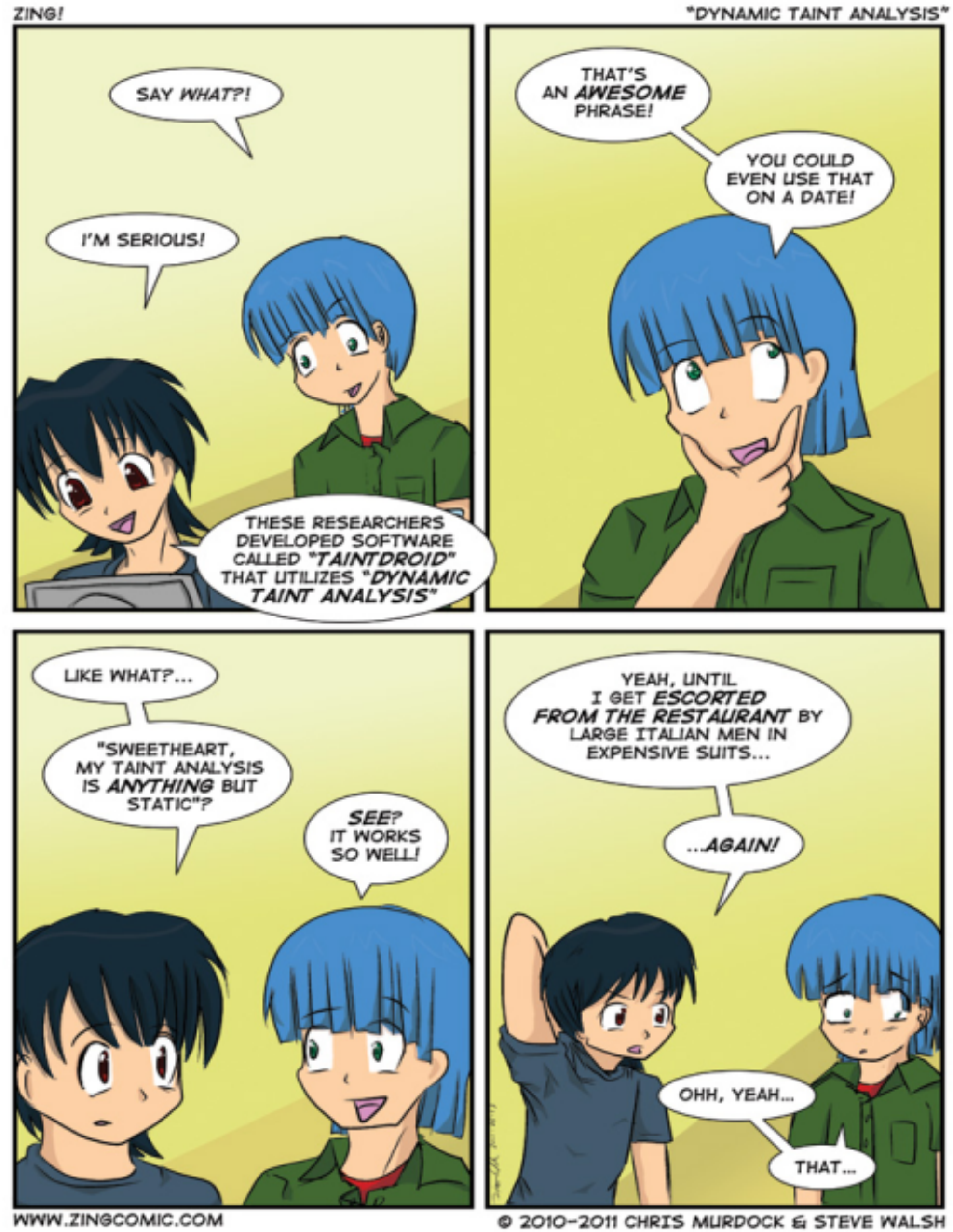
TAINT ANALYSIS

» Following tainted data flow through programms

» 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]()

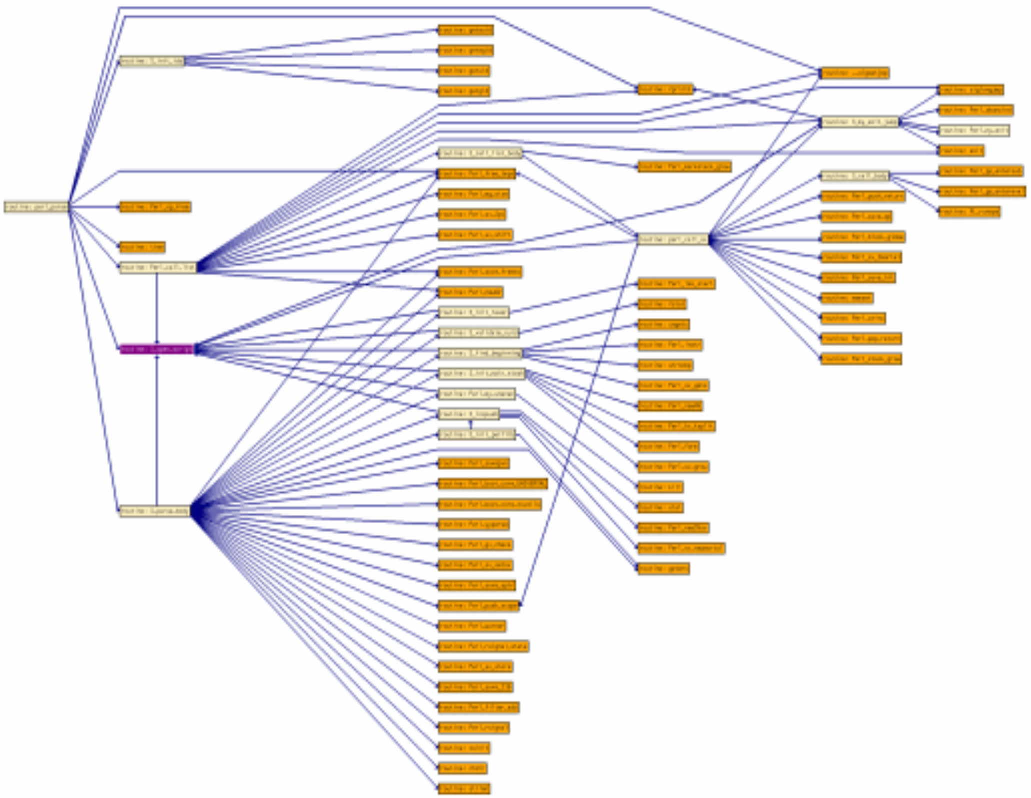
» [Flayer]()

» 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]

MORE TAINT ANALYSIS

» What can be tainted?

- Memory
- Register

» Can the flags register be tainted?

» Can the PC be tainted?

MORE TAINT ANALYSIS

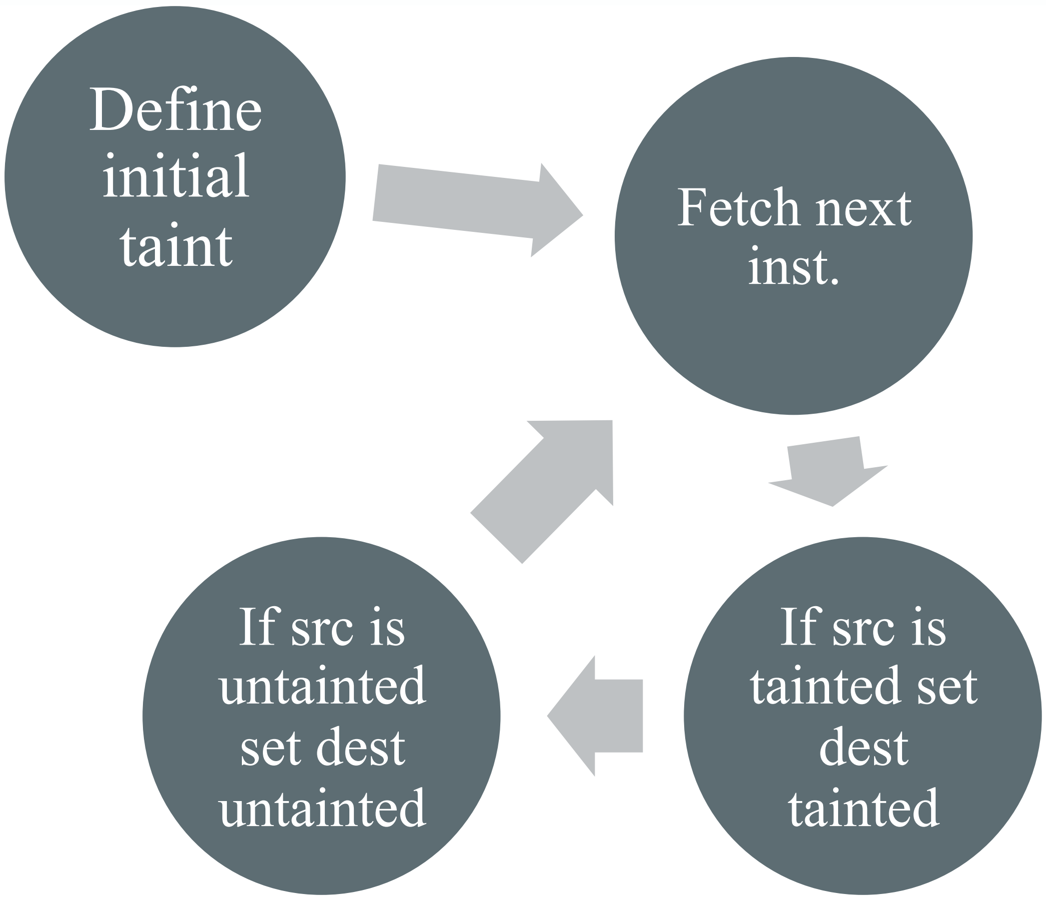
» For each instruction
  - Identify source and destination operands
    - Explicit, Implicit
  - If SRC is tainted then set DEST tainted
  - If SRC isn't tainted then set DEST not tainted

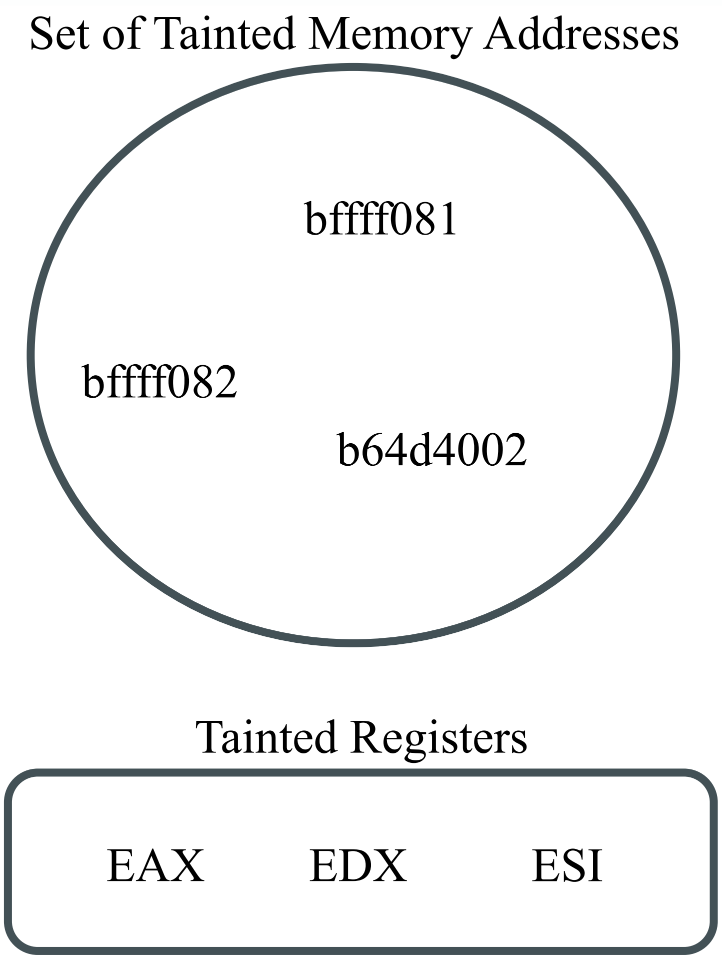
» Sounds simple, right?

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

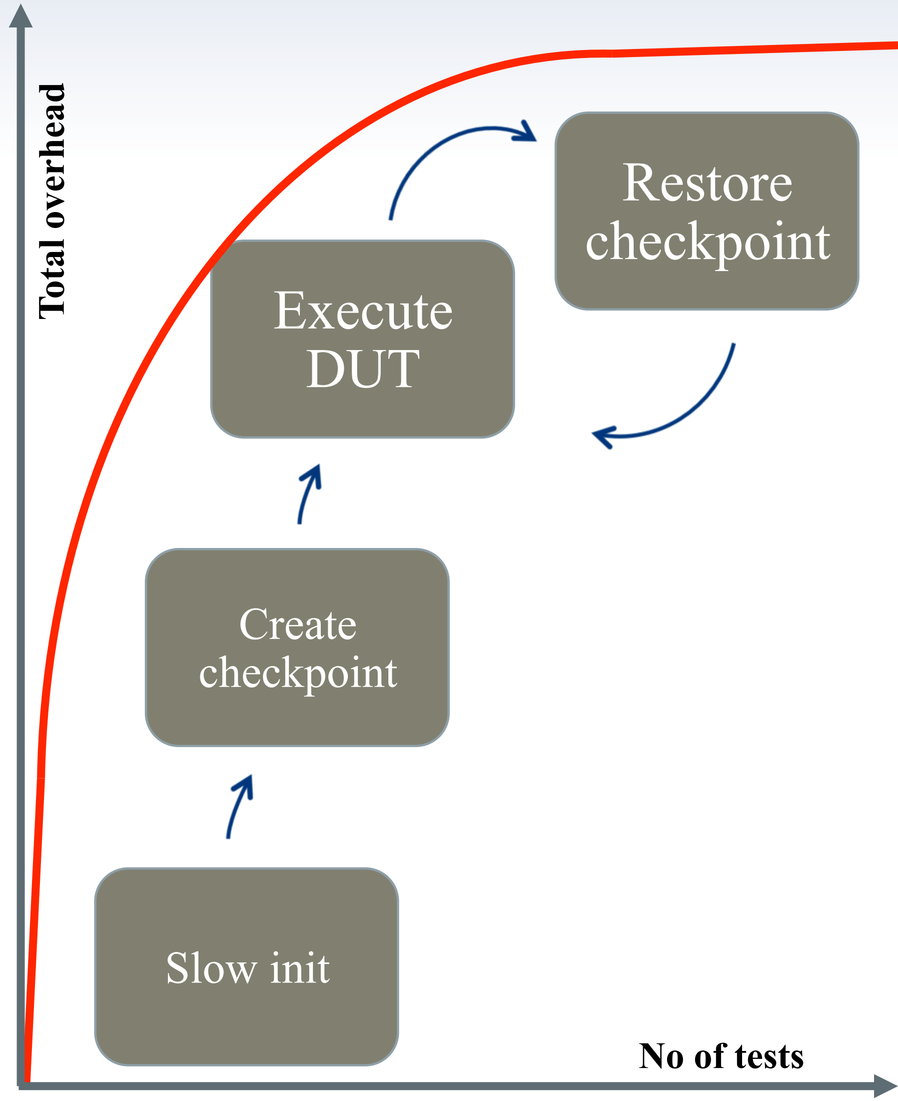
- Do we need to visualize registers?
- How to visualize memory?
- Is the PC important?

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

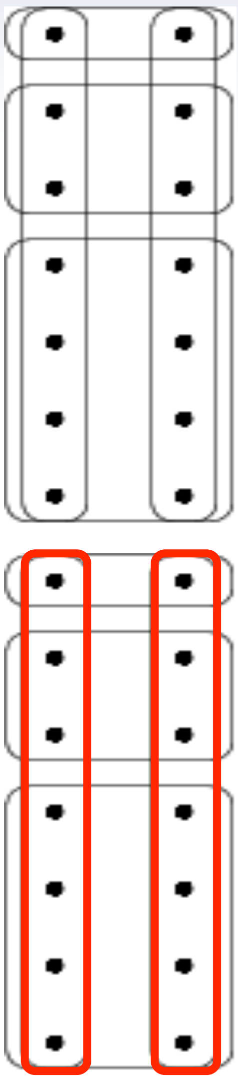
FAST FUZZING

- The main overhead of modern instrumentation comes from the first pass on the code (JIT)
- Many programs have a constant long initialization (and destruction) before what we're interested in testing
- One solution to this is checkpointing
- Over enough time:
  (init*overhead) << (init*no of tests)



CORPUS DISTILLATION

- A technique for locating "untested" code
- Corpus – the entire collection of existing inputs
- Distilled corpus – a subset of the corpus with the same code coverage
- Simple set operations or other operations like mutations allow finding new test cases from a distilled corpus that target uncovered areas



FUZZING / SECURITY TEST CASE GENERATION

» Examples:

- Tavis Ormandy @ HITB'09
- Microsoft SAGE

ADVANCED MONITORING

» Defining advanced restrictions on your program behavior and detecting violations of those

» In particular applying vulnerability detection:

- Generic:
  - Exploitable condition
  - Exploitable behavior
- Specific:
  - Illegal state or sequence of states
  - Illegal values
  - Illegal data-flow
  - Illegal control-flow

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

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
- aka “shadow stack”

```
#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 EXPLOIT DEVELOPMENT

- Known exploit techniques
- SAT/SMT

- **Want to learn more?
Look forward to my paper**

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

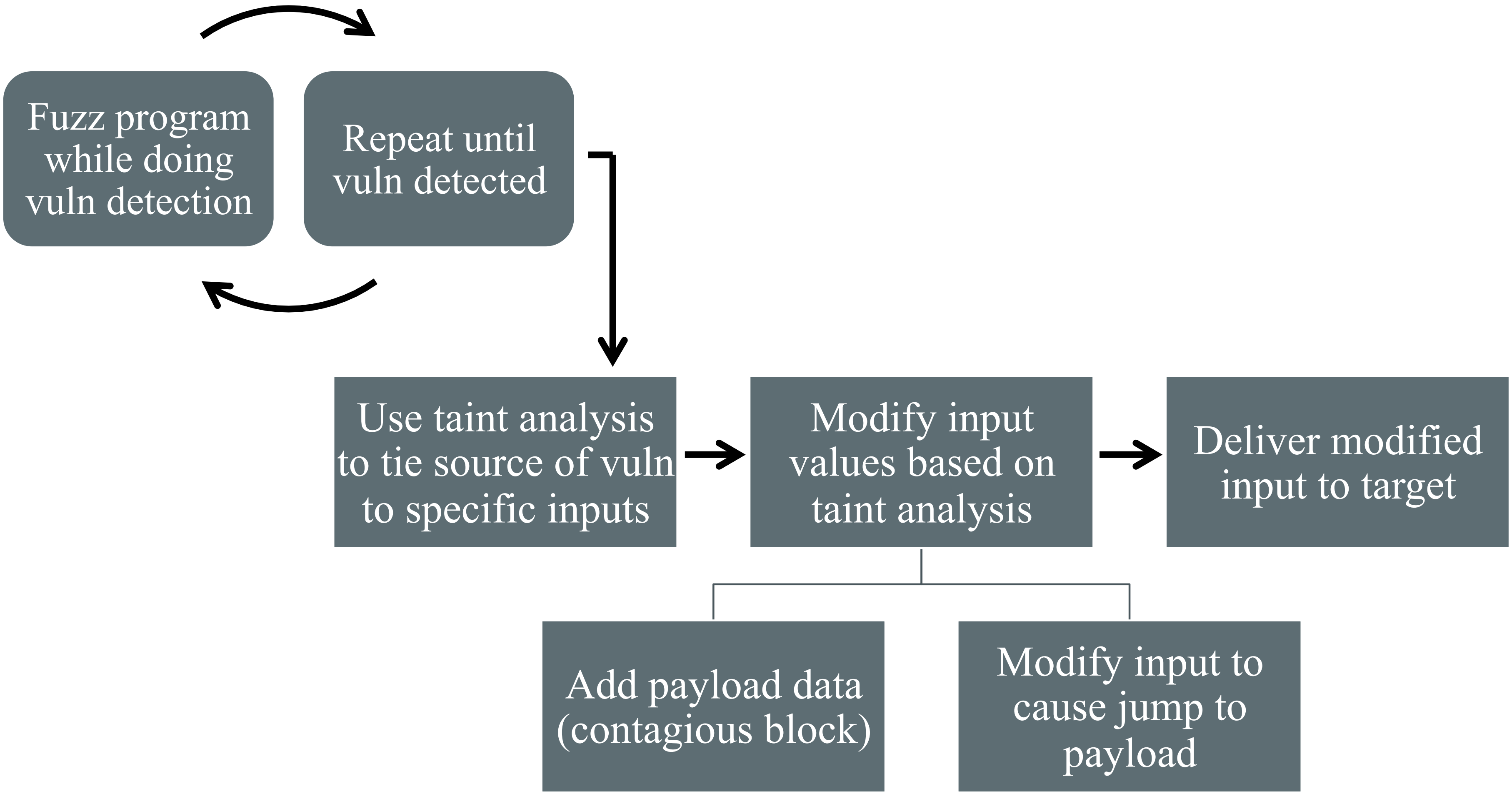
```
// This functions marks the contents of argv[1] as tainted
VOID MainAddTaint(unsigned int argc, char *argv[])
{
   if (argc != 2)
   {
      return;
   }

   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

AUTOMATIC EXPLOITATION FLOW



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

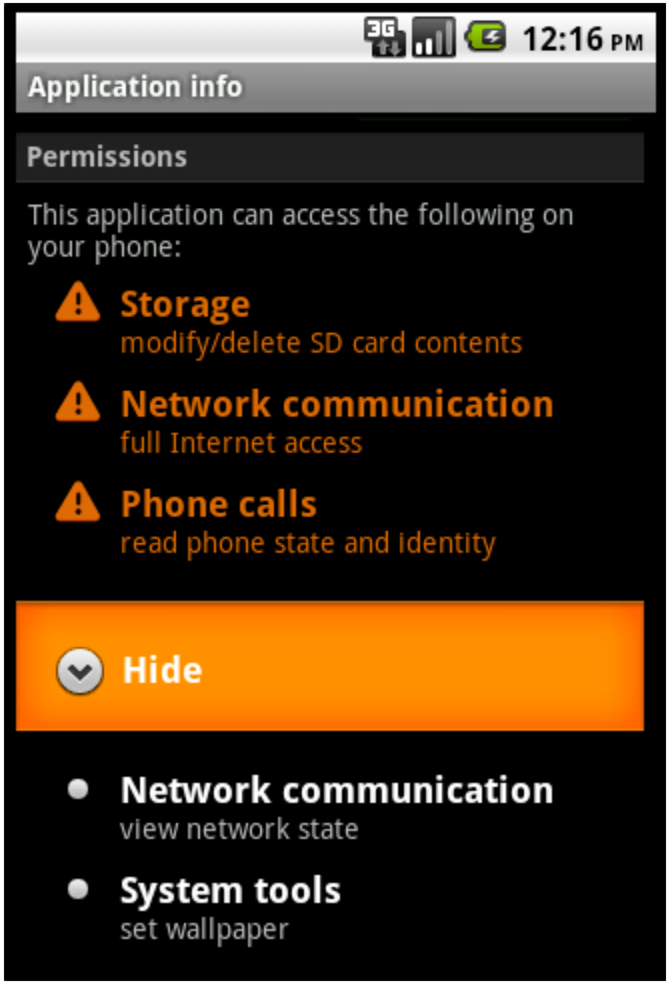
BEHAVIOR BASED SECURITY

» 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

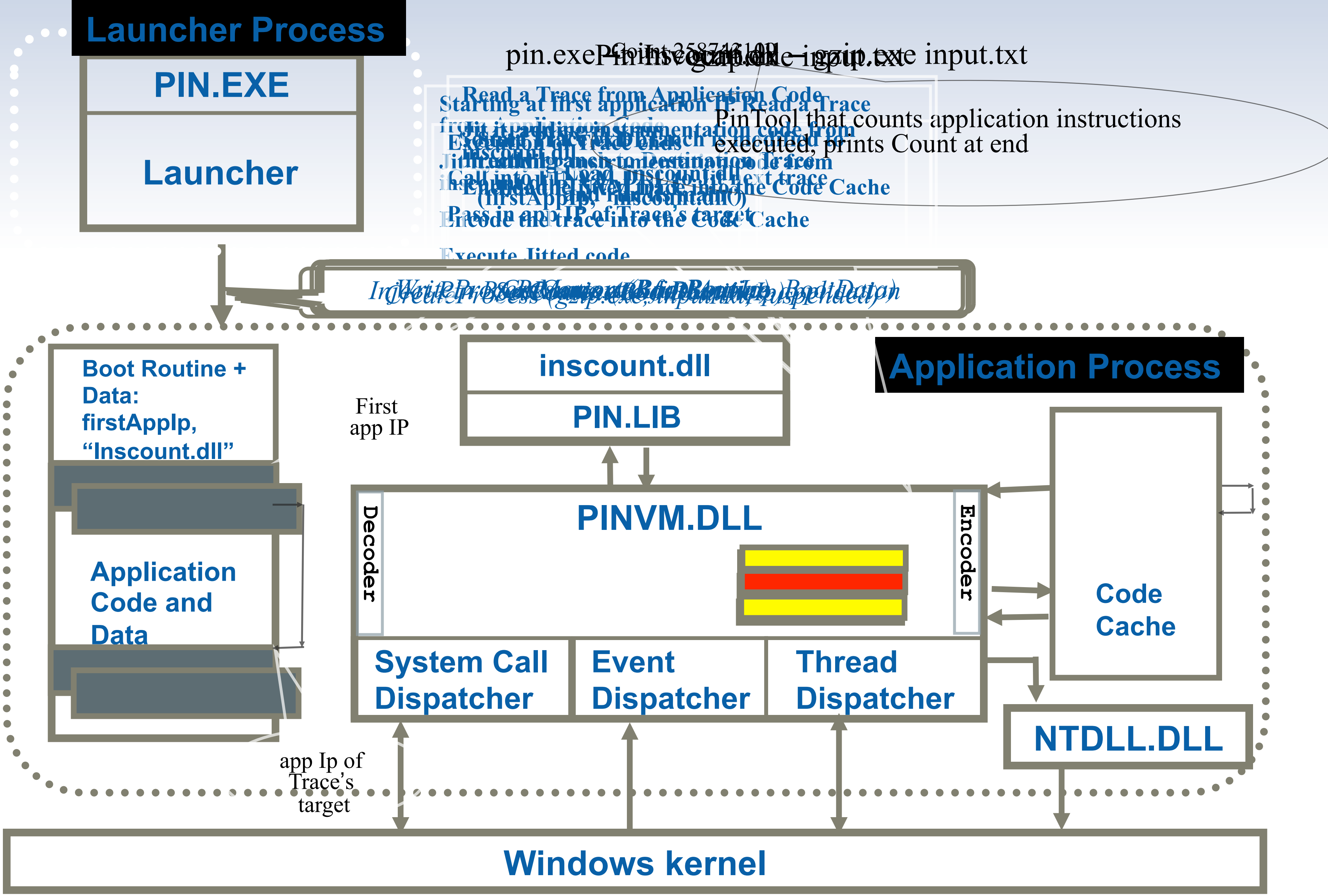
- Vulnerability classification
- Anti-virus technologies
- Forcing security practices
  - Adding stack cookies
  - Forcing ASLR
- Sandboxing
- Forensics

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

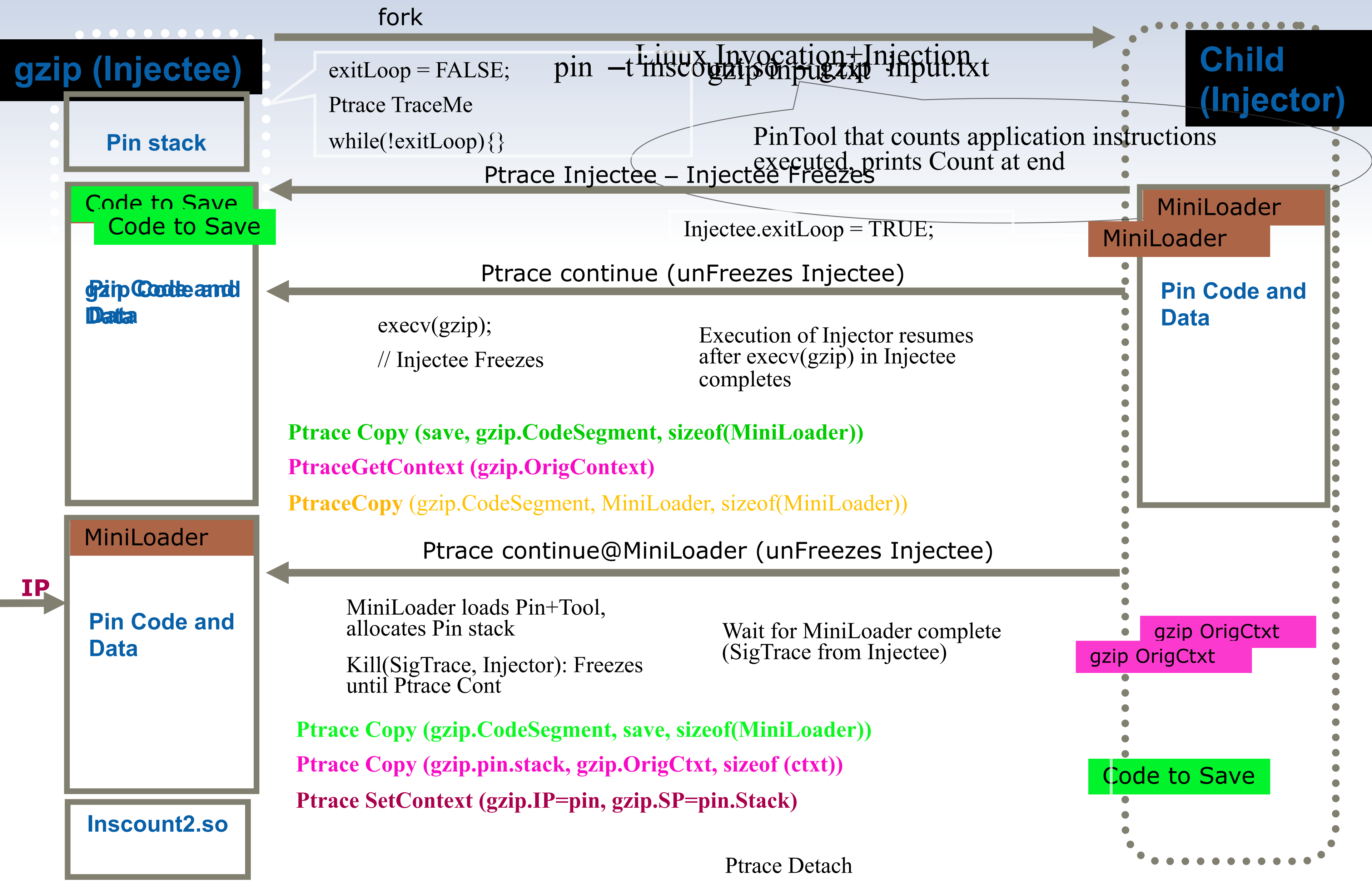
DETACHING

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


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

PIN INJECTION

- Also known as “Early Injection”
- Allows you to instrument every instruction in the process starting from the very first loader instruction


fork
gzip (Injectee)
Child (Injector)
Linux Invocation+Injection
exitLoop = FALSE;
Ptrace TraceMe
while(!exitLoop){}
pin –t inscount2.so – gzip input.txt
PinTool that counts application instructions executed, prints Count at end
Pin stack
Ptrace Injectee – Injectee Freezes
Code to Save
Code to Save
MiniLoader
MiniLoader
Injectee.exitLoop = TRUE;
gzip Code and Data
Pin Code and Data
Ptrace continue (unFreezes Injectee)
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

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