Binary instrumentation for hackers and security professionals

Gal Diskin / Intel

Special thanks to Tevi Devor from the PIN development team

Legal Disclaimer

ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

All code in this presentation is covered by the following:

Part 1

INTRODUCTION

Agenda

- Introduction
  - What is Instrumentation
  - Why should you care? Potential usages
- PIN – A binary instrumentation engine
  - About PIN
  - Understanding PIN – a case study in instruction counting
- A practical example – catching "double-free"
- Advanced stuff and closing notes
  - Probe mode and JIT mode
  - Advanced PIN capabilities
  - Alternatives
  - How to learn more
  - Call for action

Instrumentation

A technique that inserts code into a program to collect run-time information

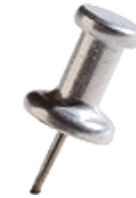
- ❑ Program analysis : performance profiling, error detection, capture & replay
- ❑ Architectural study : processor and cache simulation, trace collection

- Source-Code Instrumentation
- Static Binary Instrumentation
- **Dynamic Binary Instrumentation**
  - ❑ **Instrument code just before it runs (Just In Time – JIT)**
    - – No need to recompile or re-link
    - – Discover code at runtime
    - – Handle dynamically-generated code
    - – Attach to running processes

Why should you care?

- The previous slide didn't mention all potential usages
- Potential security related usages:
  - Sandboxing & Forcing security practices
  - Behavior-based security
  - Pre-patching
  - Reversing, unpacking & de-obfuscation
  - SW analysis, for example - Parallel studio
    - Thread checking
    - Memory checking
  - Taint-analysis
  - Anti-viruses
  - Automated vulnerability classification / analysis
  - Deterministic replay
  - …
- Do you have tool ideas? Let me know and I might help

Part 1 - Summary

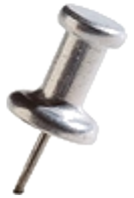


- Instrumentation – a technique to inject code into a program
- Dynamic binary instrumentation – what we will focus on today
- Instrumentation has tons of uber kwel usages for offense and defense

Part 2

PIN – A DYNAMIC BINARY INSTRUMENTATION ENGINE

What Does "Pin" Stand For?



- Three Letter Acronyms @ Intel
  - TLAs
  - 26^3 possible TLAs
    - 26^3 -1 are in use at Intel
    - Only 1 is not approved for use at Intel
    - Guess which one:

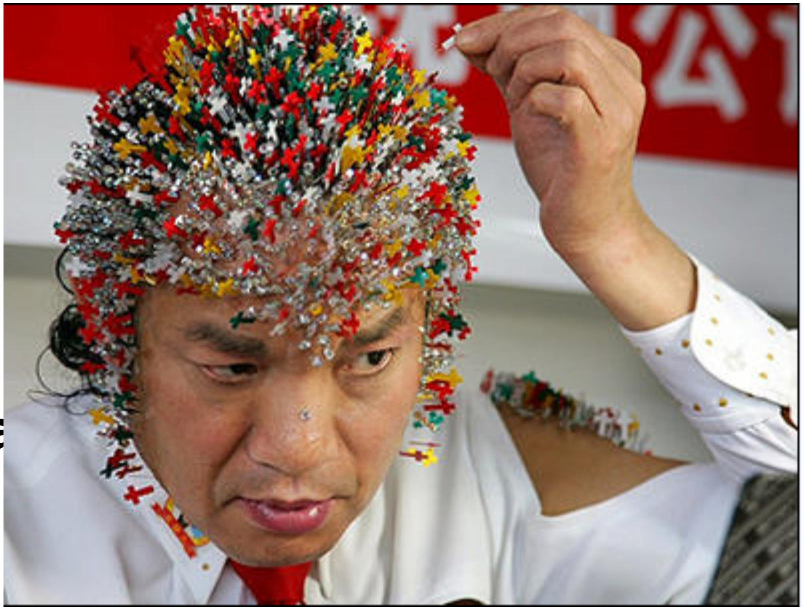
- **P**in **I**s **N**ot an acronym

Pin Instrumentation

- Multi-platform:
  - Linux
  - Windows
  - OS X (not supported anymore)
- Multi-architecture:
  - IA32
  - x86-64 (aka Intel64 / AMD64)
  - Itanium (aka IA64, only Linux)
  - ARM (not supported anymore)
- Robust & Efficient

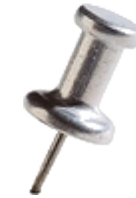
Pin availability

- **Popular and well supported**
  - 40,000+ downloads, 400+ citatio



- **Free Download**
  - **www.pintool.org**
  - **Free for non-commercial use**
  - **Includes: Detailed user manua Pin tools**
- **Pin User Group (PinHeads)**
  - **http://tech.groups.yahoo.com/group/pinheads/**
  - **Pin users and Pin developers answer questions**

Pin and Pin Tools



- Pin – the instrumentation **engine**
- Pin Tool – the instrumentation **program**

- Pin provides a framework, the Pin Tool uses the framework to do something meaningful

- Pin Tools
  - Written in C/C++ using Pin APIs
  - Many open source examples provided with the Pin kit
  - Certain do's and dont's apply

Instruction Counting Tool

```
#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;
}
```

Start Here

Instrumentation vs. Analysis

- Instrumentation routines define where instrumentation is inserted
  - e.g., before instruction
  - ☞ **Occurs *first time* an instruction is executed**

- Analysis routines define what to do when instrumentation is activated
  - e.g., increment counter
  - ☞ **Occurs *every time* an instruction is executed**

Instruction Counting Tool

```
#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;
}
```

Instrumentation Granularity

- Instruction
- Basic Block
- Trace

- Routine
- Section
- Image

- Process
- Thread
- Exception

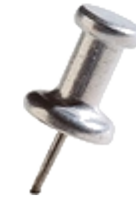
Instrumentation Points

IPOINT_BEFORE	Insert a call before an instruction or routine.
IPOINT_AFTER	Insert a call on the fall thorugh path of an instruction or return path of a routine.
IPOINT_ANYWHERE	Insert a call anywhere inside a trace or a bbl.
IPOINT_TAKEN_BRANCH	Insert a call on the taken edge of branch, the side effects of the branch are visible.

A Better Instruction Counting Tool

```
#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;
}
```

Summary of Part 2



- Pin is a dynamic binary instrumentation engine
- Pin is available freely for non-commercial purposes
- Pin is the engine, Pin Tools are programs controlling the engine

- Instrumentation routines are called once, analysis routines are called every time
- There are many levels of granularity
  - You should try to use the lowest answering your needs
- You can change instrumentation points

Part 3

A PRACTICAL EXAMPLE CATCHING “DOUBLE-FREE”

Main + Includes

```
#include "pin.H"
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <list>
```

```
int main(int argc, char *argv[])
{
    // Initialize pin & symbol manager
    PIN_InitSymbols();
    PIN_Init(argc,argv);

    // Register Image to be called to instrument functions.
    IMG_AddInstrumentFunction(Image, 0);

    PIN_StartProgram(); // Never returns

    return 0;
}
```

Instrumentation Routine

```
VOID Image(IMG img, VOID *v)  {
    // Find the malloc() function and add our function after it
    RTN mallocRtn = RTN_FindByName(img, "malloc");
    if (RTN_Valid(mallocRtn))
    {
        RTN_Open(mallocRtn);
        RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter,
                       IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);
        RTN_Close(mallocRtn);
    }
    // Find the free() function and add our function before it
    RTN freeRtn = RTN_FindByName(img, "free");
    if (RTN_Valid(freeRtn))
    {
        RTN_Open(freeRtn);
        RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)FreeBefore,
                       IARG_ADDRINT, "free",
                       IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                       IARG_END);
        RTN_Close(freeRtn);
    }
}
```

Analysis routines

```
list<ADDRINT> MallocAddrs;
```

```
VOID MallocAfter(ADDRINT ret)
{
   // Save teh address returned by malloc in our list
   MallocAddrs.push_back(ret);
}
```

```
VOID FreeBefore(CHAR * name, ADDRINT target)
{
   list<ADDRINT>::iterator p;

   p = find(MallocAddrs.begin(), MallocAddrs.end(), target);
   if (MallocAddrs.end() != p)
   {
       p = MallocAddrs.erase(p);
   }   // Delete this from the allocated address list
   else
   {   // We caught a Free of an un-allocated address
       cout << hex << target << endl;
   }   // Using cout is not a good practice, I do it for the example only
}
```

Summary of part 3

- It is relatively simple to write Pin Tools
- Writing a tool to catch double free is very simple and takes less than 50 lines of actual code
- Using simple tools we can catch vulnerabilities relatively easily
- Did anyone notice the flaw in the tool?

Part 4

ADVANCED STUFF AND CLOSING NOTES

Probe mode and JIT mode

- JIT Mode
  - Pin creates a modified copy of the application on-the-fly
  - Original code never executes

    ➢ More flexible, more common approach

- Probe Mode
  - Pin modifies the original application instructions
  - Inserts jumps to instrumentation code (trampolines)

    ➢ Lower overhead (less flexible) approach

Advanced Pin APIs

- Transparent debugging and extending the debugger
- Attaching and detaching
- System call instrumentation
- Managing Exceptions and Signals
- Instrumenting a process tree
- Accessing Decode API
- CONTEXT* and IARG_CONST_CONTEXT, IARG_CONTEXT
- Fast buffering API
- Pin Code-Cache API

Alternative instrumentation engines

- DyanmoRIO
- Valgrind
- Dtrace
- SystemTap (based on kprobes)
- Frysk
- GDB can also be seen as a DBI
- Bistro (EOL)

- Add your favorite DBI engine here

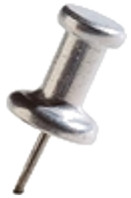
Learning more about Pin

- Website – www.pintool.org
  - Free Pin kit downloads
  - Many open source Pin Tool examples
  - An extensive manual
  - Links to papers and extensive tutorials given in conferences
- PinHeads - Mailing list / newsgroup
  - http://tech.groups.yahoo.com/group/pinheads/

Call for action

- Start learning and using binary instrumentation
- Create your own Pin Tools and share with the DC9723 community and all the security community

- Did anybody come up with a tool idea?
  - Feel free to contact me



QUESTIONS?

Contact details:

- gal.diskin@intel.com
- www.diskin.org