Win API hooking by using DBI: log me, baby!

Ricardo J. Rodríguez

rjrodriguez@unizar.es

Centro Universitario de la Defensa Zaragoza

NcN 2017

NoConName 2017

Barcelona, Spain

Credits of some slides thanks to *Gal Diskin*

Agenda

1 What is Dynamic Binary Instrumentation (DBI)?

2 The Pin framework

3 Developing your Own Pintools
- Developing Pintools: How-to
- Logging WinAPIs (for fun & profit)
  - Windows File Format
  - Uses

4 Conclusions

Dynamic Binary Instrumentation

DBI: Dynamic Binary Instrumentation

Main Words

Instrumentation	??
Dynamic	??
Binary	??

Instrumentation?

Instrumentation

- “Being able to observe, monitor and modify the behaviour of a computer program” (Gal Diskin)
- Arbitrary addition of code in executables to collect some information

Instrumentation?

Instrumentation

- “Being able to observe, monitor and modify the behaviour of a computer program” (Gal Diskin)
- Arbitrary addition of code in executables to collect some information
- Analyse and control everything around an executable code
  - Collect some information
  - Arbitrary code insertion

Dynamic Binary Instrumentation

Dynamic?

Code analysis

- Static
  - BEFORE execution
  - All possible execution paths are explored → not extremely good for performance
- Dynamic
  - DURING the execution
  - Just one execution path (it may depend on the input data!)

Binary?

Dynamic analysis

- Source code available
  - Source code
  - Compiler
- No source code (common case ☺)
  - Binary
    - Static (i.e., creating a new binary – with extras)
    - Dynamic
  - Environment
    - Emulation
    - Virtual
  - Debugging

Dynamic Binary Instrumentation

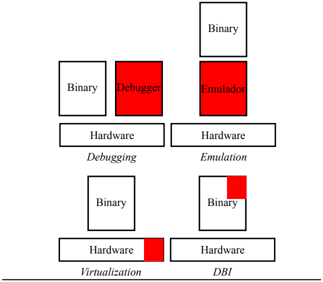
Instrumentation	Controlling what is happening. . .
Dynamic	upon execution. . .
Binary	of a binary program

Dynamic Binary Instrumentation

Placing DBI in the context of dynamic analysis

Definition (informal)

- Executable transformation
- Total control over execution
- No need of architectural support



- Virtualization
  - Total control?
- Emulation
  - Executable transformation
- Debugging
  - Architectural support (a must. . .)

J-Y. Marion, D. Reynaud *Dynamic Binary Instrumentation for Deobfuscation and Unpacking. DeepSec*, 2009

Dynamic Binary Instrumentation

Pin

- **Developed by Intel**, announced in 2005
- Three Letter Acronyms @ Intel
  - 26^3 possible TLAs; $26^3 - 1$ currently in use at Intel
  - **Only 1 not approved for use at Intel. Guess one** $\ddot\smile$
  - Pin Is Not an acronym
- **Supports Linux and Windows in both 32-bit and 64-bit architectures**
  - IA32
  - x86-64 (Intel64/AMD64)
  - Itanium (IA64, only for Linux)
- **Alllows for attaching already running processes**

Dynamic Binary Instrumentation

The Pin framework

Components

- **Pin**
  - **Instrumentation engine**
- **Pintool**
  - **Instrumentation tool**
  - Uses the instrumentation engine to build something useful
  - Written in C/C++
  - Lot of examples shipped with Pin

Dynamic Binary Instrumentation

The Pin framework

Different types of APIs

- **Basic APIs are architecture independent**:
  - Common functionalities (control-flow changes or memory accesses)
- **Architecture-specific API**: opcodes and operands
- **Call-based APIs**:
  - Instrumentation routines: defines WHERE instrumentation is inserted. Only called on the first time
  - Analysis routines: defines WHAT to do when instrumentation is activated. Called every time the object is reached
  - Callbacks routines: called whenever a certain event happens

Dynamic Binary Instrumentation

The Pin framework

Different types of APIs

- **Basic APIs are architecture independent**:
  - Common functionalities (control-flow changes or memory accesses)
- **Architecture-specific API**: opcodes and operands
- **Call-based APIs**:
  - Instrumentation routines: defines WHERE instrumentation is inserted. Only called on the first time
  - Analysis routines: defines WHAT to do when instrumentation is activated. Called every time the object is reached
  - Callbacks routines: called whenever a certain event happens

Analysis modes

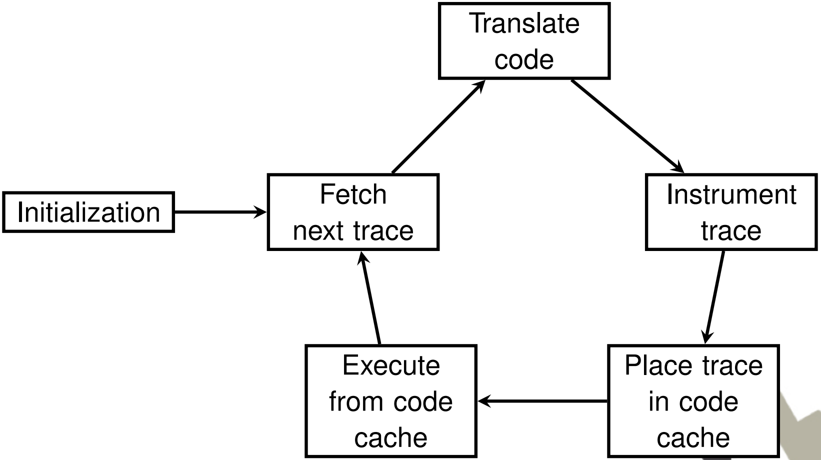
- **JIT mode**
  - Modified copy, on-the-fly
  - Original code never executes

- **Probe mode**
  - Original application instructions are modified
  - Jumps inserted (trampolines)

Dynamic Binary Instrumentation

The Pin framework

How Pin executes



Granularity provided by Pin

- **Low-level view**
  - Instruction (INS)
  - Basic block (BBL): sequence of instructions ending in some branch instruction
    - Single entry point, single exit point
  - Trace (TRACE; also called Super basic block)
    - Single entry point, multiple exit points

Granularity provided by Pin

- **Low-level view**
  - Instruction (INS)
  - Basic block (BBL): sequence of instructions ending in some branch instruction
    - Single entry point, single exit point
  - Trace (TRACE; also called Super basic block)
    - Single entry point, multiple exit points
- **Program-level view**
  - Routine (RTN)
  - Section (SEC)
  - Image (IMG)

Granularity provided by Pin

- **Low-level view**
  - Instruction (INS)
  - Basic block (BBL): sequence of instructions ending in some branch instruction
    - Single entry point, single exit point
  - Trace (TRACE; also called Super basic block)
    - Single entry point, multiple exit points
- **Program-level view**
  - Routine (RTN)
  - Section (SEC)
  - Image (IMG)
- **System-level view**
  - Process, thread, exception, syscalls, . . .

Instrumentation Points

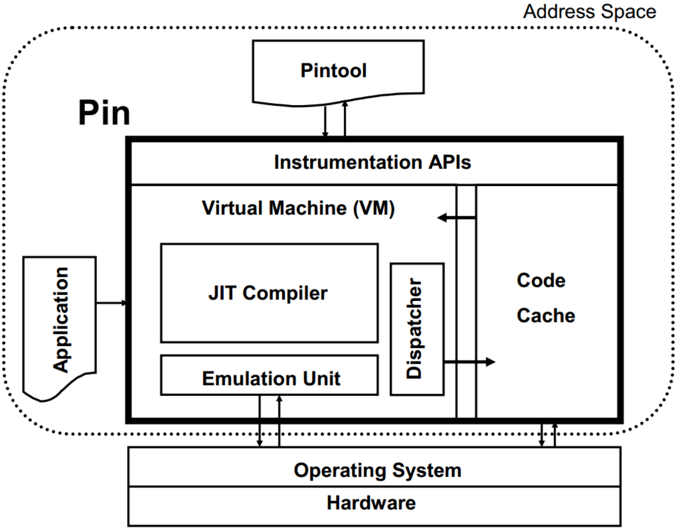
- `IPOINT_BEFORE`
  - Insert a call before an instruction or routine
- `IPOINT_AFTER`
  - Insert a call on the fall through path of an instruction or return path of a routine
- `IPOINT_ANYWHERE`
  - Insert a call anywhere inside a trace or a BBL
- `IPOINT_TAKEN_BRANCH`
  - Insert a call on the taken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

Analysis routine parameters (few examples)

- `IARG_INST_PTR`
  - Instruction pointer (program counter) value
- `IARG_UINT32 <value>`
  - An integer value
- `IARG_REG_VALUE <register name>`
  - Value of the register specified
- `IARG_BRANCH_TARGET_ADDR`
  - Target address of the branch instrumented
- `IARG_MEMORY_READ_EA`
  - Effective address of a memory read
- **More and more available, check the Pin documentation**

Dynamic Binary Instrumentation

The Pin framework



Developing your Own Pintools

Setting up the environment

1. Install VC++ compiler (+ Visual Studio, if you like)
   - I haven't tested with gcc, feel free to do it and let me know the result ⌣̈
2. Download the correct Pin framework to your VC++
   - https://software.intel.com/en-us/articles/pin-a-binary-instrumentation-tool-downloads

```
MSVC++ 9.0  _MSC_VER == 1500 (Visual Studio 2008)
MSVC++ 10.0 _MSC_VER == 1600 (Visual Studio 2010)
MSVC++ 11.0 _MSC_VER == 1700 (Visual Studio 2012)
MSVC++ 12.0 _MSC_VER == 1800 (Visual Studio 2013)
MSVC++ 14.0 _MSC_VER == 1900 (Visual Studio 2015)
MSVC++ 14.1 _MSC_VER >= 1910 (Visual Studio 2017)
```

3. Unzip in your drive
4. (Optional) If you want to use VS, follow this tutorial to configure it properly:

   http://blog.piotrbania.com/2011/06/compling-pintools-with-microsoft-visual.html

That' all!

Developing your Own Pintools

Example: inscount0.cpp

```
#include <iostream>
#include <fstream>
#include "pin.H"

ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every instruction is executed
VOID docount() { icount++; }

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v){
    // Insert a call to docount before every instruction
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

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

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v){
    // Write to a file since cout and cerr maybe closed by the application
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl; OutFile.close();
}

int main(int argc, char * argv[]){
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage(); //Usage() removed for readability
    OutFile.open(KnobOutputFile.Value().c_str());
    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);
    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);
    // Start the program, never returns
    PIN_StartProgram();
    return 0;
}
```

- `#include "pin.h"`
- `PIN_Init(argc, argv)`
    - Mandatory
    - Initialize Pin

Developing your Own Pintools

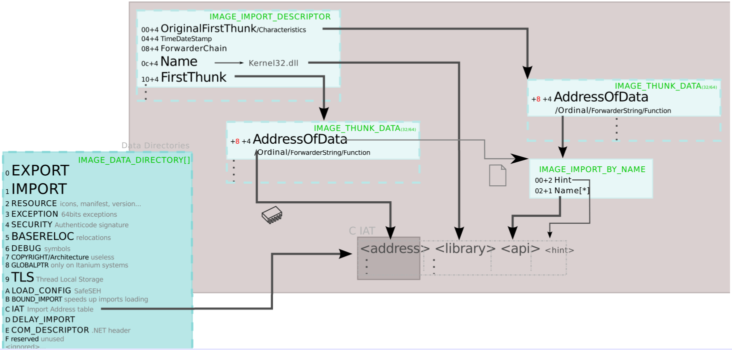
Example: inscount0.cpp

```
#include <iostream>
#include <fstream>
#include "pin.H"

ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every instruction is executed
VOID docount() { icount++; }

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v){
    // Insert a call to docount before every instruction
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

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

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v){
    // Write to a file since cout and cerr maybe closed by the application
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl; OutFile.close();
}

int main(int argc, char * argv[]){
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage(); //Usage() removed for readability
    OutFile.open(KnobOutputFile.Value().c_str());
    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);
    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);
    // Start the program, never returns
    PIN_StartProgram();
    return 0;
}
```

- `#include "pin.h"`
- `PIN_Init(argc, argv)`
  - Mandatory
  - Initialize Pin
- **Instrumentation routines**: `INS_AddInstrumentFunction`, `INS_InsertCall`
  - Prefix determines type of granularity
- **Analysis routine**: `docount`
- **End routines**: `INS_AddFiniFunction`
- `PIN_StartProgram()`
  - Starts execution and never returns

Windows File Format

Import symbols


IMAGE_IMPORT_DESCRIPTOR
00+4 OriginalFirstThunk/Characteristics
04+4 TimeDateStamp
08+4 ForwarderChain
0c+4 Name
Kernel32.dll
10+4 FirstThunk
IMAGE_THUNK_DATA(32/64)
+8 +4 AddressOfData
/Ordinal/ForwarderString/Function
IMAGE_THUNK_DATA(32/64)
+8 +4 AddressOfData
/Ordinal/ForwarderString/Function
IMAGE_IMPORT_BY_NAME
00+2 Hint
02+1 Name[*]
C IAT
<address> <library> <api> <hint>
Data Directories
IMAGE_DATA_DIRECTORY[]
0 EXPORT
1 IMPORT
2 RESOURCE icons, manifest, version...
3 EXCEPTION 64bits exceptions
4 SECURITY Authenticode signature
5 BASERELOC relocations
6 DEBUG symbols
7 COPYRIGHT/Architecture useless
8 GLOBALPTR only on Itanium systems
9 TLS Thread Local Storage
A LOAD_CONFIG SafeSEH
B BOUND_IMPORT speeds up imports loading
C IAT Import Address table
D DELAY_IMPORT
E COM_DESCRIPTOR .NET header
F reserved unused
<ignored>...

- Functions/data imported from DLLs. Located at `.idata` (usually)
- External DLLs are automatically loaded, their dependencies as well
- External addresses written to the Import Address Table (IAT)

Pintool examples

Example of WinAPI logging: detection of double-free vulnerabilities (naïf example)

Double-free vulnerabilities

- **Most common memory error vulnerability**
- **When a heap chunk is freed twice, w/o being reallocated in between**

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

char* reserveMemory(int size){
    char *temp = (char *) malloc(size);
    return temp;
}

int main(void){
    /* Create an array for storing dummy data */
    char *c = reserveMemory(10);
    printf("(malloc) %p\n", c);
    c[0] = 5;

    char *c2 = reserveMemory(10);
    printf("(malloc) %p\n", c2);
    free(c);
    free(c2);
    free(c2); // double free
    c[3] = 3;
}
```

Pintool examples

```
#include "pin.H"
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <list>
#include <string.h>
#include <stdio.h>
#include <ostream>

list<ADDRINT> MallocAddrs;

VOID FreeBefore(ADDRINT target, ADDRINT inst)
{
   list<ADDRINT>::iterator p;
   p = find(MallocAddrs.begin(), MallocAddrs.end(), target);
   if ( (!(MallocAddrs.empty())) && (MallocAddrs.end() != p) ){
       p = MallocAddrs.erase(p); // Delete this from the allocated @ list
   }else{   // We caught a Free of an un-allocated address
       cerr << "DOUBLE-FREE DETECTED: " << hex
                 << target << " @" << inst << endl;
   }  // Using cerr is not a good practice,
      // I do it oly for the sake of the example
}

VOID MallocAfter(ADDRINT ret, ADDRINT inst)
{
   // Save the address returned by malloc in our list
   if (ret != 0){
      list<ADDRINT>::iterator p;
      p = find(MallocAddrs.begin(), MallocAddrs.end(), ret);

      if (MallocAddrs.end() == p){ //not found
         MallocAddrs.push_back(ret);
         cerr << "Saving " << hex << ret
                  << " in the address list @" << inst << endl;
      }else{
         // malloc address already in the list?!
         cerr << "already saved" << hex << " @" << inst << endl;
      }
   }else{
         cerr << "Malloc fail" << endl;
   }
}
```

```
// Instrument the malloc() and free() functions.
// note that there are malloc and free in the os loader and in libc
VOID Image(IMG img, VOID *v)
{
    cerr << "Hooking img: " << IMG_Name(img) << endl;

    // Find the malloc() function and add our function after it
    RTN mallocRtn = RTN_FindByName(img, "malloc");
    if (RTN_Valid(mallocRtn)){
        // print function name
        cerr << "Function name: " << RTN_Name(mallocRtn) << endl;
        RTN_Open(mallocRtn);
        RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter,
                           IARG_FUNCRET_EXITPOINT_VALUE,
                           IARG_INST_PTR, IARG_END);
        // IARG_FUNCRET_EXITPOINT_VALUE function result,
        // valid only at return instruction
        RTN_Close(mallocRtn);
    }

    // Find the free() function and add our function before it
    RTN freeRtn = RTN_FindByName(img, "free");
    if (RTN_Valid(freeRtn)) {
        // print function name
        cerr << "Function name: " << RTN_Name(freeRtn) << endl;
        RTN_Open(freeRtn);
        RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)FreeBefore,
                           IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_INST_PTR,
                           IARG_END);
                // IARG_FUNCARG_ENTRYPOINT_VALUE int argument,
                // valid only at the entry point of a routine
        RTN_Close(freeRtn);
    }
}

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

    IMG_AddInstrumentFunction(Image, 0);
    PIN_StartProgram(); // Never returns

    return 0;
}
```

Pintool examples

Analysis of a malware sample – live demo

MD5: `0de9765c9c40c2c2f372bf92e0ce7b68`

Conclusions

Take-home messages

- **DBI allows us to (easily) execute arbitrary code at arbitrary locations upon execution of a binary program**
  - No (target) source needed
  - No relinking needed
- DBI frameworks available in the market: Pin, Valgrind, DynamoRio, ...
- **Pin provides a very extensive and rich API for developing your own analysis tools**
  - **Easy and fast prototyping**
  - Furthermore, **different granularity enriches the analysis capabilities!**

Hope to see your great tools next year!

Win API hooking by using DBI: log me, baby!

Ricardo J. Rodríguez

rjrodriguez@unizar.es

Centro Universitario de la Defensa Zaragoza

NcN 2017

NoConName 2017
Barcelona, Spain

Credits of some slides thanks to *Gal Diskin*