Light and Dark side of Code Instrumentation

Dmitriy “D1g1” Evdokimov
DSecRG, Security Researcher
#whoami

- Security Researcher in DSecRG
  - RE
  - Fuzzing
  - Mobile security
- Organizer: DCG #7812
- Editor in “XAKEP”
Agenda

1. Instrumentation .
2. Instrumentation ..
3. Instrumentation ...
4. Instrumentation ....
5. Instrumentation ..... 
6. Instrumentation ......
7. Instrumentation .......
“It has been proved by scientists that a new point of evolution, any technical progress appears when a Man makes up a new type of tool, but not a product.”
Instrumentation

Instrumentation is a technique adding extra code to a program/environment for monitoring/change some program behavior.
Why is it necessary?

- Simulation
- Virtualization
- Performance analysis
- Automated debugging
- Error detection
- Binary translation
- Parallel optimization
- Emulation
- Testing
- Collecting code metrics
- Optimization
- Memory debugging
- Software profiling
- Correctness checking
- Memory leak detection
Instrumentation in information security

- Control flow analysis
- Virtual patching
- Privacy monitoring
- Sandboxing
- Deobfuscation
- Behavior based security
- Transparent debugging
- Security enforcement

- Unpack
- Data flow analysis
- Security test case generation
- Code coverage
- Malware analysis
- Taint analysis
- Program shepherding
- Shellcode detection
- Reverse engineering
- Data Structure Restoring

- Vulnerability detection
- Antivirus technology
- Fuzzing
- Forensic
- Transparent debugging
- Security enforcement
## Analysis

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Static analysis</th>
<th>Dynamic analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code vs. data</td>
<td>Problem</td>
<td>No problem</td>
</tr>
<tr>
<td>Code coverage</td>
<td>Big (but not all)</td>
<td>One way</td>
</tr>
<tr>
<td>Information about values</td>
<td>No information</td>
<td>All information</td>
</tr>
<tr>
<td>Self-modifying code</td>
<td>Problem</td>
<td>No problem</td>
</tr>
<tr>
<td>Interaction with the environment</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Unused code</td>
<td>Analysis</td>
<td>No analysis</td>
</tr>
<tr>
<td>JIT code</td>
<td>Problem</td>
<td>No problem</td>
</tr>
</tbody>
</table>
Code Discovery

After static analysis

Memory

<table>
<thead>
<tr>
<th>Instr 1</th>
<th>Instr 2</th>
<th>Instr 3</th>
<th>Instr 4</th>
<th>Instr 5</th>
<th>Instr 6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>jump reg</td>
<td>DATA</td>
<td></td>
<td>PADDING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jmp 0x0ABCD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After dynamic analysis

www.dsecrg.com
The general scheme of code instrumentation

1. Find points of instrumentation;
2. Insert instrumentation;
3. Take control from program;
4. Save context of the program;
5. Execute own code;
6. Restore context of the program;
7. Return control to program.
Source Data

Source code

Byte code

Binary code

```c
if (FirstChance)
{
    DEBUG_VALUE Reg, Ecx, Edx;
    // Query EIP, EAX and ECX
    if (g_Registers->GetValue(g_Registers->GetValue(g_Registers->GetValue(g_Registers->GetValue(;
        char szParam[MAX_PATH];
        ULONG ReadedBytes = 0;
        // Read current instr
        ZeroMemory(szParam, si
        HRESULT Hr = g_DataSpa
        if (Hr != S_OK)
```

```
0x0 const/4 v5 ,[# + 0], {0}
0x2 const/4 v4 ,[# + 0], {0}
0x4 invoke-super v6 , [meth@
0xa sget-boolean v2 , [field@
0xe if-eqz v2 , [+ 30]
0x12 invoke-static v6 , [meth
0x18 move-result-object v0
0x1a if-eqz v0 , [+ 24]
0x1e new-instance v2 , [type
0x22 invoke-virtual v6 , [met
0x28 move-result-object v3
0x2a invoke-direct v2 , v6 , v
0x30 const/high16 v3 ,[# + 81
0x34 invoke-virtual v2 , v3 , [  
```

```
push ebp
mov ebp, esp
sub esp, 28h
mov eax, __security_c
xor eax, ebp
mov [ebp+var_8], eax
mov [ebp+b], 0
mov eax, [ebp+param]
push eax
; a
call ??func_next@@YAHHA2
add esp, 4
mov [ebp+b], eax
mov ecx, [ebp+input]
push ecx
lea edx, [ebp+buf]
```

www.dsecrg.com

CONFidence Krakow 2012
Classification of target instrumentation

- With source code
  - Source code
  - Linker/Compiler

- Without source code
  - Byte code
    - Byte code
    - Interpreter/VM
  - Binary code
    - Executable file
    - Process
    - Environment
    - Hardware

- Link-time/Compilation-time instrumentation
- Static instrumentation
  - Load-time
  - Dynamic
Source code instrumentation

• Source code*
  – Source code instrumentation
    • Manual skills
    • Plugins for IDE
  – Link-time/Compilation-time instrumentation
    • Options of linker/compiler

• Tools: Visual Studio Profiler, gcc, TAU, OPARI, Diablo, Phoenix, LLVM, Rational Purify, Valgrind

*Unreal condition for security specialist =)
Unmoral programming

/*
 +
 +
 +
 +
 */


#include <stdio.h>

<stdio.h> in file main.c

int main() {
    FILE *p;
    int A, k, a, r, i;

    // Initialize variables
    //...

    return 0;
}

[14/40]
Byte code instrumentation

Byte code – intermediate representation between source code and machine code.

Java VM  Dalvik VM  AVM/AVM2  CLR

www.dsecrg.com  CONFidence Krakow 2012
Instrumentation byte-code (I)
Instrumentation byte code (II)

• Byte-code
  – Static instrumentation
    • Static byte code instrumentation
  – Load-time instrumentation
    • Custom byte code loader
  – Dynamic instrumentation
    • Dynamic byte-code instrumentation
Instrumentation Java (I)

Mechanisms:
- java.lang.instrument package;
- Java Platform Debugger Architecture (JPDA).

```
Components                      Debugger Interfaces

/                             <-------- JVM TI - Java VM Tool Interface
/                             
debuggee --------------------(-----------------------------------
| VM                          
| back-end                    
|                             <-------- JDWP - Java Debug Wire Protocol
|
comm_channel ----(            |
|                             
| front-end                   |
|                             <-------- JDI - Java Debug Interface
|                             |
| UI                          |
```
Instrumentation Java (II)

- Static instrumentation
  - Modification *.class files
- Load-time instrumentation
  - ClassFileLoadHook
  - Custom ClassLoader
- Dynamic instrumentation
  - ClassFileLoadHook -> RetransformClasses

Tools: Javassist, ObjectWeb ASM, BCEL, JOIE, reJ
JavaSnoop, Serp, JMangler
Instrumentation .NET

• Static instrumentation
  – Modification DLL files

• Load-time instrumentation
  – AppDomain.Load()/Assembly.Load()
  – Joint redirection
  – Via event handler

Tools: ReFrameworker, MBEL, RAIL, Cecil
Instrumentation ActionScript (I)

• ActionScript2
  – AVM
  – Tags that (can) contain bytecode:
    • DefineButton (7), DefineButton2 (34), DefineSprite (39), DoAction (12), DoInitAction (59), PlaceObject2 (26), PlaceObject3 (70).

• ActionScript3
  – AVM2
  – Tags that (can) contain bytecode:
    • DoABC (82), RawABC (72).
AVM2 Architecture

AS3
function (x:int):int {
    return x+10
}

.getlocal 1
.pushint 10
.add
.returnvalue

MIR
@1 arg +8 // argv
@2 load [@1+4]
@3 imm 10
@4 add (@2,@3)
@5 ret @4 // @4:eax

x86
mov eax,(eap+8)
mov eax,(eax+4)
add eax,10
ret
Instrumentation ActionScript (I)

Original SWF file

Header

AVM tag

Tags

Instrumented SWF file
Instrumentation AVM (II)

• Static instrumentation
  – Add:
    • trace()
    • dump()
    • debug()
    • debugfile()
    • debugline()
  – Modification:
    • Create own class + change class name = hook!
Instrumentation binary code

- The executable file
  - Static code instrumentation
    - Static binary instrumentation
- Process
  - Debuggers
    - Debugging API
  - Modifying call table/other structure
    - IAT
    - ...
  - Dynamic code instrumentation
    - Dynamic binary instrumentation
- Hardware
  - Hardware debug features
    - Debug registers
    - Hardware debuggers
    - ...

- Environment
  - Modifying call table
    - IDT, CPU MSRs, GDT, SSDT, IRP table
    - ...
  - Modifying OS options
    - SHIM
    - LD_PRELOAD
    - AppInt_DLLs
    - DLL injection
    - ...
  - Reproduction of the environment
    - Emulation
    - Virtualization
Static Binary Instrumentation (I)

Static binary instrumentation/Physical code integration/Static binary code rewriting

• Realization:
  – With reallocation:
    • Level of segment;
    • Level of function;
  – Without reallocation.
Static Binary Instrumentation (II)

Reallocation:
1) Function Displacement + Entry Point Linking;
2) Branch Conversion;
3) Instruction Padding;
4) Instrumentation.

Tools: DynInst, EEL, ATOM, PEBIL, ERESI, TAU, Vulcan, BIRD, Aslan(4514N)
Debuggers

- Breakpoints:
  - Software
  - Hardware

- Debugger + scripting:
  - WinDBG + pykd
  - OllyDBG + python = Immunity Debuggers
  - GDB + PythonGDB

- Python library's*: Buggery, IDAPython, ImmLIB, lldb, PyDBG, PyDbgEng, pygdb , python-ptrace , vtrace, WinAppDbg, ...

*See “Python Arsenal for Reverse Engineering”
Dynamic Binary Instrumentation

Dynamic binary instrumentation/Virtual code integration/Dynamic binary rewriting

Tools: PIN, DynamoRIO, DynInst, Valgrind, BAP, KEDR, Fit, ERESI, Detour, Vulcan, SpiderPig
Dynamic Binary Instrumentation

- Dynamic Binary Instrumentation (DBI) is a process control and analysis technique that involves injecting instrumentation code into a running process.
- Dynamic binary analysis (DBA) tools such as profilers and checkers help programmers create better software.
- Dynamic binary instrumentation (DBI) frameworks make it easy to build new DBA tools.

DBA tools consist:
- instrumentation routines;
- analysis routines.
Kinds of DBI

Mode:
- user-mode;
- kernel-mode.

Modes of execution:
- Interpretation-mode;
- Probe-mode;
- JIT-mode.

Mode of work:
- Start to finish;
- Attach.

Performance vs. Functionality:
- JIT
- Probe
## DBI Frameworks*

<table>
<thead>
<tr>
<th>Frameworks</th>
<th>OS</th>
<th>Arch</th>
<th>Modes</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIN</td>
<td>Linux, Windows, MacOS</td>
<td>x86, x86-64, Itanium, ARM</td>
<td>JIT, Probe</td>
<td>Attach mode</td>
</tr>
<tr>
<td>DynamoRIO</td>
<td>Linux, Windows</td>
<td>x86, x86-64</td>
<td>JIT, Probe</td>
<td>Runtime optimization</td>
</tr>
<tr>
<td>DynInst</td>
<td>Linux, FreeBSD, Windows</td>
<td>x86, x86-64, ppc32, ARM, ppc64</td>
<td>Probe</td>
<td>Static &amp; Dynamic binary instrumentation</td>
</tr>
<tr>
<td>Valgrind</td>
<td>Linux, MacOS</td>
<td>x86, x86-64, ppc32, ARM, ppc64</td>
<td>JIT</td>
<td>IR – VEX, Heavyweight DBA tools</td>
</tr>
</tbody>
</table>

*For more details see “DBI: Intro” presentation from ZeroNights conference*
Start work with DBI

```c
C:\\> type dbi.txt
//Launching PIN (JIT mode)
pin <pinargs> -t <pintool> <pintoolargs> -- <app> <appargs>
//Launching DynamoRIO (JIT mode)
drrun <drrunargs> -client <client> <clientargs> <app> <appargs>
//Launching Valgrind (JIT mode)
valgrind <valgrindargs> --tool=<toolname> <app> <appargs>
//Launching PIN (Probe mode)
pin -probe -t <pintool> <pintoolargs> -- <app> <appargs>
//Launching DynamoRIO (Probe mode)
drrun -mode probe -client <client> <clientargs> <app> <appargs>
//Attaching to a process in PIN
pin <pinargs> -t <pintool> <pintoolargs> -pid <app_pid>
C:\\>
```
Levels of granularity

- Instruction;
- Basic Block*;
- Trace/Superblock;
- Function;
- Section;
- Events;
- Binary image.
Self-modifying code & DBI

```c
void InsertSmcCheck()
{
    traceAddr = (VOID *) TRACE_Address(trace);
    traceSize = TRACE_Size(trace);
    TraceCopyAddr = malloc(traceSize);
    if (traceCopyAddr != 0) {
        memcpy(TraceCopyAddr, traceAddr, traceSize);
        TRACE_InsertCall(trace, IPOINT_BEFORE, (AFUNPTR)DoSmcCheck, IAGR_PTR, traceAddr,
                         IAGR_PTR, traceCopyAddr, IAGR_UINT32, traceSize, IAGR_CONTEXT, IAGR_END);
    }
}

void DoSmcCheck(VOID * traceAddr, VOID * traceCopyAddr, USIZE traceSize, CONTEXT * ctxP) {
    if (memcmp(traceAddr, traceCopyAddr, traceSize) != 0) {
        smcCount++;
        free(traceCopyAddr);
        CODECACHE_InvalidateTrace((ADDRINT)traceAddr);
        PIN_ExecuteAt(ctxP);
    }
}

void main (int argc, char **argv) {
    PIN_Init(argc, argv);
    TRACE_AddInstrumentationFunction(InsertSmcCheck, 0);
    PIN_StartProgram();
}
```
Overhead

\[ O = X + Y \]
\[ Y = N \times Z \]
\[ Z = K + L \]

O – Tool Overhead;
X – Instrumentation Routines Overhead;
Y – Analysis Routines Overhead;
N – Frequency of Calling Analysis Routine;
Z – Work Performed in the Analysis Routine;
K – Work Required to Transition to Analysis Routine;
L – Work Performed Inside the Analysis Routine.
Rewriting instructions

• Platforms:
  – with fixed-length instruction;
  – with variable-length instructions.

Distribution by length

- 2 bytes: 28.9%
- 3 bytes: 20.4%
- 4 bytes: 15.6%
- 5 bytes: 10.0%
- 6 bytes: 9.0%
- 7 bytes: 8.5%
- 8 bytes: 6.2%
- 9 bytes: 1.0%
- 10 bytes: 0.0%
- 11 bytes: 0.3%
- Others: 0.1%
Rewriting code (I)

- Easy / simple / boring / regular example
  - Rewriting prolog function

```assembly
; int __stdcall CFileOpenBrowser__GetPBITItemFromCSIDL@CFileOpenBrowser@@QAEHKPAU

; public: int __thiscall CFileOpenBrowser::OnSetCursor@CFileOpenBrowser@@QAEHXZ proc near
```

```assembly
; SUBROUTINE =========

db 5 dup(90h)
```

```assembly
; mov __edi, edi
push ebp
mov ebp, esp
push ebx
push esi
mov esi, [ebp+ppidl]
push esi
push [ebp+csidl] ; csidl
xor ebx, ebx
push ebx
; hwnd
call ds:__imp__SHGetSpecialFolderLocation@12
```

```assembly
test eax, eax
il short loc 763044c3
```

www.dsecrg.com

CONFidence Krakow 2012
Rewriting code (II)

• Hardcore example:
  – Mobile phone firmware rewriting

![Diagram of mobile phone firmware rewriting process]

- Bootloader
- Flash
- SHELLCODE 1
- AMSS
- GSM
- Baseband processor
- Malicious SMS
- Reboot
Instrumentation in ARM

ARM modes:

- ARM
  - Length(instr) = 4 byte
- Thumb
  - Length(instr) = 2 byte
- Thumb2
  - Length(instr) = 2/4 byte
- Jazzle

For more detail see “A Dynamic Binary Instrumentation Engine for the ARM Architecture” presentation.
Emulation

App1

Emulator

OS

OS

Processor
Instrumentation & Bochs

• Bochs can be called with instrumentation support.

  ```
  ./configure [...] --enable-instrumentation
  ./configure [...] --enable-instrumentation="instrument/stubs"
  ```

• C++ callbacks occur when certain events happen:
  - Poweron/Reset/Shutdown;
  - Branch Taken/Not Taken/Unconditional;
  - Opcode Decode (All relevant fields, lengths);
  - Interrupt /Exception;
  - Cache /TLB Flush/Prefetch;
  - Memory Read/Write.

• “bochs-python-instrumentation” patch by Ero Carrera
Virtualization

Native VMM

Hosted VMM

*VMM - Virtual Machine Monitor
Instrumentation & virtualization

Stages:
1. Save the VM-exit reason information in the VMCS;
2. Save guest context information;
3. Load the host-state area;
4. Transfer control to the hypervisor;
5. Run own code.

*VMCS - Virtual Machine Control Structure*
## Instrumentation in Mobile World

<table>
<thead>
<tr>
<th>Mobile Platform</th>
<th>Language</th>
<th>Executable file format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>Java</td>
<td>Dex</td>
</tr>
<tr>
<td>iOS</td>
<td>Objective-C</td>
<td>Mach-O</td>
</tr>
<tr>
<td>Windows Phone</td>
<td>.NET</td>
<td>PE</td>
</tr>
</tbody>
</table>
Conclusion

One can implement instrumentation of everything!
Contact

Twitter: @evdokimovdsvds
E-mail: d.evdokimov@dsecreg.com
Windows 8

- Apps:
  - C++ & DirectX
  - C# & XAML
  - HTML & JavaScript & CSS