Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection

Brendan Dolan-Gavitt*, Tim Leek†, Michael Zhivich†, Jonathon Giffin*, and Wenke Lee*

* Georgia Institute of Technology
† MIT Lincoln Laboratory

This work was sponsored by IARPA under Air Force Contract FA8721-05-C-0002. Opinions, interpretation, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.
Virtual Machine Introspection

Security VM

Security Apps

Introspection

Hypervisor

Guest VM (insecure)
Open Problem: The Semantic Gap

• Isolation can provide security

• Isolation makes it hard to see what’s going on

• View exposed by VMM is low-level (physical memory, CPU state)

• Need to reconstruct high-level view using introspection routines
What You Want...

Files

Processes

Networking

Drivers
What You Get
Introspection Challenges

- Introspection routines are currently built manually
- Building routines requires detailed knowledge of OS internals
- Often requires reverse engineering
- OS updates and patches break existing introspection utilities
Contributions

• We generate introspection routines automatically

• No knowledge of OS internals or reverse engineering required

• Routines can be regenerated easily for new OS versions / patches
Idea: Code Extraction

Security VM

Guest VM

Hypervisor
Idea: Code Extraction

Security VM   Guest VM

Hypervisor

Oakland ’11  Virtuoso  5/24/2011
Idea: Code Extraction

Security VM

Guest VM

Hypervisor
Idea: Code Extraction

Security VM

Introspection

Guest VM

Hypervisor
Idea: Code Extraction

Security VM

Introspection

Guest VM

Hypervisor

Tuesday, May 24, 2011
Idea: Code Extraction

Security VM

Introspection

Guest VM

Hypervisor

Tuesday, May 24, 2011
Goals

- **Generality**: generate useful introspection programs on multiple operating systems
- **Reliability**: generate working programs using dynamic analysis
- **Security**: ensure that programs are unaffected by guest compromise
Challenges

• Assume no prior knowledge of OS internals

• Code extraction must be whole-system
  • Much of the code we want is in the kernel
  • Existing work (BCR, Inspector Gadget) only extracts small pieces of userland code
Training Environment

Trace Logger

Instruction Traces

Training Phase
Overview

Security VM

Runtime

Introspection
Program

COPY
ON
WRITE

Untrusted VM

User

Kernel

Runtime Phase

Tuesday, May 24, 2011
Training

- Write in-guest *training program* using system APIs

```c
#define __WIN32_LEAN_AND_MEAN__
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
#include <stdio.h>
#include "vmnotify.h"

int main(int argc, char **argv) {
    EnumProcesses(pids, 256, &outcb);
    return 0;
}
```
Training

- Write in-guest *training program* using system APIs

```c
#define __WIN32_LEAN_AND_MEAN__
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
#include <stdio.h>
#include "vmnotify.h"

int main(int argc, char **argv) {
    DWORD *pids = (DWORD *) malloc(256);
    DWORD outcb;

    EnumProcesses(pids, 256, &outcb);

    return 0;
}
```
Training

• Annotate program with start/end markers

```c
#define __WIN32_LEAN_AND_MEAN__
#include <windows.h>
#include <psapi.h>
#pragma comment(lib, "psapi.lib")
#include <stdio.h>
#include "vmnotify.h"

int main(int argc, char **argv) {
    DWORD *pids = (DWORD *) malloc(256);
    DWORD outcb;

    vm_mark_buf_in(&pids, 4);
    EnumProcesses(pids, 256, &outcb);
    vm_mark_buf_out(pids, 256);
    return 0;
}
```
Training

• Run program in QEMU to generate *instruction trace*

• Traces are in QEMU µOp format

```
INTERRUPT(0xfb, 0x200a94, 0x0)
TB_HEAD_EIP(0x80108028)
MOVL_T0_IM(0x0)
OPREG_TEMPL_MOVL_A0_R(0x4)
SUBL_A0_4()
OPS_MEM_STL_T0_A0(0x1, 0xf186fe8, 0x8103cfe8,
                     0xffffffff, 0x215d810, 0x920f0, 0xfb)
OPREG_TEMPL_MOVL_R_A0(0x4)
MOVL_T0_IM(0xfb)
OPREG_TEMPL_MOVL_A0_R(0x4)
SUBL_A0_4()
OPS_MEM_STL_T0_A0(0x1, 0xf186fe4, 0x8103cfe4,
                     0xffffffff, 0x215d810, 0x920f0, 0xfb)
```
Whole-System Traces

• Includes all instructions between start and end markers
• Includes software and hardware interrupts and exceptions
• Includes concrete addresses of memory reads/writes
• What subset of this trace is relevant?

• Initial preprocessing:
  • Remove hardware interrupts
  • Replace malloc/realloc/calloc with summary functions

• Next, executable dynamic slicing (Korel and Laski, 1988) is done to identify relevant instructions
1. Follow data def/use chain backward, starting with output buffer

2. Examine CFG and add necessary control flow statements to slice (and their dependencies)

3. Perform *slice closure*:
   - If *any* instance of an instruction is included in the slice, *all* instances of that instruction must be marked
• Since analysis is dynamic, we only see one path through program

• So: run program multiple times and then merge results
Trace Merging

- Since analysis is dynamic, we only see one path through the program.
- So: run the program multiple times and then merge the results.
Program Translation

- Goal: convert in-guest $\rightarrow$ out-of-guest
- Generates Python code that runs inside Volatility memory analysis framework
- Changes:
  - Memory reads come from guest VM
  - Memory writes are copy-on-write
  - CPU registers become local vars
Translation Example

Original x86

```
test byte [ebp+0x1c], 0x10
mov edi, ebx
jnz 0xc02533a9
```

QEMU µOps

```
[TB @0xc0253368L *]
  IFLO_TB_HEAD_EIP(0xc0253368)
  IFLO_INSN_BYTES(0xc0253368, 'f6451c10')
  * IFLO_OPREG_TEMPL_MOVX_A0_R(0x5)
  * IFLO_ADDL_A0_IM(0x1c)
  * IFLO_OPS_MEM_LDUB_T0_A0(...)
  * IFLO_MOVX_T1_IM(0x10)
  * IFLO_TESTL_T0_T1_CC()
    IFLO_INSN_BYTES(0xc025336c, '89df')
  * IFLO_OPREG_TEMPL_MOVX_T0_R(0x3)
  * IFLO_OPREG_TEMPL_MOVX_R_T0(0x7)
    IFLO_INSN_BYTES(0xc025336e, '7539')
  * IFLO_SET_CC_OP(0x16)
  * IFLO_OPS_TEMPLATE_JZ_SUB(0x0, 0x1)
    IFLO_GOTO_TB1(0x60afcab8)
    IFLO_MOVX_EIP_IM(0xc0253370)
    IFLO_MOVX_T0_IM(0x60afcab9)
    IFLO_EXIT_TB()
```
Translation Example

QEMU µOps

[TB @0xc0253368L *]
IFLO_TB_HEAD_EIP(0xc0253368)
IFLO_INSN_BYTES(0xc0253368,'f6451c10')
* IFLO_OPS_MEM_LDUB_T0_A0(...)
* IFLO_MOVVL_T1_IM(0x10)
* IFLO_TESTL_T0_T1_CC()
  IFLO_INSN_BYTES(0xc025336e,'89df')
* IFLO_OPS_MEM_LDVAL_T0_R(0x7)
* IFLO_INSN_BYTES(0xc025336e,'7539')
* IFLO_SET_CC_OP(0x16)
* IFLO_OPS_TEMPLATE_JZ_SUB(0x0,0x1)
  IFLO_GOTO_TB1(0x60afcab8)
  IFLO_MOVVL_EIP_IM(0xc0253370)
  IFLO_MOVVL_T0_IM(0x60afcab9)
  IFLO_EXIT_TB()

Python

A0 = EBP
A0 += UInt(0x1c)
T0 = UInt8(mem.read(A0,1))
T1 = UInt(0x10)
CC_DST = T0 & T1
T0 = EBX
EDI = T0
CC_OP = 0x16
if (Byte(CC_DST) == 0):
  raise Goto(0xc0253370)
raise Goto(0xc02533a9)
Results: Generality

- Generated 6 useful introspection programs on each of 3 operating systems
Introspection Programs

- **getpid**: Gets the PID of the currently running process.
- **pslist**: Gets a list of PIDs of all running processes.
- **getpsfile**: Gets the name of an executable from its PID.
- **lsmod**: Gets the base addresses of all kernel modules.
- **getdrvfile**: Gets the name of a kernel module from its base address.
- **gettime**: Gets the current system time.
Results: Reliability

- Analysis is dynamic, so programs may be incomplete
- How many traces are needed to produce reliable programs?
- Complicating factors: caching, difficulty of deciding ground truth for coverage
Results: Security

• Verified that introspection programs are not affected by in-guest code manipulation

• Training program (**pslist**) generated on clean system

• Resulting introspection program still detects processes hidden by Hacker Defender

• Note: DKOM attacks can still be effective against Virtuoso
Limitations

- Multiple processes/IPC
- Multithreaded code (synchronization)
- Code/data relocation (ASLR)
- Self-modifying code
Conclusions

- Programs generated by Virtuoso can be useful, reliable, and secure
- Uses novel whole-system executable dynamic slicing and merging
- Virtuoso can greatly reduce time and effort needed to create introspection programs
  - Weeks of reverse engineering vs. minutes of computation