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.
Standard virtualization security layout: insecure guests, isolated security apps in their own VM.
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*

Isolation is not a panacea; it makes it hard to see what’s going on. The view the VMM gives is not what we want.
These are the objects relevant to security monitoring.
What You Get

This is the view exposed by the VMM -- physical memory. This is from a memory dump of a Windows 2003 system.
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.

Note story about sec. vendor that had to spend 60 hours reverse engineering Vista’s new TCP/IP stack. Virtuoso can reduce this to a few minutes by a non-expert.
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

Only programmer’s knowledge of public system APIs needed.
Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.
Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.
Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.
Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.
Idea: Code Extraction

Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.
Idea: Code Extraction

Basic idea: guest VM already has code that does the introspection. So we extract it out to another VM, transform it to introspect on another VM. Now even if original is compromised we have a known-good copy.
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

These goals mirror the results we present later on.
Challenges

• Assume no prior knowledge of OS internals
• Code extraction must be \textit{whole-system}
  • Much of the code we want is in the kernel
• Existing work (BCR, Inspector Gadget) only extracts small pieces of userland code
Make clear here that we write a small in-guest program that gets the data we want!
Once we have the traces, we process them and translate them into an out-of-guest introspection program.
The introspection program can then be deployed to a Security VM to monitor our untrusted VM and applications.
• 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
#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;
}
```

Of course, you need a little bit of boilerplate.
Next, inform Virtuoso of where logging should begin and end, and where the buffer containing the output of the introspection is.
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(0x1,0x8103cfe8,0x215d810,0x920f0,0xfb)
```

This produces instruction traces. They’re not x86, but QEMU.
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

Memory reads/writes are necessary so that we can do data flow analysis later.
Trace Analysis

• 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

System is doing a lot of other stuff that may not be relevant to the introspection at hand. HW interrupts: clearly not relevant. Malloc & friends: implementation artifact.
Data def/use chains give us our initial set of relevant instructions without control flow. Next step operates on dynamic CFG. Slice closure is what makes it executable.
Trace Merging

• Since analysis is dynamic, we only see one path through program

• So: run program multiple times and then merge results

Note why we didn’t go with static analysis here: too much domain knowledge.
• Since analysis is dynamic, we only see one path through program

• So: run program multiple times and then merge results

Note why we didn’t go with static analysis here: too much domain knowledge.
Program Translation

• Goal: convert in-guest → 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

Why Python? Volatility written in Python. Any code generation would do though. Copy-on-write is necessary so as not perturb the guest.
Example: a test and a conditional jump. Asterisks mean “included in slice”.

Original x86

```
test byte [ebp+0x1c], 0x10
mov edi, ebx
jnz 0xc02533a9
```
Translation to Python: conditional jump is now Python if statement.
Results: Generality

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

Windows: everyone uses it. Linux: we use it. Haiku: we don’t know its internals, no temptation to cheat.
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.

Describe these by group and why relevant to security: examine features of processes and drivers.
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

Caching: early runs may execute much more code. Difficulty of ground truth: hard to say what the complete set of code is, or how many paths (program testing has this problem too).
This is cross-evaluation: take 24 traces, and then take differently sized random subsets to create final program. Describe axes, then walk through one program => not reliable, 12 programs => pretty reliable. Mention caching effect again as explanation for why this graph is pessimistic.
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

DKOM is something we’ll look at in future work.
Limitations

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

Multiple processes: key problem is that we don’t know where data for a specific process might be at runtime. Multithreaded code: VM is paused, so waiting on a lock is bad. Relocation: where’s our data? Self-modifying code: code is only translated once (kernel’s
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