Guarded Modules: Adaptively Extending the VMM’s Privileges Into the Guest

Kyle C. Hale
Peter Dinda

Department of Electrical Engineering and Computer Science
Northwestern University

http://halek.co
http://presciencelab.org
http://v3vee.org
http://xstack.sandia.gov/hobbes
Redefining the boundaries between VMM and guest OS
Redefining the boundaries between VMM and guest OS
We want to evolve the VMM-guest relationship

...where the interface between the two is more flexible

...and where parts of the VMM may actually live inside the guest

The latter is the focus of this work
GEARS*

VMM Guest

Virtual service

privileged operations
Guarded Module can still interact with guest
How can we isolate and protect pieces of code in a guest OS that run at higher privilege than the rest of the guest?

How can we allow legacy code continue to use guest functionality?

We show how with two examples
Palacios VMM

- OS-independent, embeddable VMM
- Support for multiple host OSes (Linux, Kitten LWK)
- Open source, available at http://v3vee.org/palacios

**Palacios**
An OS Independent Embeddable VMM
Outline

• Motivation
• Christoization
• Threat Model and Runtime Invariants
• Runtime System and Border Crossings
• Examples
  – Selectively Privileged PCI Passthrough
  – Selectively Privileged MONITOR/MWAIT
• Conclusions
Guarded Module Transformation

Linux Kernel Module

Guarded Module

guarded module compiler

Christoization
Christoization

Compile-time and link-time *wrapping*

we use gcc toolchain to instrument (wrap) all calls out of and into the module

Christo and Jeanne-Claude wrap the Reichstag, Berlin, 1995
The guest is not to be trusted

We assume a threat model in which a malicious kernel wants to hijack a service’s privilege

Execution paths entering and leaving the guarded module must be checked
We maintain control flow integrity

Christoization allows VMM to trap all entries into and exits from module

VMM validates the environment for unauthorized changes to execution path (e.g. return oriented attacks)
We maintain control flow integrity

Guarded Module

Guest

Environment ok?
Entry point valid?

VMM

return

= privileged operation
...and code integrity

Guarded Module

write to module code

Guest

VMM

= privileged operation
...and data integrity
What we don’t provide

Parameter checking

Module cloaking

Currently no support for interactions between guarded modules
Programmer’s perspective

1. Write a Linux kernel module (or use an existing one)
2. Run it through our guarded module compiler (christoization)
3. Optional: verify identified module entry points
4. Pass to administrator, who registers guarded module with VMM at runtime
RECAP: Guarded Modules are guest context virtual services

...that can have elevated privilege, are protected from the untrusted guest that they run in, yet can still use its functionality

The implementation is small: ~220 lines of Perl, ~260 lines of Ruby, and ~1000 lines of C

(includes both the GM compilation toolchain and runtime system)

available online at http://v3vee.org/palacios
Runtime: module entries/exits trap to the VMM

We call these trapped events *border crossings*
Wrapper stubs

exit_wrapped:
  popq %r11
  pushq %rax
  movq $border_out_call, %rax
  vmmcall
  popq %rax
  callq exit
  pushq %rax
  movq $border_in_ret, %rax
  vmmcall
  popq %rax
  pushq %r11
  ret (to into guarded module)

  Trap to VMM, record environment
  Trap to VMM, Check integrity of environment
Guest Kernel

Guarded Module

Border Crossings

Border-out

Border-in

VMM

Border Control State Machine

Privileged VMM Access

Privileged Hardware Access

Hardware
Typical Border Crossing

\[
\text{guarded module} \quad \xrightarrow{\text{printk()}} \quad \text{return} \quad \text{module\_init()} \quad \xrightarrow{\text{privileged operation}} \quad \text{guest}
\]
Nested Border Crossings

guarded module

guest

foo() bar()
callback()
return

= privileged operation
Border Crossing from External Event

Guarded module

guest

iret
interrupt

= privileged operation
Experimental Setup

• Dell PowerEdge R415
  – 2 sockets, 4 cores each => 8 total cores
  – 2.2 GHz AMD Opteron 4122
  – 16 GB memory
  – Host kernel: Fedora 15 with Linux 2.6.38
  – Guest: single vcore with Busybox environment, Linux 2.6.38
System-independent overhead is low

Cost of any VM exit
...but ensuring control-flow integrity is expensive
Outline

• Motivation
• Christoization
• Threat Model and Runtime Invariants
• Runtime System and Border Crossings

• Examples
  – Selectively Privileged PCI Passthrough
  – Selectively Privileged MONITOR/MWAIT

• Related Work and Conclusions
We transformed a NIC driver into a guarded module

no manual modifications to NIC driver!
Selectively expose the PCI BARs to the guest

- guarded module 1: privileged mode
- guarded module 2: unprivileged mode

PCI I/O and memory BARs

Hardware NIC
Bandwidth drops, but border crossing count is very high!

Each border crossing is ~16,000 cycles (7.3 µs)

<table>
<thead>
<tr>
<th></th>
<th>Packet Sends</th>
<th>Packet Receives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Border-in</td>
<td>1.06</td>
<td>4.64</td>
</tr>
<tr>
<td>Border-out</td>
<td>1.06</td>
<td>4.64</td>
</tr>
<tr>
<td>Border Crossings / Packet Send</td>
<td>2.12</td>
<td></td>
</tr>
<tr>
<td>Border Crossings / Packet Receive</td>
<td>9.28</td>
<td></td>
</tr>
</tbody>
</table>

Many of these are leaf functions!
We implemented an adaptive idle loop with selective privilege

MONITOR/MWAIT instructions allow a CPU to go into a low-power state until a write occurs to a region of memory

```
MONITOR [addr]
...
MWAIT
```
VMs can’t typically use these instructions

Puts *physical* core to sleep. Other VMs/processes on that core will starve

But if the guest *knows* how many guests are on the machine (VMM state), we can let it run these instructions when idling

Can’t let untrusted parts of guest hijack this capability!
Adaptive mwait_idle() as a guarded module
**Scenario 1:**  
*a single vcore*

**Scenario 2:**  
*vcores sharing a physical core*
Guarded Modules as *adaptive, guest-context virtual services*

We’re leveraging VMM global information about the environment

That information is *only* exposed to the guarded module—this presents a new way to adapt VMs at runtime
Related Work

*Nooks* – isolate faulty code in kernel modules with wrappers. Kernel requires modification, protecting guest from modules

[Swift, SOSP ‘03]

*LXFI, SecVisor* – protect kernel against attack with VMM-authorized code

[Mao, SOSP ’11], [Seshadri, SOSP ‘07]

*SIM* – guest-resident VMM code, but special-purpose, uses completely separate address space

[Sharif, CCS ‘09]
Conclusions

We’ve shown the feasibility of adaptively extending the VMM into the guest with guarded modules.

General technique to automatically transform kernel modules into guarded modules.

Two proof-of-concept examples:
- Selective Privilege for Commodity NIC driver
- Selective Privilege for MONITOR/MWAIT
Future Work

Feasibility of automatically inlining leaf functions into modules (making them more self-contained)

Further motivating examples for guarded modules

Generalization of guarded modules—modules with VMM-controlled, specialized execution modes
We are rethinking system software interfaces

This talk focused on virtualization, but we’re thinking bigger (HW/VMM/OS/app)

It’s time to reconsider the structure of our system software stacks

We can adapt software services, but can we adapt their organization/structure?
For more info:

Kyle C. Hale
http://halek.co
http://presciencelab.org
http://v3vee.org
http://xstack.sandia.gov/hobbes