Defeating Signed BIOS Enforcement

- Corey Kallenberg
- John Butterworth
- Sam Cornwell
- Xeno Kovah

MITRE

BIOS Rootkits

- **Very powerful due to being the first code executed on the platform.**
- **Can leverage System Management Mode, which is the most powerful mode of execution on the x86 platform.[1]**
- **Survives OS reinstalls.**

- **However, we don't see many "in the wild" BIOS Rootkits.**
  - Less portable and more difficult to implement than their OS level counterparts.
  - Perhaps we will see more in the future as the OS becomes more locked down.

[1] There is a lot of prior work on leveraging SMM for nefarious purposes, I encourage you to look it up…

Recent Noteworthy BIOS Security Results

- **“Hardware Backdooring Is Practical” by J. Brossard**
  - Contrary to previous thinking, BIOS rootkits are not that difficult to implement thanks to opensource firmware projects.
- **“A Tale Of One Software Bypass Of Windows 8 Secure Boot” by Bulygin et al.**
  - If you can get onto the flash chip, you can defeat Secure Boot.
- **“BIOS Chronomancy” by Butterworth et al.**
  - BIOS Rootkits can defeat TPM detection.
  - BIOS Rootkits can survive BIOS reflashes.

Related Work

- **“Attacking Intel BIOS” by Rafal Wojtczuk and Alexander Tereshkin**
  - Exploited memory corruption vulnerability in parsing of unsigned custom bootup picture in signed Intel BIOS.
  - Allowed reflashing BIOS with arbitrary (malicious) image despite signed enforcement.
  - Blackhat USA 2009

Protecting your BIOS

- The previous results are dependent on an attacker being able to get a foothold on the SPI flash chip that contains your platform firmware (BIOS or UEFI).
- Signed firmware update enforcement protects against malicious writes to the flash chip.
- Most new systems offer or even require signed firmware update enforcement.[1]

[1] More on this later…

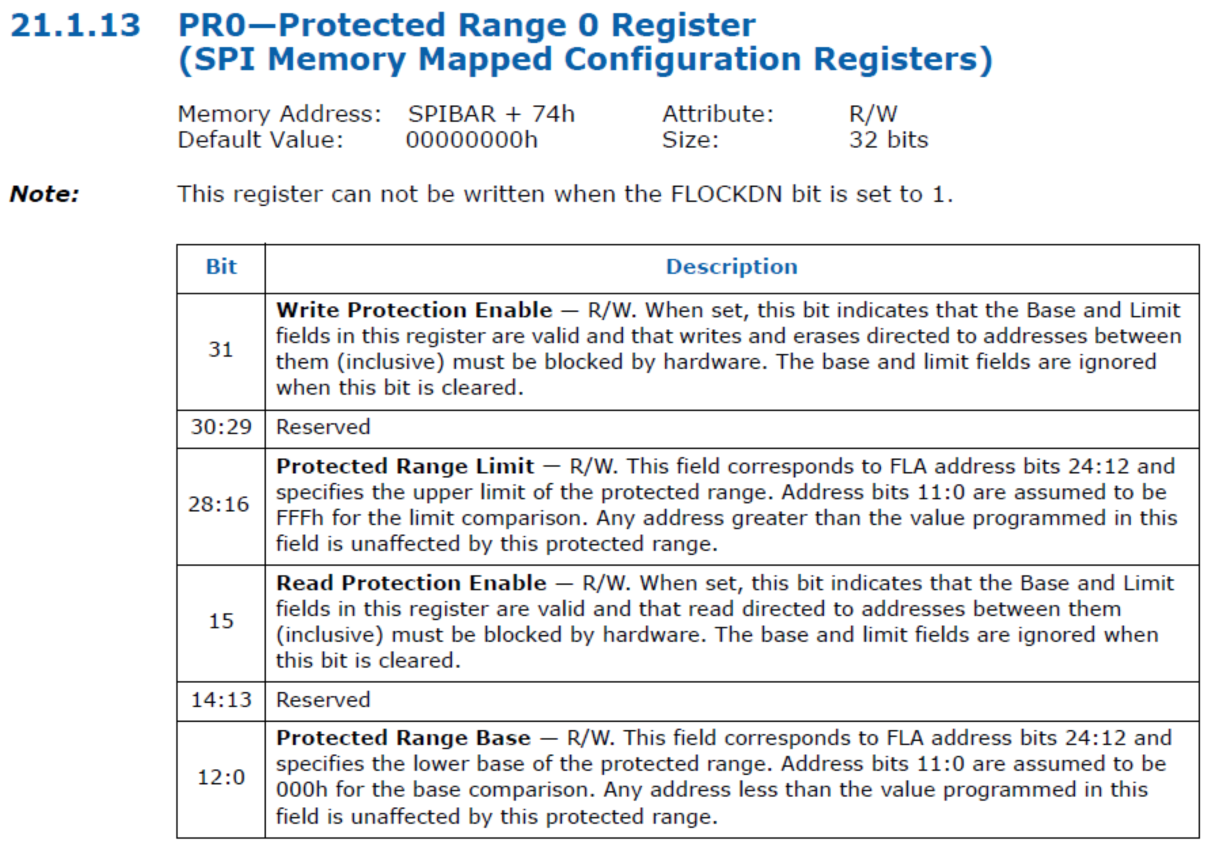
How is this implemented?

- **Intel provides a number of protection mechanisms that can “lock down” the flash chip.**
  - You can read all about these in the ICH documentation for your chipset.
  - These protections have remained relatively static recently.
- **It’s then up to the OEM to leverage these flash lock down mechanisms to roll their own signed bios enforcement.**
  - This includes correctly configuring a surprisingly complicated set of flash lock down controls…
  - As well as implementing an update routine that doesn’t contain any bugs…

Follow Along

- **I encourage you to download a tool my colleagues and I have written to read out your flash lock down configuration.**
  - http://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/copernicus-question-your-assumptions-about
  - Direct link to binary: http://www.mitre.org/sites/default/files/copernicus_pr.zip

Protected Range SPI Flash Protections



21.1.13 PR0—Protected Range 0 Register (SPI Memory Mapped Configuration Registers)

Memory Address:	SPIBAR + 74h	Attribute:	R/W
Default Value:	00000000h	Size:	32 bits

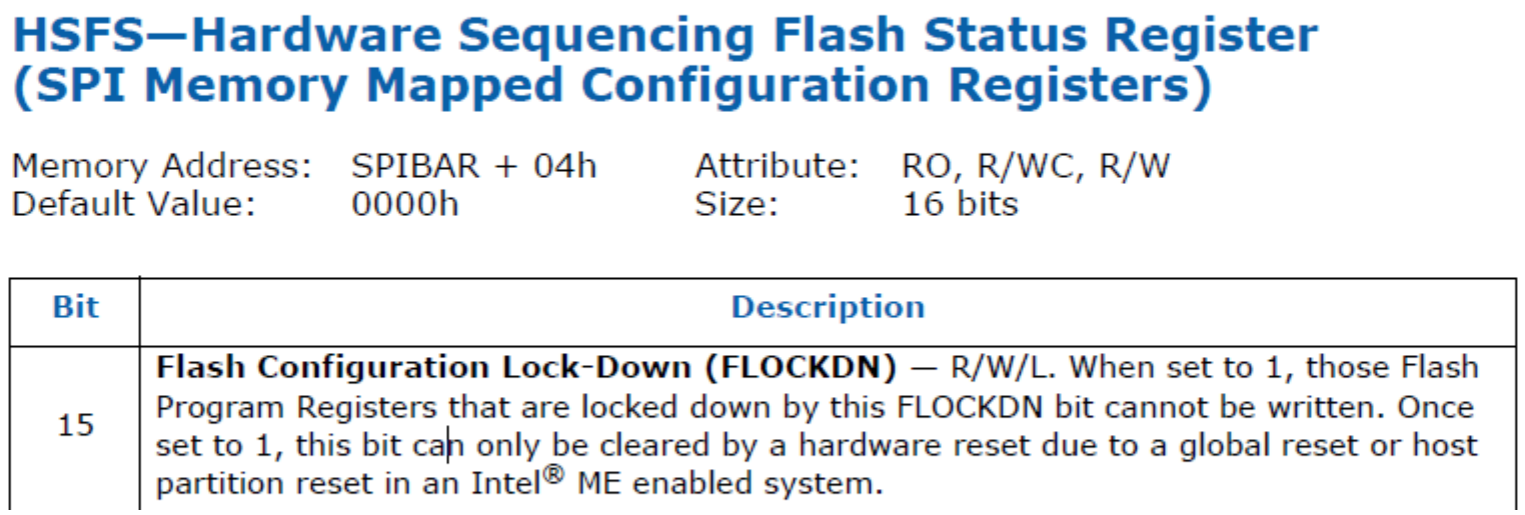
Note: This register can not be written when the FLOCKDN bit is set to 1.

Bit	Description
31	**Write Protection Enable** — R/W. When set, this bit indicates that the Base and Limit fields in this register are valid and that writes and erases directed to addresses between them (inclusive) must be blocked by hardware. The base and limit fields are ignored when this bit is cleared.
30:29	Reserved
28:16	**Protected Range Limit** — R/W. This field corresponds to FLA address bits 24:12 and specifies the upper limit of the protected range. Address bits 11:0 are assumed to be FFFh for the limit comparison. Any address greater than the value programmed in this field is unaffected by this protected range.
15	**Read Protection Enable** — R/W. When set, this bit indicates that the Base and Limit fields in this register are valid and that read directed to addresses between them (inclusive) must be blocked by hardware. The base and limit fields are ignored when this bit is cleared.
14:13	Reserved
12:0	**Protected Range Base** — R/W. This field corresponds to FLA address bits 24:12 and specifies the lower base of the protected range. Address bits 11:0 are assumed to be 000h for the base comparison. Any address less than the value programmed in this field is unaffected by this protected range.

- **Protected Range registers can provide write protection to the flash chip.**

http://www.intel.com/content/www/us/en/chipsets/6-chipset-c200-chipset-datasheet.html

HSFS.FLOCKDN



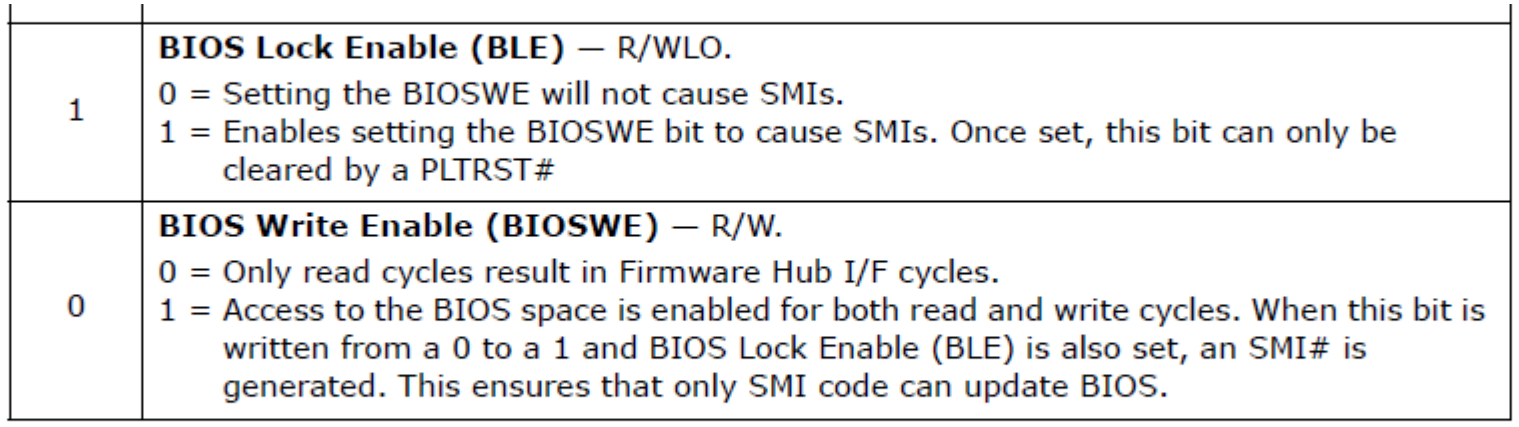
HSFS—Hardware Sequencing Flash Status Register (SPI Memory Mapped Configuration Registers)

Memory Address:	SPIBAR + 04h	Attribute:	RO, R/WC, R/W
Default Value:	0000h	Size:	16 bits

Bit	Description
15	**Flash Configuration Lock-Down (FLOCKDN)** — R/W/L. When set to 1, those Flash Program Registers that are locked down by this FLOCKDN bit cannot be written. Once set to 1, this bit can only be cleared by a hardware reset due to a global reset or host partition reset in an Intel® ME enabled system.

- **HSFS.FLOCKDN bit prevents changes to the Protected Range registers once set.**

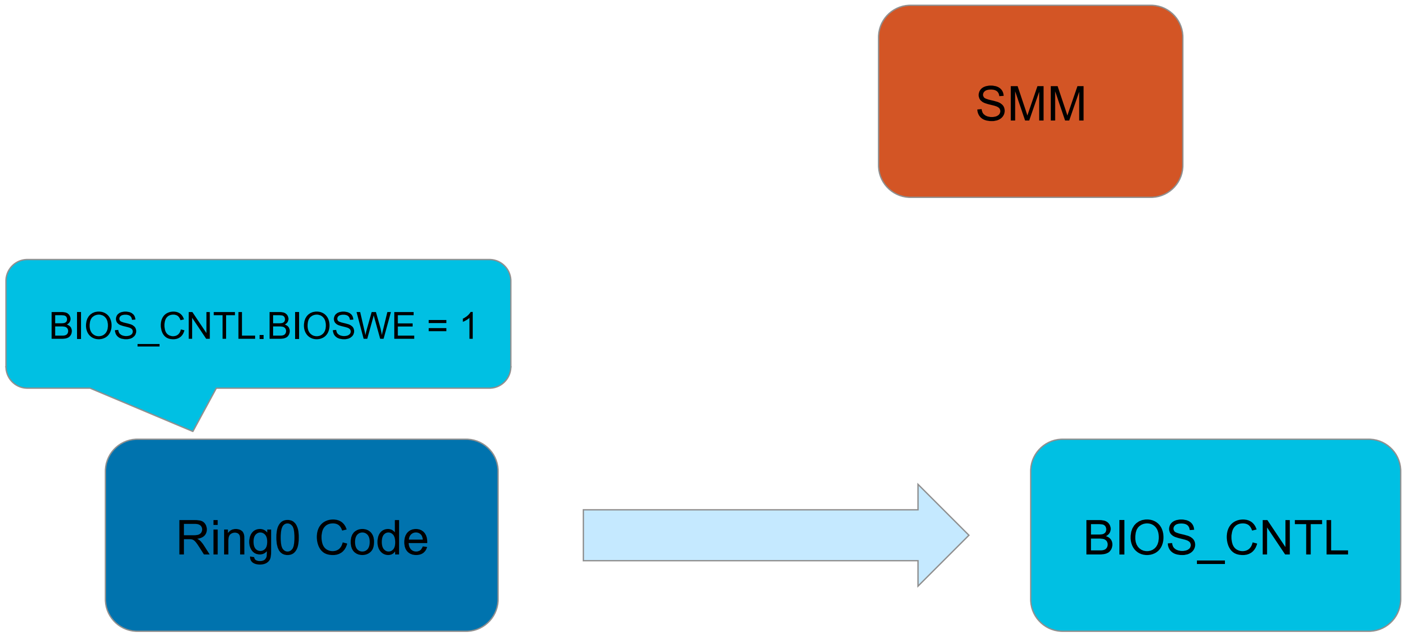
BIOS_CNTL



1	**BIOS Lock Enable (BLE)** — R/WLO. 0 = Setting the BIOSWE will not cause SMIs. 1 = Enables setting the BIOSWE bit to cause SMIs. Once set, this bit can only be cleared by a PLTRST#
0	**BIOS Write Enable (BIOSWE)** — R/W. 0 = Only read cycles result in Firmware Hub I/F cycles. 1 = Access to the BIOS space is enabled for both read and write cycles. When this bit is written from a 0 to a 1 and BIOS Lock Enable (BLE) is also set, an SMI# is generated. This ensures that only SMI code can update BIOS.

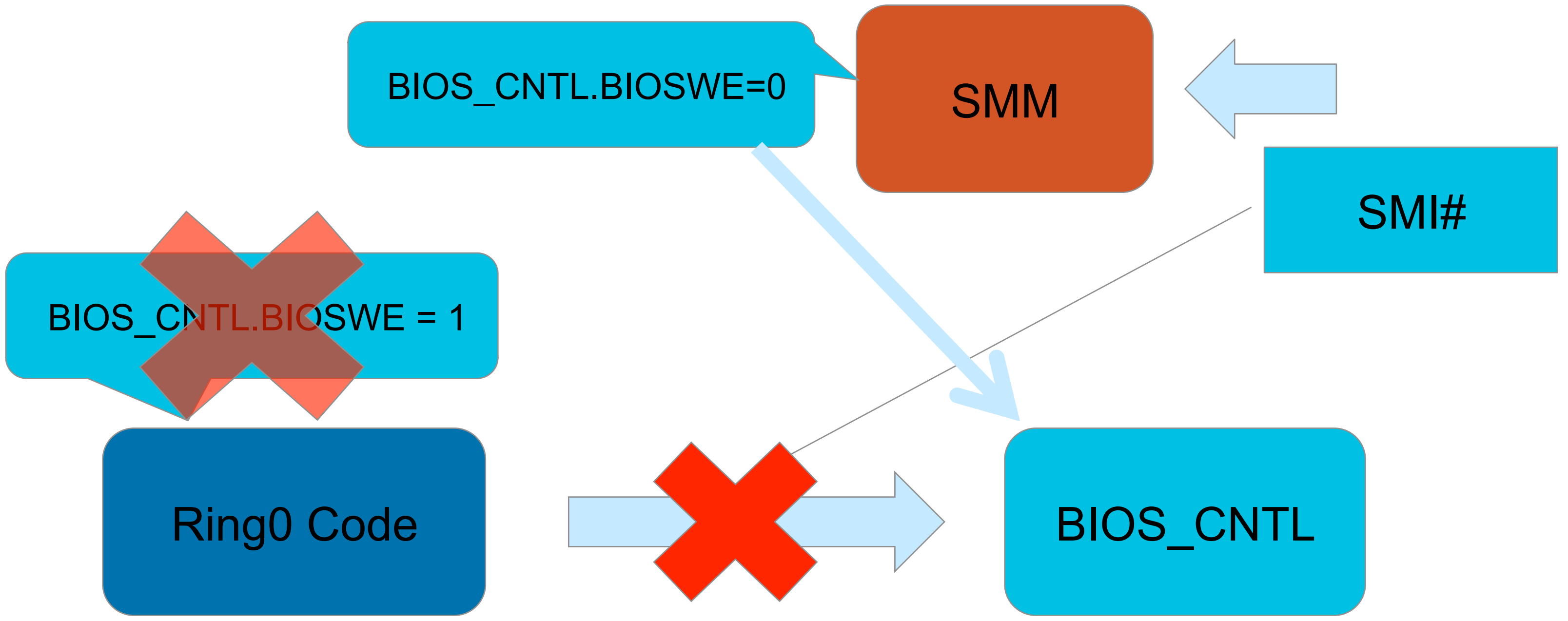
- **The above bits are part of the BIOS_CNTL register on the ICH.**
- **BIOS_CNTL.BIOSWE bit enables write access to the flash chip.**
- **BIOS_CNTL.BLE bit provides an opportunity for the OEM to implement an SMM routine to protect the BIOSWE bit.**

SMM BIOSWE protection (1 of 2)



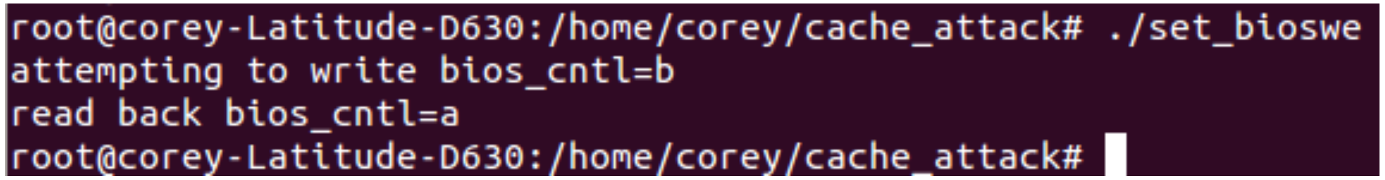
- **Here the attacker tries to set the BIOS Write Enable bit to 1 to allow him to overwrite the BIOS chip.**

SMM BIOSWE protection (2 of 2)



- The write to the BIOSWE bit generates an SMI.
- The SMI immediately writes 0 back to the BIOSWE bit.
- The end result is that BIOSWE is always 0 when non-SMM code is running.

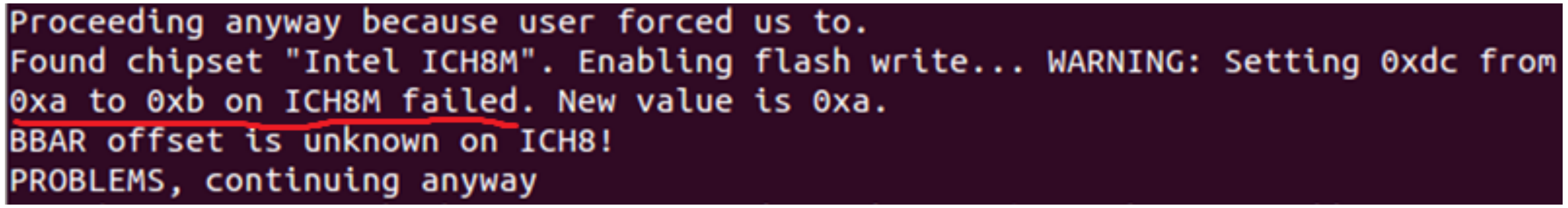
BIOSWE Protection Demo (1 of 2)



```
root@corey-Latitude-D630:/home/corey/cache_attack# ./set_bioswe
attempting to write bios_cntl=b
read back bios_cntl=a
root@corey-Latitude-D630:/home/corey/cache_attack#
```

- **Set_bioswe is a simple program that attempts to set the BIOSWE bit in the BIOS_CNTL register.**
- **BIOS_CNTL = 0xB implies BIOSWE is set.**
- **BIOS_CNTL = 0xA implies BIOSWE is not set.**
- **Notice that our attempt to set BIOSWE=1 in the above output has failed as SMM is protecting the BIOSWE value.**

BIOSWE Protection Demo (2 of 2)



```
Proceeding anyway because user forced us to.
Found chipset "Intel ICH8M". Enabling flash write... WARNING: Setting 0xdc from
0xa to 0xb on ICH8M failed. New value is 0xa.
BBAR offset is unknown on ICH8!
PROBLEMS, continuing anyway
```

- **Attempting to write to the BIOS chip using the flashrom open source utility fails because BIOS_CNTL=0xA (BIOSWE=0), implying write access is not allowed to the BIOS chip.[1]**

[1] Command: flashrom –p internal:laptop_I_want_a_brick,ich_spi_mode=swseq –w bios.bin

Intel Protections Summary

- The Protected Range and BIOS_CNTL registers provide duplicative protection of the SPI flash chip that contains the platform firmware.
- These protections are reset upon platform reset, and must be correctly configured by the platform firmware during power on.

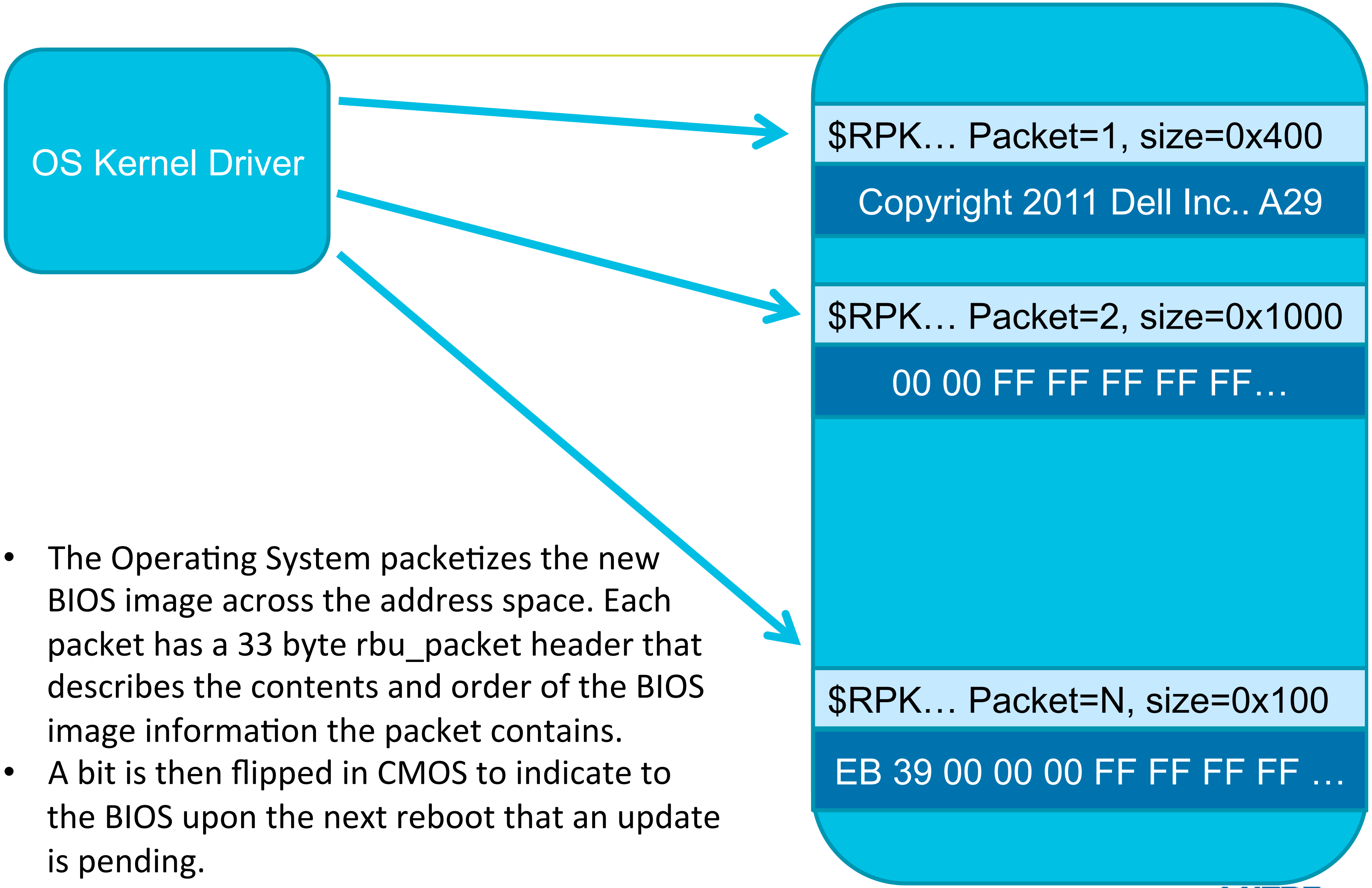
OEM BIOS Update Routines

- **We will use Dell BIOS as a case study in how OEM's use the Intel flash protection mechanisms to implement signed BIOS enforcement.**
- **The following code is from the Dell Latitude E6400 BIOS, but the BIOS update routine in question is shared among 20+ other Dell models.**

Dell E6400 BIOS Update

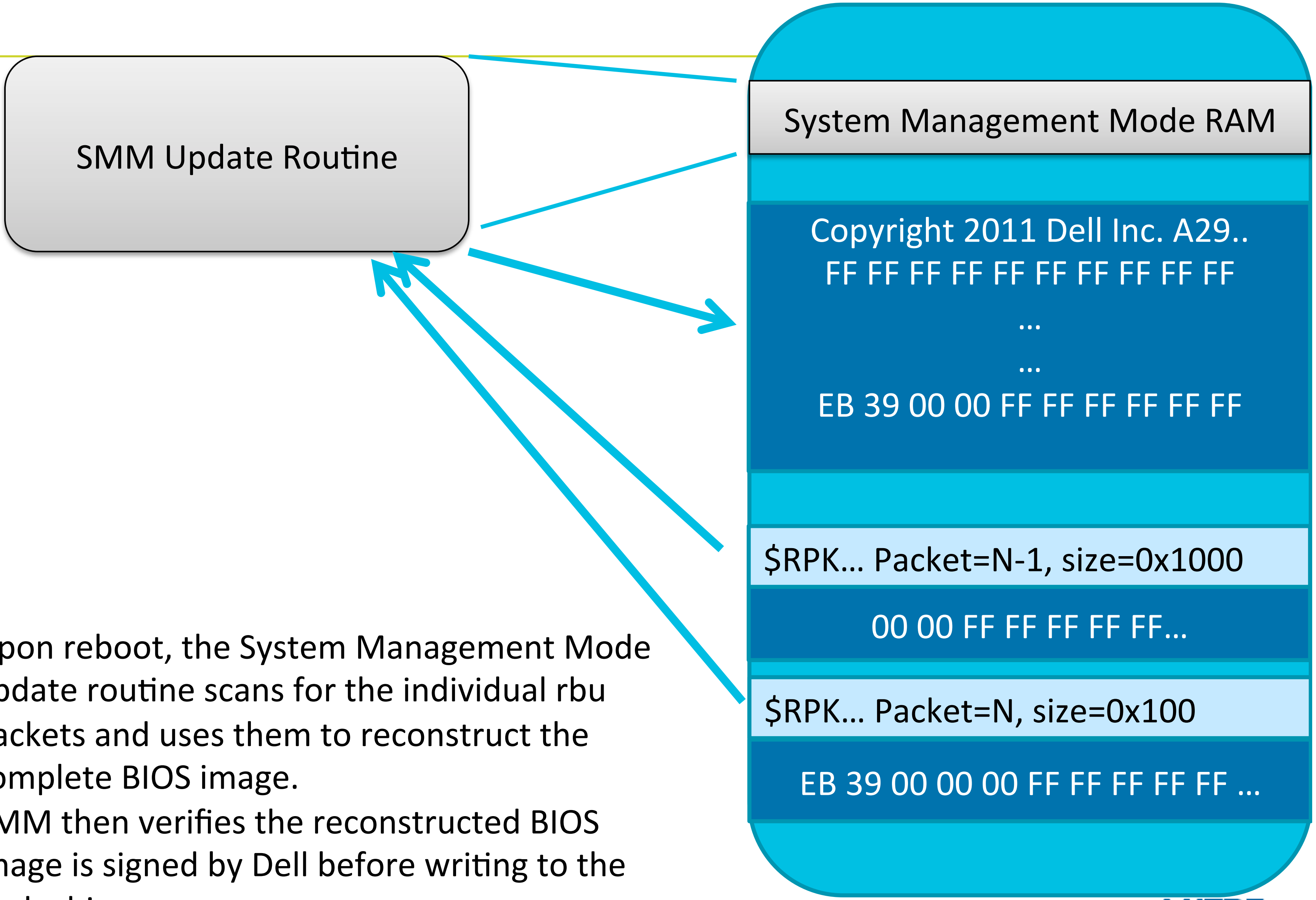
1. **Firmware update binary (“HDR”) is copied to kernel memory**
   - Default method is to packetize the HDR file into “rbu packets”
   - HDR contains more than just the BIOS update (Keyboard Controller, Management Engine, too)
2. **A bit in CMOS byte 0x78 is flipped**
3. **The system is rebooted**
4. **BIOS sees CMOS bit is flipped and triggers an SMI to execute the SMM BIOS Update routine**
5. **After the BIOS Update routine has occurred, the appropriate Intel flash protection mechanisms are set so that no more writes to the flash chip can occur.**

BIOS Update Routine (1 of 2)


OS Kernel Driver
$RPK… Packet=1, size=0x400
Copyright 2011 Dell Inc.. A29
$RPK… Packet=2, size=0x1000
00 00 FF FF FF FF FF…
$RPK… Packet=N, size=0x100
EB 39 00 00 00 FF FF FF FF …

- The Operating System packetizes the new BIOS image across the address space. Each packet has a 33 byte rbu_packet header that describes the contents and order of the BIOS image information the packet contains.
- A bit is then flipped in CMOS to indicate to the BIOS upon the next reboot that an update is pending.

BIOS Update Routine (2 of 2)

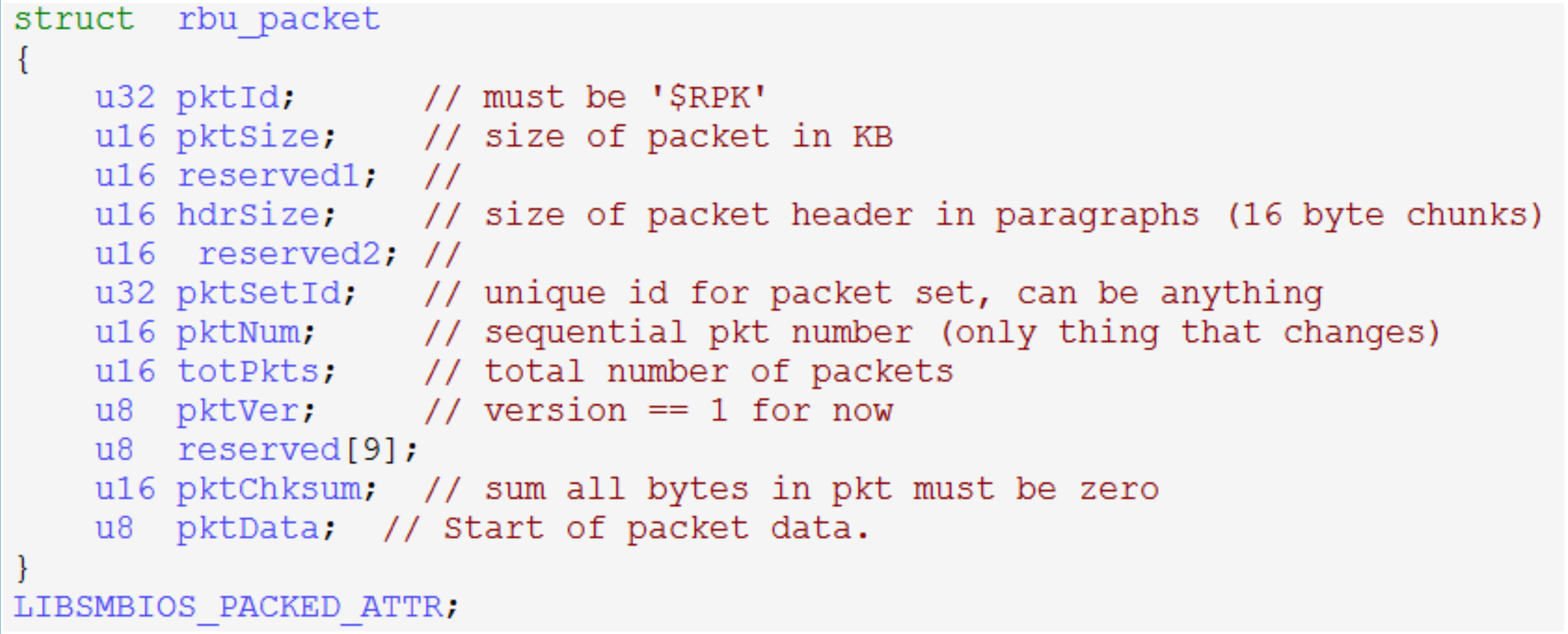

SMM Update Routine
System Management Mode RAM
Copyright 2011 Dell Inc. A29..
FF FF FF FF FF FF FF FF FF FF
…
…
EB 39 00 00 FF FF FF FF FF FF
$RPK… Packet=N-1, size=0x1000
00 00 FF FF FF FF FF…
$RPK… Packet=N, size=0x100
EB 39 00 00 00 FF FF FF FF FF …

- Upon reboot, the System Management Mode update routine scans for the individual rbu packets and uses them to reconstruct the complete BIOS image.
- SMM then verifies the reconstructed BIOS image is signed by Dell before writing to the flash chip.

Attacker Objective and Plan

- Reflash BIOS chip with arbitrary image despite signed BIOS enforcement.
- Method: find a memory corruption vulnerability in the parsing of the BIOS update information (RBU packets). This will allow us to seize control of SMM and reflash the BIOS chip at will.
- The memory corruption vulnerability must occur before the signature on the bios update image is checked.
- SMM parses the 33 byte rbu_packet header that describes metadata about the BIOS update image. This parsing occurs before the signature check.

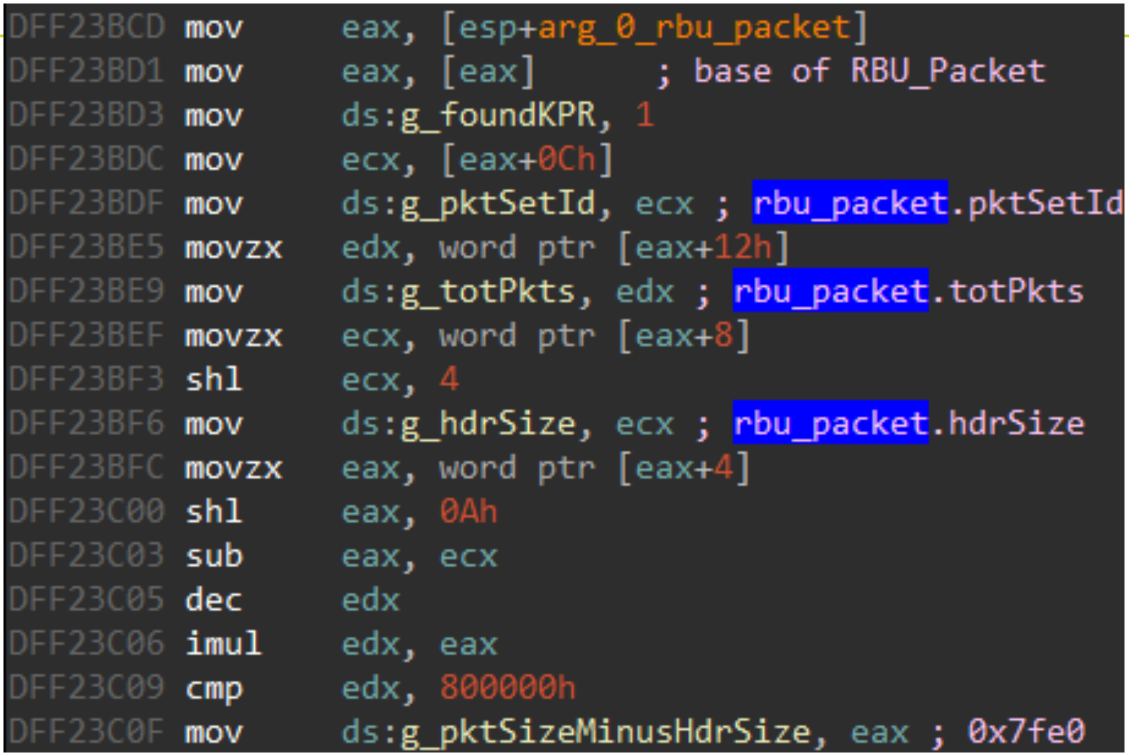
Attack Surface



```
struct  rbu_packet
{
    u32 pktId;      // must be '$RPK'
    u16 pktSize;    // size of packet in KB
    u16 reserved1;  //
    u16 hdrSize;    // size of packet header in paragraphs (16 byte chunks)
    u16  reserved2; //
    u32 pktSetId;   // unique id for packet set, can be anything
    u16 pktNum;     // sequential pkt number (only thing that changes)
    u16 totPkts;    // total number of packets
    u8  pktVer;     // version == 1 for now
    u8  reserved[9];
    u16 pktChksum;  // sum all bytes in pkt must be zero
    u8  pktData;  // Start of packet data.
}
LIBSMBIOS_PACKED_ATTR;
```

http://linux.dell.com/libsmbios/main/RbuLowLevel_8h-source.html

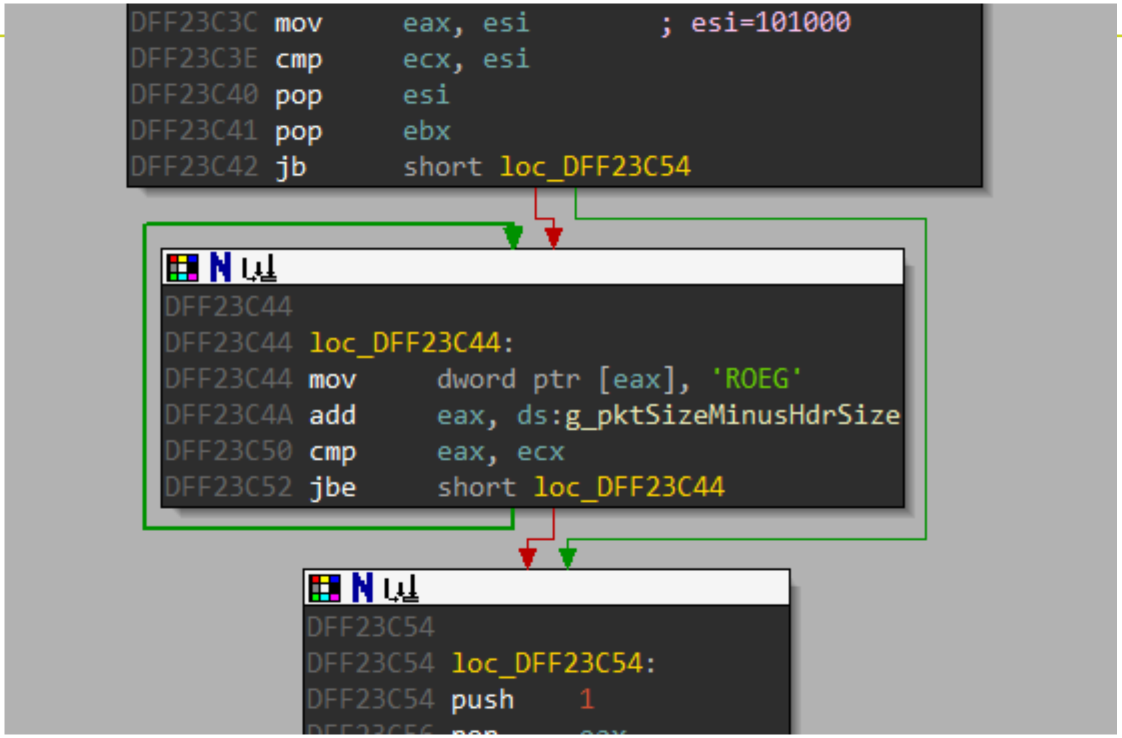
Packet Parsing



```
DFF23BCD mov     eax, [esp+arg_0_rbu_packet]
DFF23BD1 mov     eax, [eax]       ; base of RBU_Packet
DFF23BD3 mov     ds:g_foundKPR, 1
DFF23BDC mov     ecx, [eax+0Ch]
DFF23BDF mov     ds:g_pktSetId, ecx ; rbu_packet.pktSetId
DFF23BE5 movzx   edx, word ptr [eax+12h]
DFF23BE9 mov     ds:g_totPkts, edx ; rbu_packet.totPkts
DFF23BEF movzx   ecx, word ptr [eax+8]
DFF23BF3 shl     ecx, 4
DFF23BF6 mov     ds:g_hdrSize, ecx ; rbu_packet.hdrSize
DFF23BFC movzx   eax, word ptr [eax+4]
DFF23C00 shl     eax, 0Ah
DFF23C03 sub     eax, ecx
DFF23C05 dec     edx
DFF23C06 imul    edx, eax
DFF23C09 cmp     edx, 800000h
DFF23C0F mov     ds:g_pktSizeMinusHdrSize, eax ; 0x7fe0
```

- **SMM first locates the RBU packet by scanning for an ASCII signature upon page aligned boundaries.**
- **Once located, members of the RBU packet are stored in an SMM data area for use in later calculations…**

Curious GEOR?



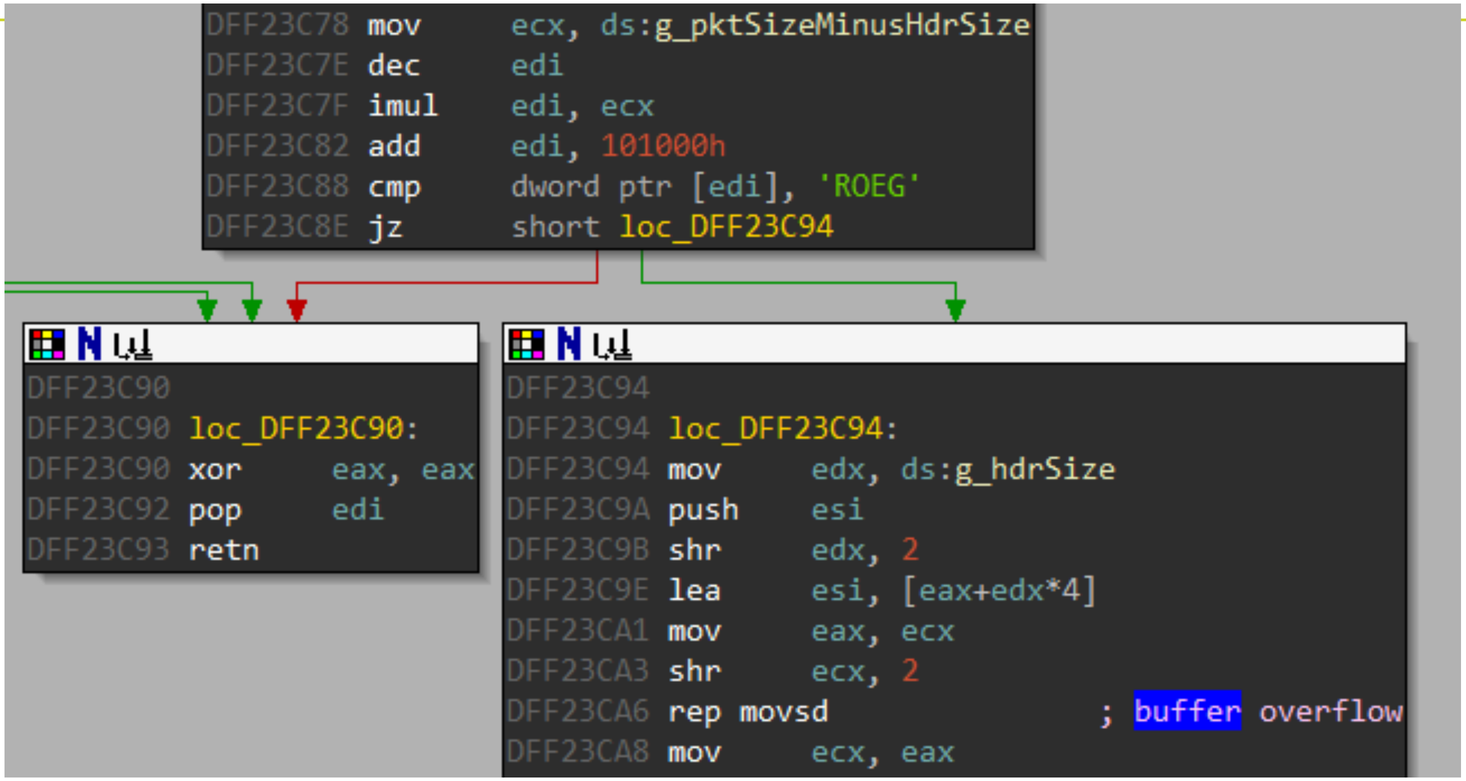
```
DFF23C3C mov     eax, esi          ; esi=101000
DFF23C3E cmp     ecx, esi
DFF23C40 pop     esi
DFF23C41 pop     ebx
DFF23C42 jb      short loc_DFF23C54
```

```
DFF23C44
DFF23C44 loc_DFF23C44:
DFF23C44 mov     dword ptr [eax], 'ROEG'
DFF23C4A add     eax, ds:g_pktSizeMinusHdrSize
DFF23C50 cmp     eax, ecx
DFF23C52 jbe     short loc_DFF23C44
```

```
DFF23C54
DFF23C54 loc_DFF23C54:
DFF23C54 push    1
```

- **When reconstructing the BIOS image from the rbu packets, SMM writes an initialization string "GEOR" to the destination memory space where the BIOS image is being reconstructed….**

RBU Packet Copied



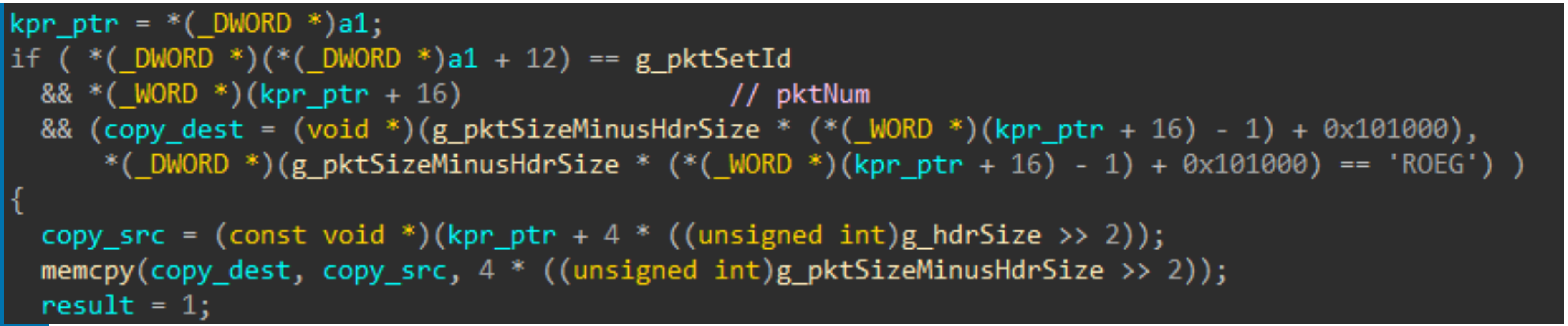
```
DFF23C78 mov     ecx, ds:g_pktSizeMinusHdrSize
DFF23C7E dec     edi
DFF23C7F imul    edi, ecx
DFF23C82 add     edi, 101000h
DFF23C88 cmp     dword ptr [edi], 'ROEG'
DFF23C8E jz      short loc_DFF23C94
```

```
DFF23C90
DFF23C90 loc_DFF23C90:
DFF23C90 xor     eax, eax
DFF23C92 pop     edi
DFF23C93 retn
```

```
DFF23C94
DFF23C94 loc_DFF23C94:
DFF23C94 mov     edx, ds:g_hdrSize
DFF23C9A push    esi
DFF23C9B shr     edx, 2
DFF23C9E lea     esi, [eax+edx*4]
DFF23CA1 mov     eax, ecx
DFF23CA3 shr     ecx, 2
DFF23CA6 rep movsd                     ; buffer overflow
DFF23CA8 mov     ecx, eax
```

- Eventually the portion of the BIOS image described by the RBU packet is copied to the reconstruction location in memory.
- Notice the size parameter (ecx) for the inline memcpy (rep movsd) is derived from attacker data (g_pktSizeMinusHdrSize).

RBU Packet Parsing Vulnerability



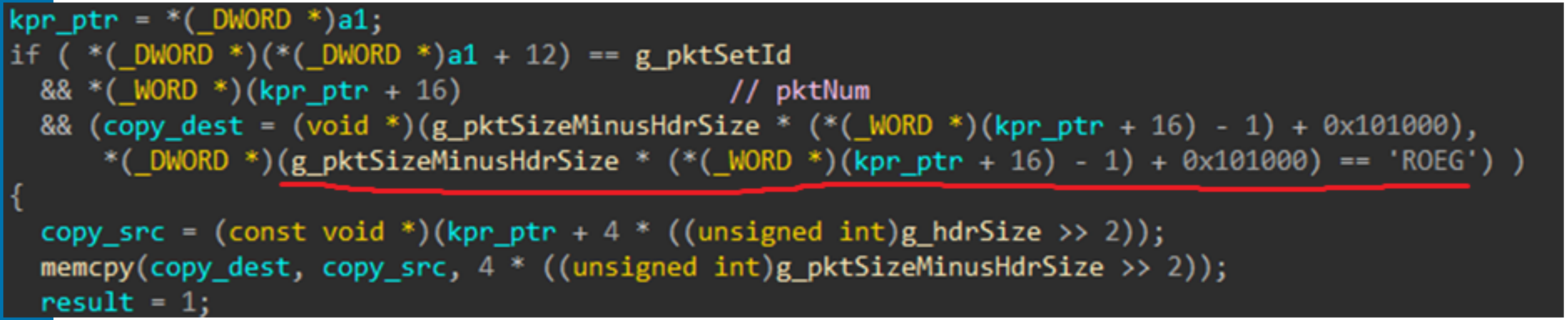
```
kpr_ptr = *(_DWORD *)a1;
if ( *(_DWORD *)(*(_DWORD *)a1 + 12) == g_pktSetId
  && *(_WORD *)(kpr_ptr + 16)                            // pktNum
  && (copy_dest = (void *)(g_pktSizeMinusHdrSize * (*(_WORD *)(kpr_ptr + 16) - 1) + 0x101000),
      *(_DWORD *)(g_pktSizeMinusHdrSize * (*(_WORD *)(kpr_ptr + 16) - 1) + 0x101000) == 'ROEG') )
{
  copy_src = (const void *)(kpr_ptr + 4 * ((unsigned int)g_hdrSize >> 2));
  memcpy(copy_dest, copy_src, 4 * ((unsigned int)g_pktSizeMinusHdrSize >> 2));
  result = 1;
```

- **In fact, the copy destination and copy source are also both derived from attacker data read in from the current rbu_packet.**
- **This is an exploitable buffer overflow.**

Lack of Mitigations

- **System Management Mode is missing all of the traditional exploit mitigations you would expect to find in modern applications.**
- **No ASLR, NX, stack canaries, and so on….**
- **This means we can pursue any target with our overwrite, such as the return address for the rbu packet copying function…**

Exploiting the Vulnerability



```
kpr_ptr = *(_DWORD *)a1;
if ( *(_DWORD *)(*(_DWORD *)a1 + 12) == g_pktSetId
  && *(_WORD *)(kpr_ptr + 16)                           // pktNum
  && (copy_dest = (void *)(g_pktSizeMinusHdrSize * (*(_WORD *)(kpr_ptr + 16) - 1) + 0x101000),
      *(_DWORD *)(g_pktSizeMinusHdrSize * (*(_WORD *)(kpr_ptr + 16) - 1) + 0x101000) == 'ROEG') )
{
  copy_src = (const void *)(kpr_ptr + 4 * ((unsigned int)g_hdrSize >> 2));
  memcpy(copy_dest, copy_src, 4 * ((unsigned int)g_pktSizeMinusHdrSize >> 2));
  result = 1;
```

- **There are actually a number of constraints on the RBU packet data that make exploiting this buffer overflow tricky.**

Constraints Overview

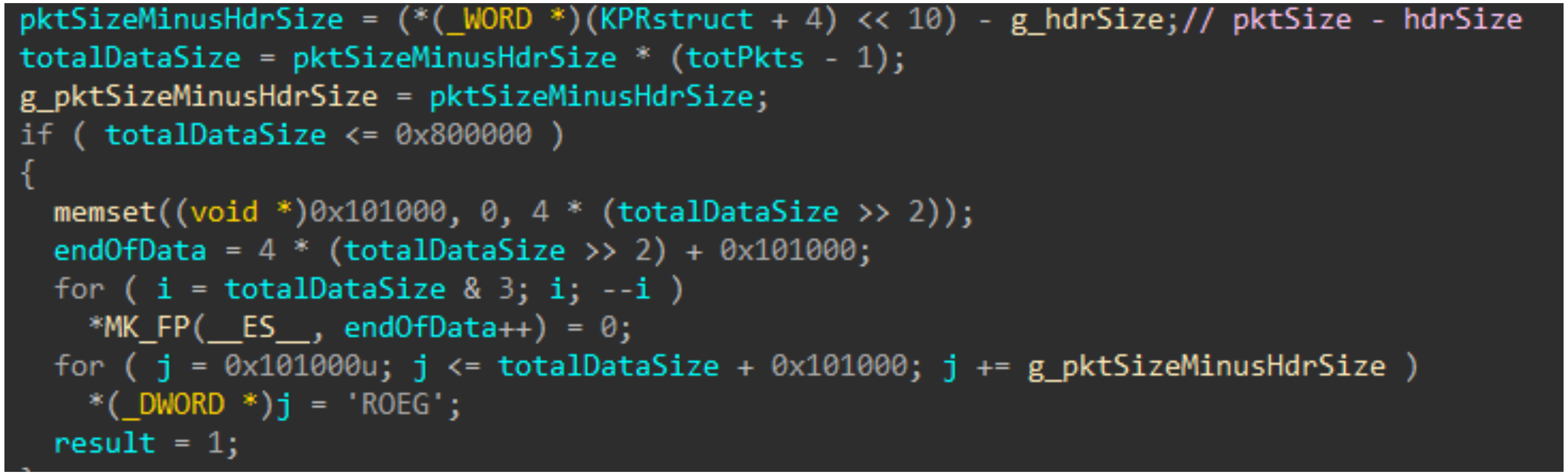
- **Our copy destination must point to an area pre-initialized with the "GEOR" string.**
- **Copy_dest must be lower in memory than the return address.**
- **We can't overwrite too much lest we die in the inline memcpy and never return.**
- **Copy source must be positioned such that attacker controlled data in the address space ends up overwriting the saved return address.**
- **Others….**

More Problems

```
copy_dest =  ((rbu_packet.pktSize << 10) - (rbu_packet.hdrSize << 4)) * ((rbu_packet.pktNum-1) + 0x101000);
copy_src = rbu_packet.pktSize * (rbu_packet.hdrSize << 2);
copy_size = (rbu_packet.pktSize << 10) - (rbu_packet.hdrSize << 4)
```

- **The source, destination and size operands are all derived from the same rbu_packet members.**
- **Changing one operand, changes the others.**
- **All of the constraints previously mentioned must be satisfied.**
- **Exploitation of this vulnerability can be modeled as a constraints solving problem.**

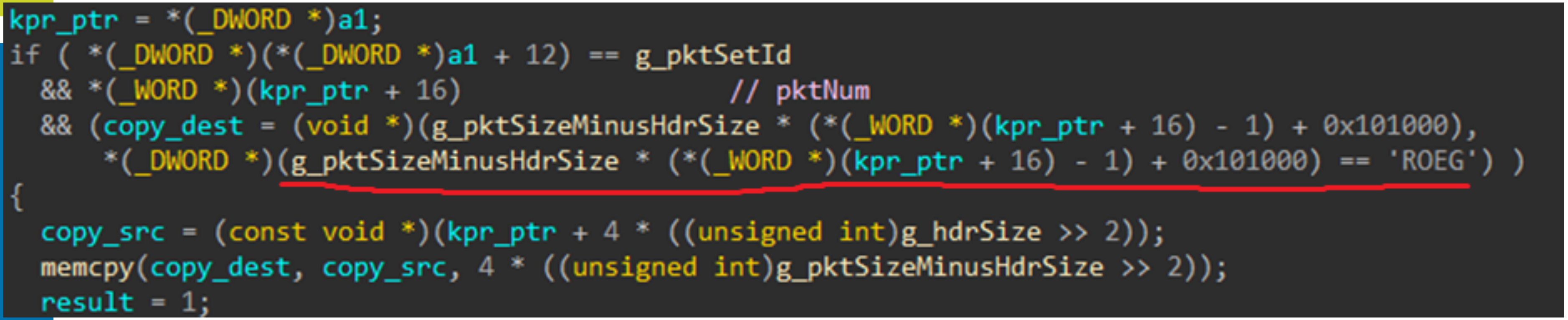
Constraints Corollary



```
pktSizeMinusHdrSize = (*(_WORD *)(KPRstruct + 4) << 10) - g_hdrSize;// pktSize - hdrSize
totalDataSize = pktSizeMinusHdrSize * (totPkts - 1);
g_pktSizeMinusHdrSize = pktSizeMinusHdrSize;
if ( totalDataSize <= 0x800000 )
{
  memset((void *)0x101000, 0, 4 * (totalDataSize >> 2));
  endOfData = 4 * (totalDataSize >> 2) + 0x101000;
  for ( i = totalDataSize & 3; i; --i )
    *MK_FP(__ES__, endOfData++) = 0;
  for ( j = 0x101000u; j <= totalDataSize + 0x101000; j += g_pktSizeMinusHdrSize )
    *(_DWORD *)j = 'ROEG';
  result = 1;
```

- An initialization routine populates the “GEOR” string at the expected copy dest location under “normal” circumstances.
- In order to pass the totalDataSize sanity check, we set totPkts to 1, forcing totalDataSize to 0.
- This means the expected “GEOR” string won’t naturally occur in the address space, and we will have to inject it somehow to satisfy the *copy_dest = “GEOR” constraint.

Faux GEOR

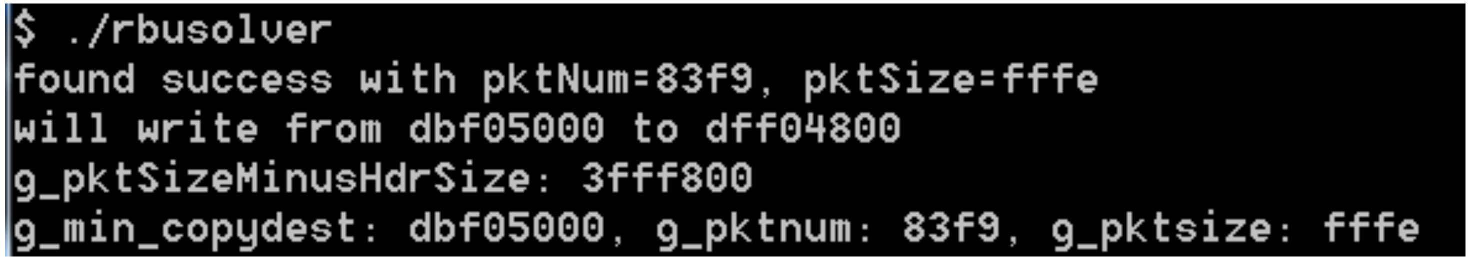


```
kpr_ptr = *(_DWORD *)a1;
if ( *(_DWORD *)(*(_DWORD *)a1 + 12) == g_pktSetId
  && *(_WORD *)(kpr_ptr + 16)                        // pktNum
  && (copy_dest = (void *)(g_pktSizeMinusHdrSize * (*(_WORD *)(kpr_ptr + 16) - 1) + 0x101000),
      *(_DWORD *)(g_pktSizeMinusHdrSize * (*(_WORD *)(kpr_ptr + 16) - 1) + 0x101000) == 'ROEG') )
{
  copy_src = (const void *)(kpr_ptr + 4 * ((unsigned int)g_hdrSize >> 2));
  memcpy(copy_dest, copy_src, 4 * ((unsigned int)g_pktSizeMinusHdrSize >> 2));
  result = 1;
```

- **The vulnerable memcpy will only execute if the copy destination points to a location containing this GEOR string.**
- **We use a Windows kernel driver that performs memory mapped i/o to write the GEOR string as high up in memory as possible, to allow us to force copy_dest to be within striking distance of the return address we want to overwrite.**
- **Like the BIOS update process, we are abusing the fact RAM remains intact during a soft reboot so the GEOR strings we wrote will remain in the address space.**

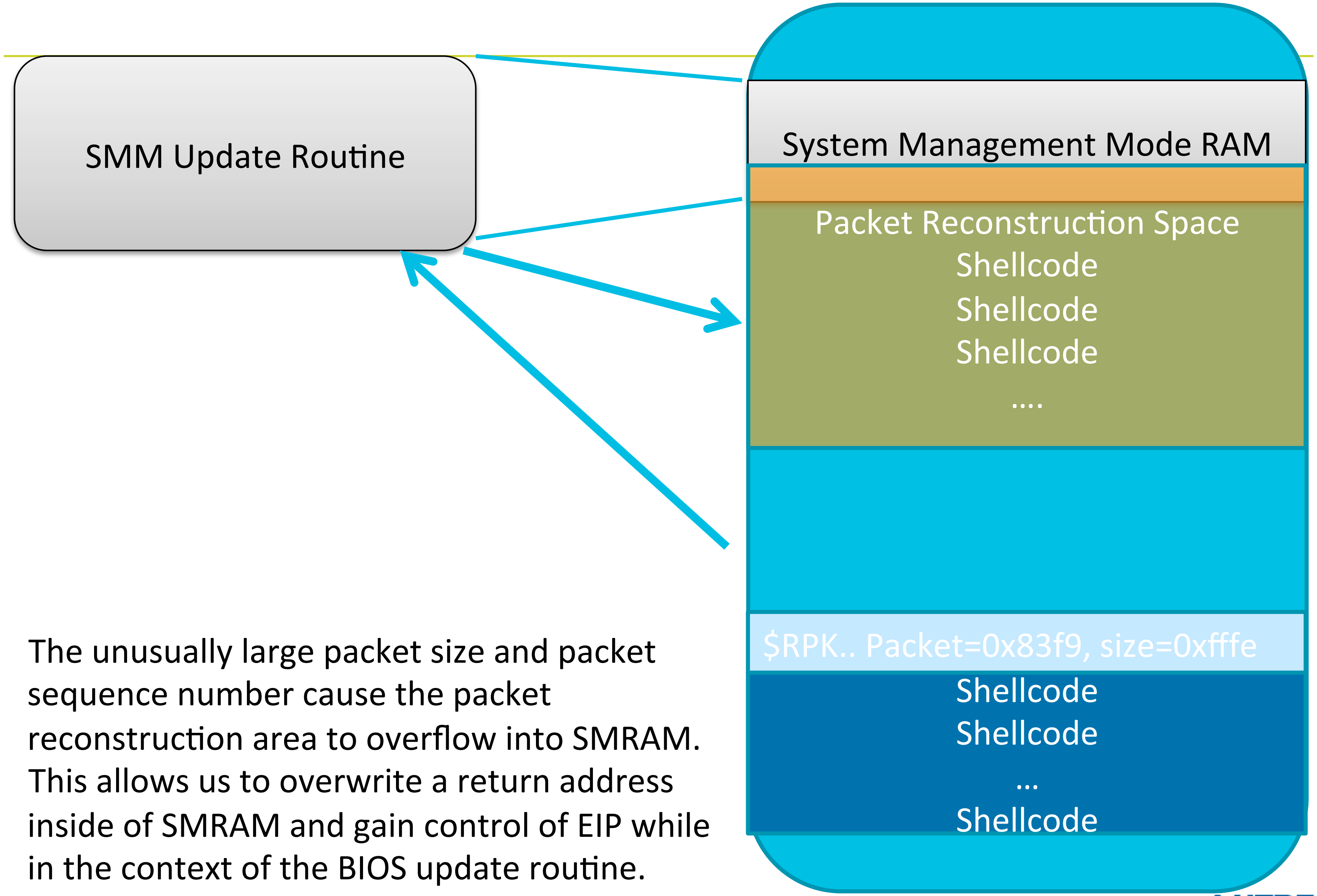
RBU Packet Solution

- With all those constraints in mind, we brute force an rbu_packet configuration that allows us to pass the sanity checks and overwrite the return address gracefully.



```
$ ./rbusolver
found success with pktNum=83f9, pktSize=fffe
will write from dbf05000 to dff04800
g_pktSizeMinusHdrSize: 3fff800
g_min_copydest: dbf05000, g_pktnum: 83f9, g_pktsize: fffe
```

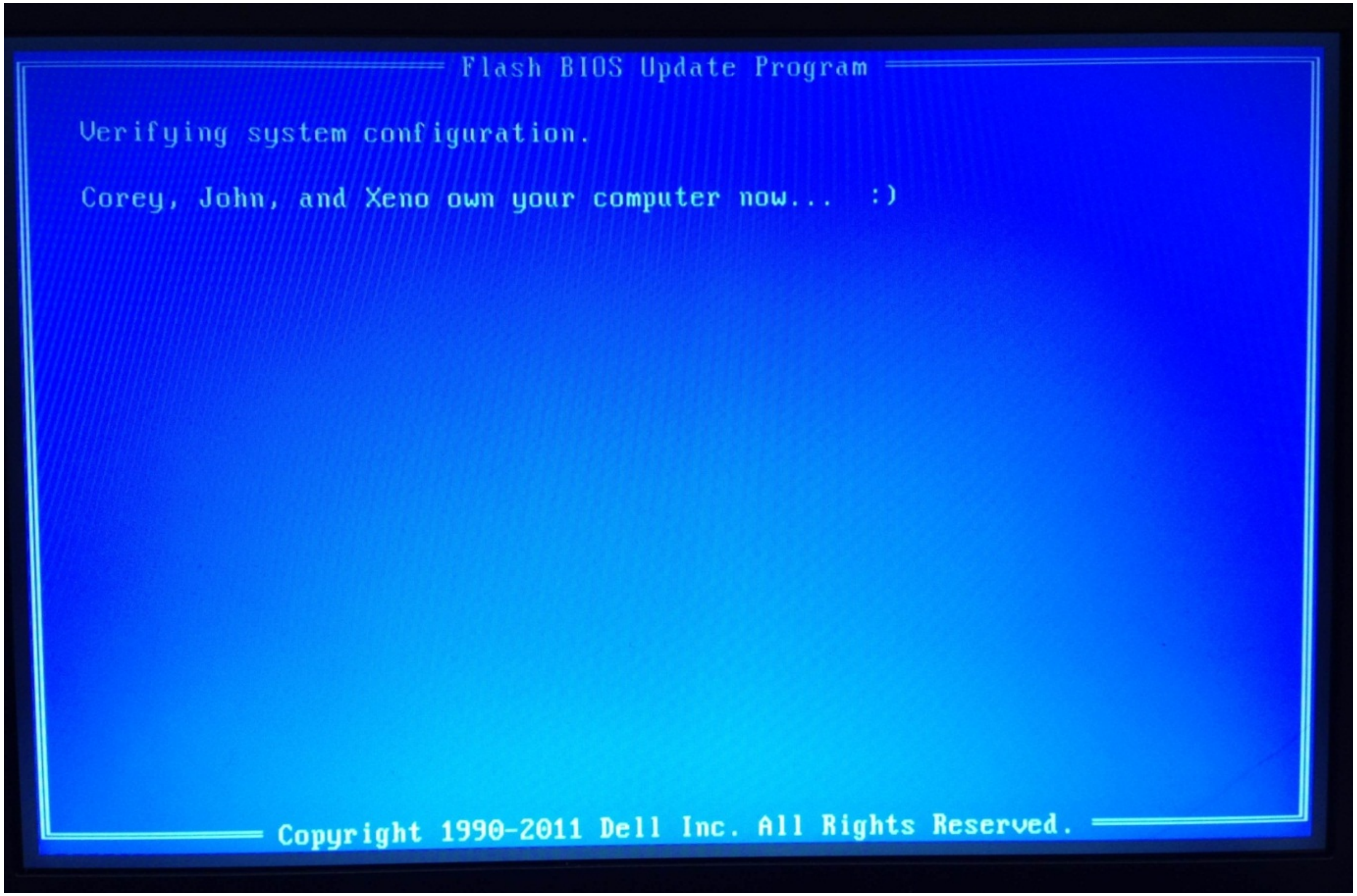
Malicious BIOS Update


SMM Update Routine
System Management Mode RAM
Packet Reconstruction Space
Shellcode
Shellcode
Shellcode
….
$RPK.. Packet=0x83f9, size=0xfffe
Shellcode
Shellcode
…
Shellcode

- The unusually large packet size and packet sequence number cause the packet reconstruction area to overflow into SMRAM.
- This allows us to overwrite a return address inside of SMRAM and gain control of EIP while in the context of the BIOS update routine.

PoC Demonstration Video

http://youtu.be/V_ea21CrOPM



```
Flash BIOS Update Program

Verifying system configuration.

Corey, John, and Xeno own your computer now...   :)

Copyright 1990-2011 Dell Inc. All Rights Reserved.
```

Vulnerability Conclusion

- **The vulnerability allows an attacker to take control of the BIOS update process and reflash the BIOS with an arbitrary image despite the presence of signed bios enforcement.**
- **CVE-2013-3582**
- **We suspect other firmware update routines also contain vulnerabilities because:**
  - They were probably developed before signed BIOS enforcement was even a consideration.
  - It is difficult to locate and reverse engineer the update code due to the proprietary nature of BIOS images, thus these routines have likely seen little (if any) peer review.
- **Locating and exploiting a vulnerability in the Dell BIOS update routine was quite difficult, perhaps that is an easier way…**

MITRE

Attacking the Intel Protections

- **As a reminder, the BIOS_CNTL and Protected Range/FLOCKDN registers are the primary protections against arbitrary flash writes.**
- **Interestingly, it seems as though most OEM's opt to rely entirely on the BIOS_CNTL register for flash protection.**
  - Of the 5197 systems that implemented signed BIOS enforcement in our enterprise environment, 4779 relied exclusively on BIOS_CNTL for protection!
  - Approximately 92% of the systems we surveyed don't configure Protected Range registers!
- **This entangles the security of SMRAM with the security of the flash chip in a dangerous way.**

An Old Bug Revisited

- **In 2009 Invisible Things Lab and Duflot et al. identified an attack that abused Intel architecture caching features to execute arbitrary code in the context of System Management Mode (SMM)[1] [2].**
- **The ITL/Duflot cache poisoning attack was originally thought to be a temporary attack on System Management RAM (SMRAM); any attacker code injected into SMRAM would be flushed by a platform reset.**
- **However, on some systems the cache poisoning attack can lead to an arbitrary reflash of the BIOS chip.**
  - Because the BIOS is responsible for instantiating SMRAM, this would allow the attacker permanent residence in SMM.

[1] "Attacking SMM Memory via Intel Cache Poisoning" by Rafal Wojtczuk and Joanna Rutkowska

[2] "Getting into SMRAM: SMM Reloaded" by L. Duflot, O. Levillain, B. Morin and O. Grumelard.

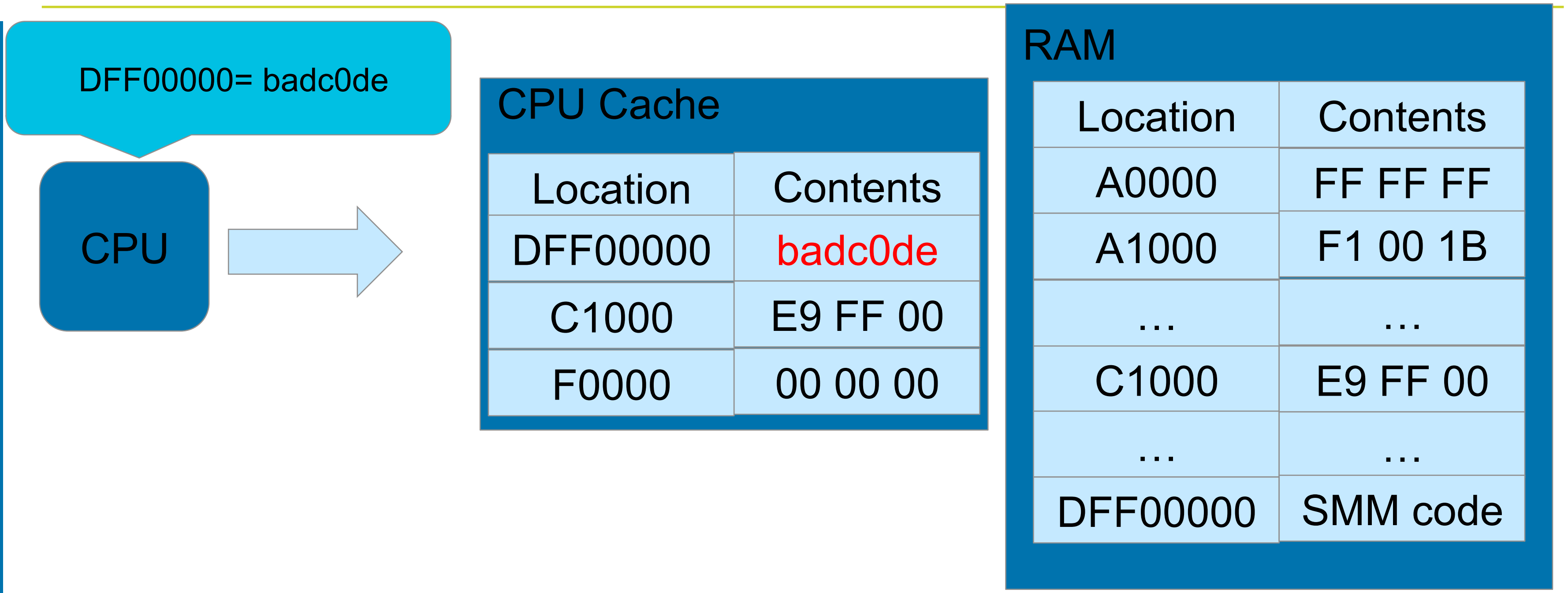
Cache Poisoning Attack Overview (1 of 2)

- **SMRAM is only writeable or readable by the CPU when it is executing in the context of SMM. Any attempt to ready SMRAM outside of SMM will be blocked by the Memory Controller Hub (MCH).**
- **The default caching policy for SMRAM is uncacheable; reads and writes happen directly to and from RAM, and are not stored in the cache.**

Cache Poisoning Attack Overview (2 of 2)

- **However, It is possible to program the MTRR's such that SMRAM is "Write Back" cacheable.**
- **An attacker can then pollute the cache entries corresponding to SMRAM by writing malicious code to the memory range associated with SMRAM.**
- **Although these changes will not actually be reflected in SMRAM, they will be reflected in the cache lines for the SMRAM memory locations.**
- **When the CPU next enters into SMM, it will fetch the SMM code from the SMRAM cache entries (instead of SMRAM actual).**
- **This results in arbitrary code execution in the context of SMM.**

Cache Attack (1 of 2)



CPU Cache

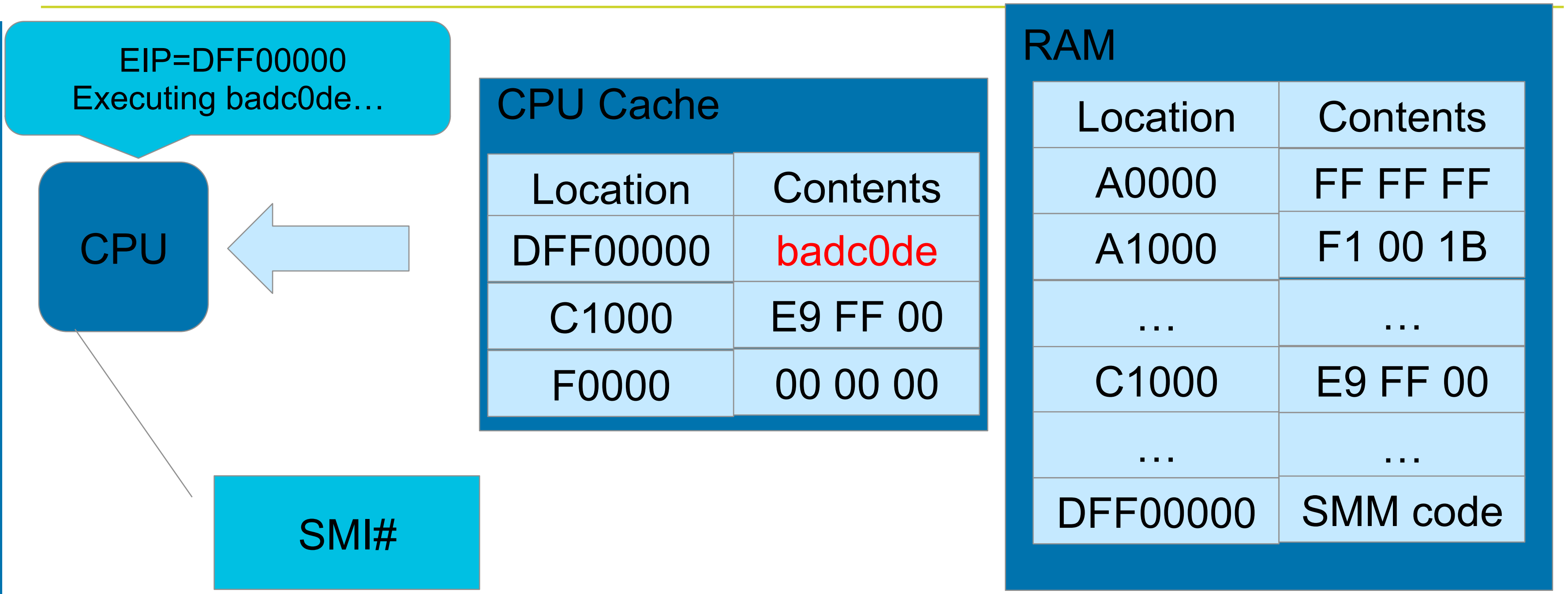
Location	Contents
DFF00000	badc0de
C1000	E9 FF 00
F0000	00 00 00

RAM

Location	Contents
A0000	FF FF FF
A1000	F1 00 1B
…	…
C1000	E9 FF 00
…	…
DFF00000	SMM code

- **In this case, SMRAM is based at DFF00000.**
- **First the attacker sets the SMRAM region to WriteBack cacheable using the MTRRs.**
- **Next the attacker pollutes the cache lines corresponding to SMRAM by attempting to write to SMRAM locations.**

Cache Attack (2 of 2)



CPU Cache

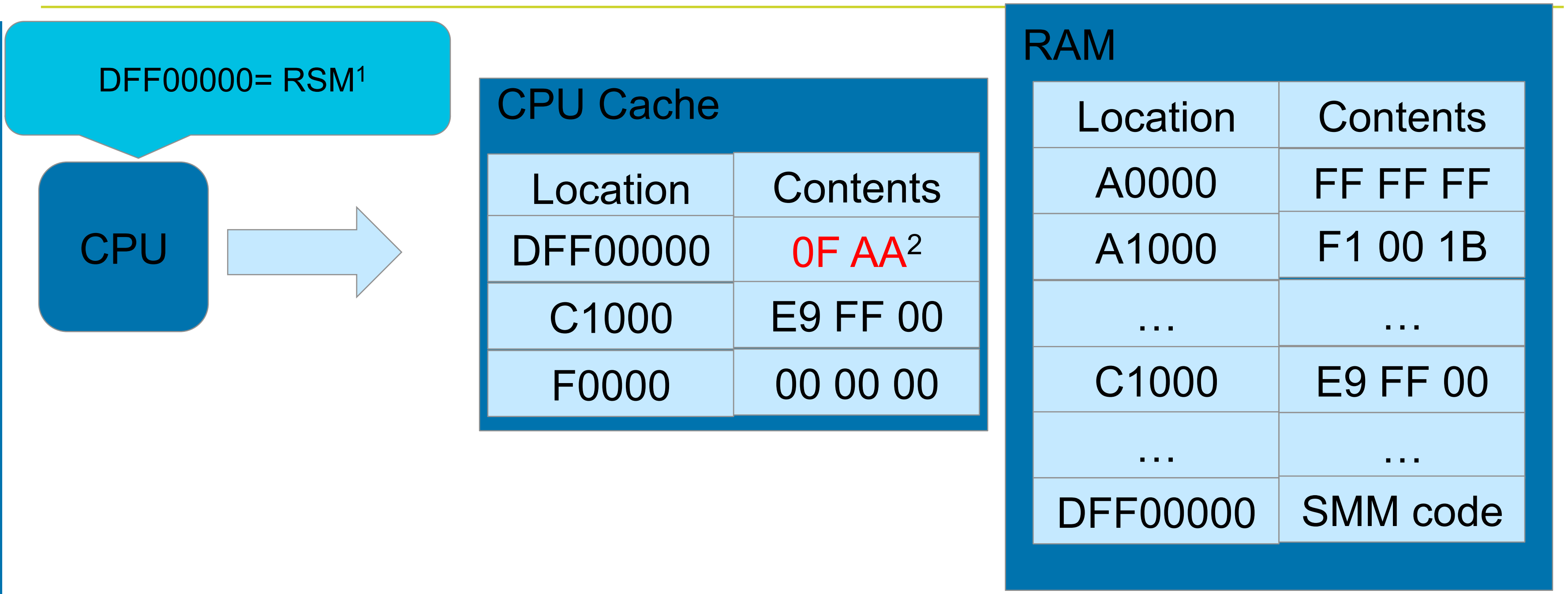
Location	Contents
DFF00000	badc0de
C1000	E9 FF 00
F0000	00 00 00

RAM

Location	Contents
A0000	FF FF FF
A1000	F1 00 1B
…	…
C1000	E9 FF 00
…	…
DFF00000	SMM code

- **Finally the attacker generates a System Management Interrupt (SMI#) to force the CPU to enter SMM and subsequently use the polluted cache entries.**
- **The attacker is now executing arbitrary code in the context of the super privileged SMM.**

BIOSWE Cache Attack



CPU Cache

Location	Contents
DFF00000	0F AA[2]
C1000	E9 FF 00
F0000	00 00 00

RAM

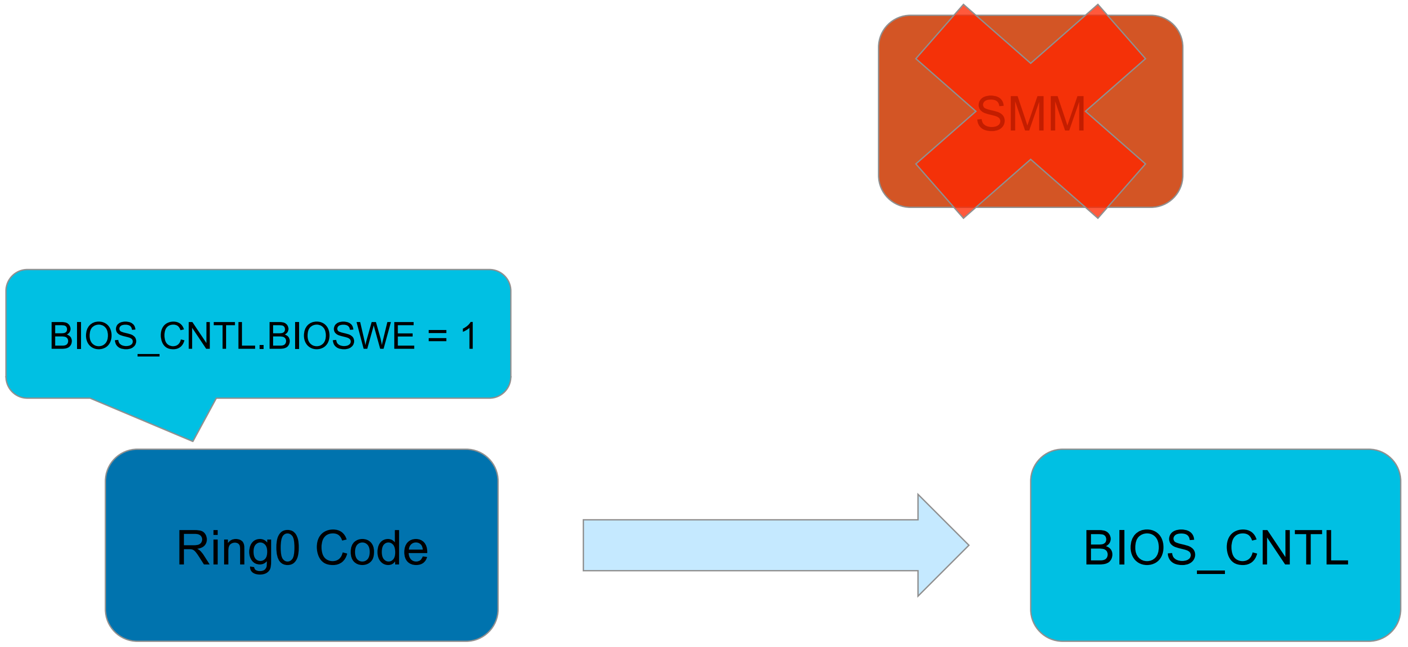
Location	Contents
A0000	FF FF FF
A1000	F1 00 1B
…	…
C1000	E9 FF 00
…	…
DFF00000	SMM code

- **We pollute the SMI entry point with an immediate return from SMM instruction.**
- **This will result in SMM failing to protect BIOSWE from being set.**

[1] RSM is the return from system management opcode

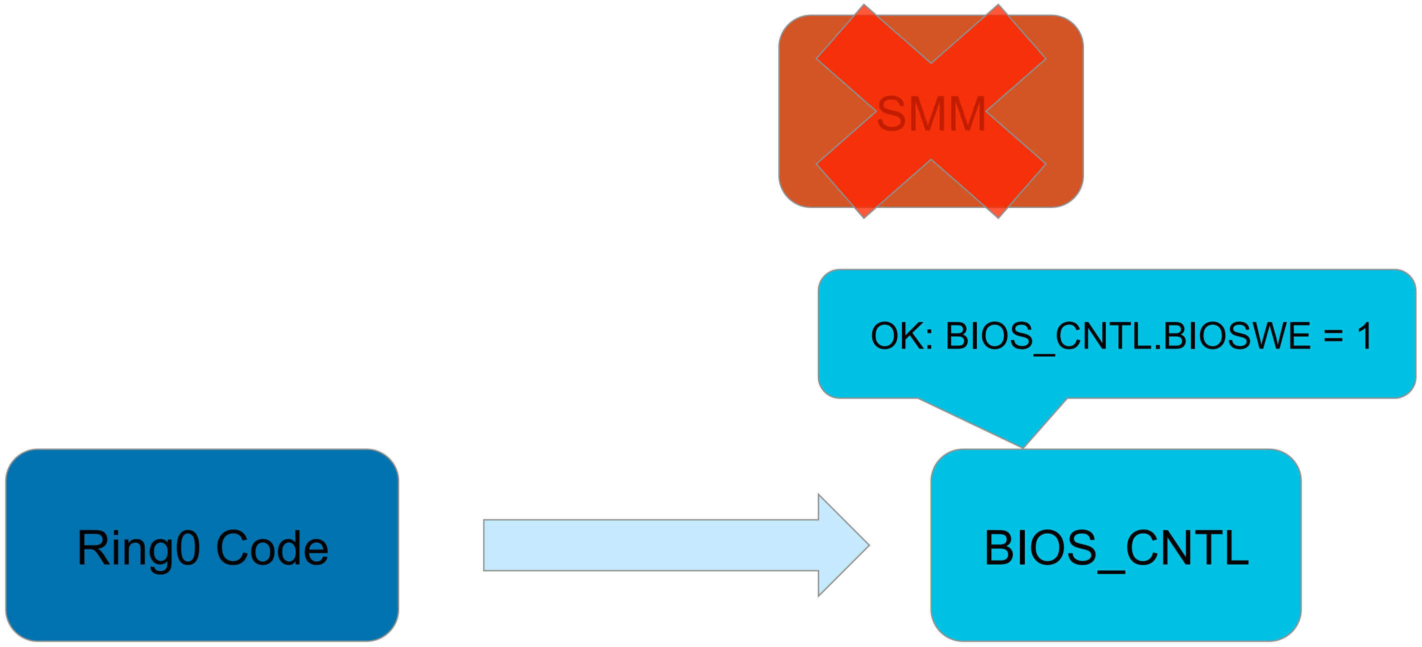
[2] 0F AA are the opcodes for the RSM instruction

Disabled BIOSWE protection (1 of 2)



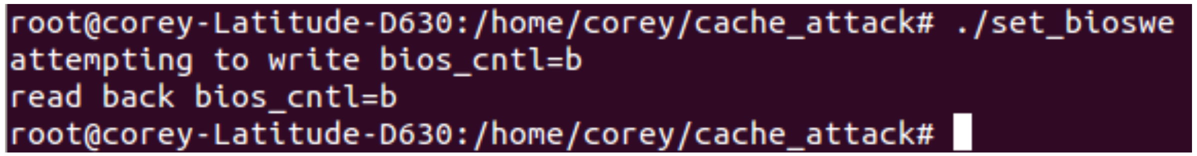
- **Again the attacker tries to set the BIOS Write Enable bit to 1 to allow him to overwrite the BIOS chip.**

Disabled BIOSWE protection (2 of 2)

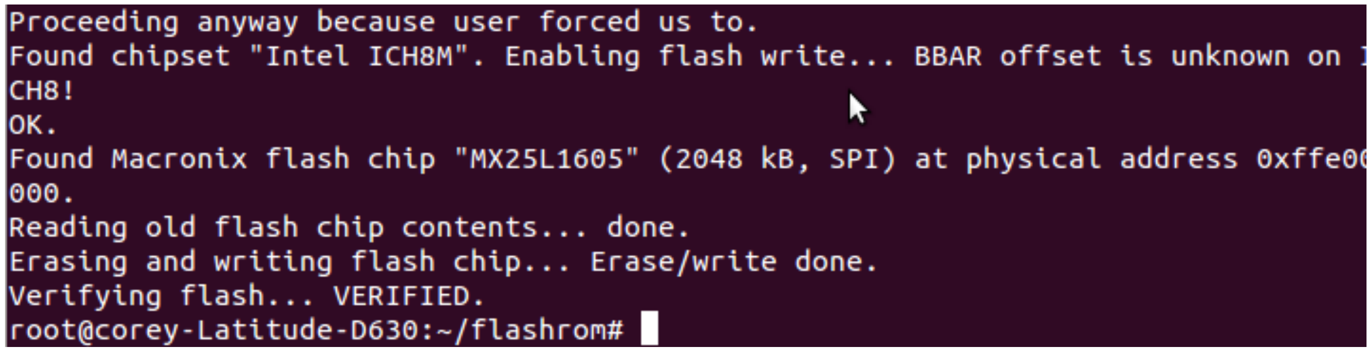


- **An SMI is generated on the write as before, but this time SMM just immediately returns instead of resetting the BIOSWE bit to 0.**

Disabled BIOSWE Protection Demo



```
root@corey-Latitude-D630:/home/corey/cache_attack# ./set_bioswe
attempting to write bios_cntl=b
read back bios_cntl=b
root@corey-Latitude-D630:/home/corey/cache_attack#
```



```
Proceeding anyway because user forced us to.
Found chipset "Intel ICH8M". Enabling flash write... BBAR offset is unknown on I
CH8!
OK.
Found Macronix flash chip "MX25L1605" (2048 kB, SPI) at physical address 0xffe0
000.
Reading old flash chip contents... done.
Erasing and writing flash chip... Erase/write done.
Verifying flash... VERIFIED.
root@corey-Latitude-D630:~/flashrom#
```

- We are able to set the BIOSWE bit (BIOS_CNTL = 0xB), and subsequently reflash the BIOS chip with an arbitrary image.
- This bypasses the signed firmware update requirement which is supposed to prevent arbitrary flash overwrites.

Poison Reflash Bug Conclusion

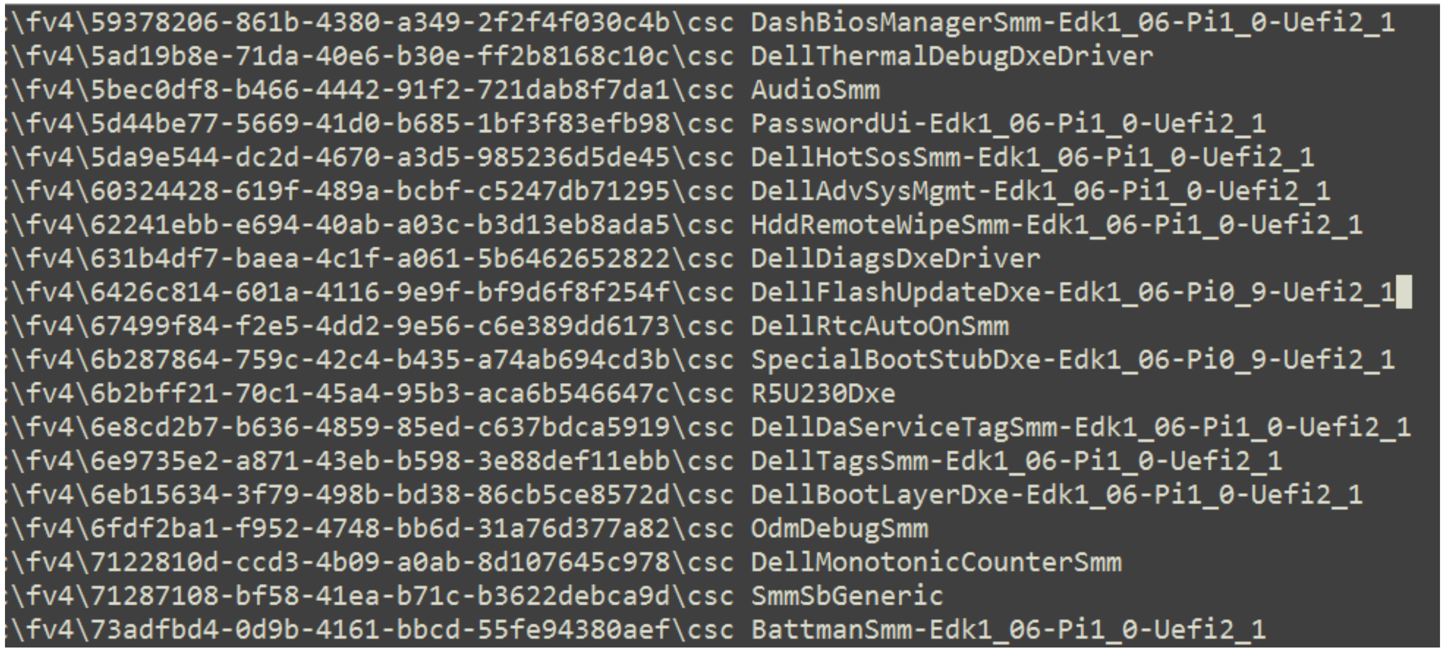
- **Currently reported to CERT as VU#255726.**
- **This bug has been largely fixed on newer systems by the introduction of “SMM Range Registers” which when programmed correctly prevent the SMM Cache Poisoning Attack.**
- **Important takeaway:**
  - Due to many OEM’s sole reliance on BIOS_CNTL protection of the flash chip, it follows that any vulnerabilities that allow you to modify SMRAM can be leveraged to reflash the BIOS.

Unified Extensible Firmware Interface

- **Does UEFI solve these problems?**
  - No. The underlying Intel flash protection mechanisms are the same. Many vendors are still relying only on BIOS_CNTL register for protection, and hence are vulnerable to any SMRAM compromises that may occur.
  - Vendor's are still implementing their own custom firmware update routines.
  - There are even UEFI systems shipping with completely unlocked flash chips…

- **In some ways, UEFI makes things easier for an attacker…**

UEFI Reversing is Easier

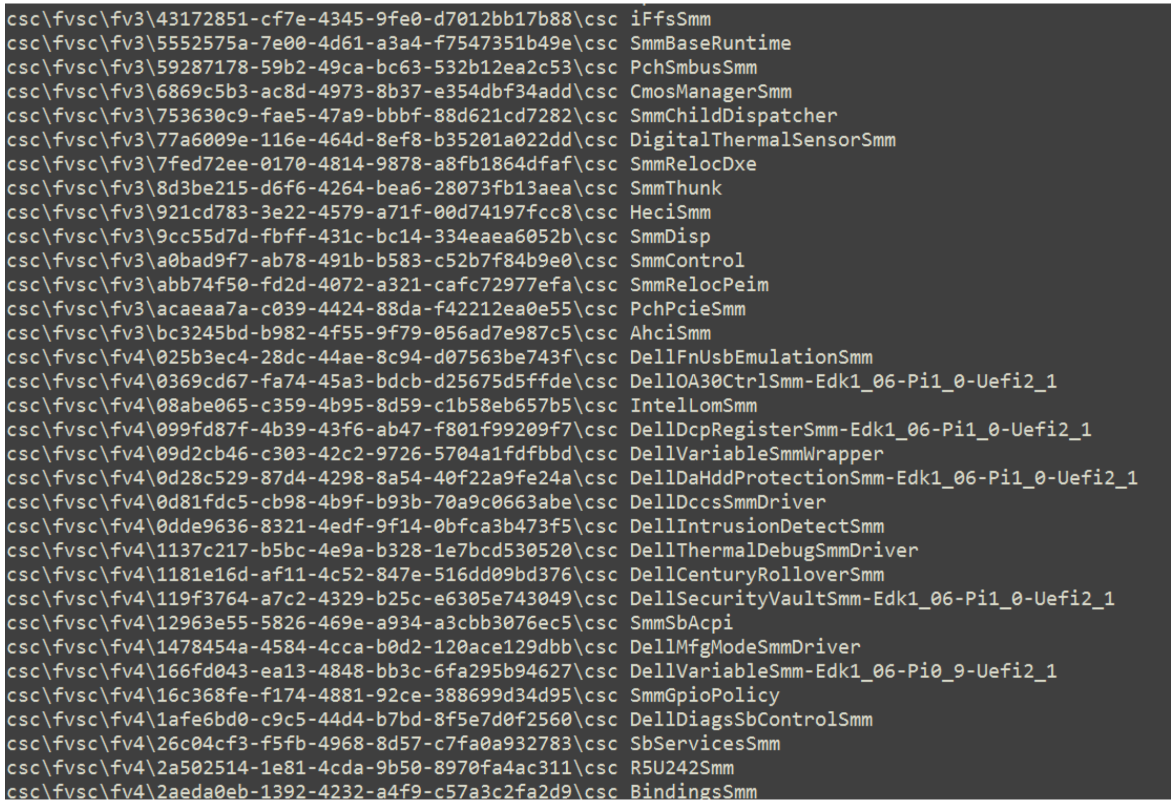


```
:\fv4\59378206-861b-4380-a349-2f2f4f030c4b\csc DashBiosManagerSmm-Edk1_06-Pi1_0-Uefi2_1
:\fv4\5ad19b8e-71da-40e6-b30e-ff2b8168c10c\csc DellThermalDebugDxeDriver
:\fv4\5bec0df8-b466-4442-91f2-721dab8f7da1\csc AudioSmm
:\fv4\5d44be77-5669-41d0-b685-1bf3f83efb98\csc PasswordUi-Edk1_06-Pi1_0-Uefi2_1
:\fv4\5da9e544-dc2d-4670-a3d5-985236d5de45\csc DellHotSosSmm-Edk1_06-Pi1_0-Uefi2_1
:\fv4\60324428-619f-489a-bcbf-c5247db71295\csc DellAdvSysMgmt-Edk1_06-Pi1_0-Uefi2_1
:\fv4\62241ebb-e694-40ab-a03c-b3d13eb8ada5\csc HddRemoteWipeSmm-Edk1_06-Pi1_0-Uefi2_1
:\fv4\631b4df7-baea-4c1f-a061-5b6462652822\csc DellDiagsDxeDriver
:\fv4\6426c814-601a-4116-9e9f-bf9d6f8f254f\csc DellFlashUpdateDxe-Edk1_06-Pi0_9-Uefi2_1
:\fv4\67499f84-f2e5-4dd2-9e56-c6e389dd6173\csc DellRtcAutoOnSmm
:\fv4\6b287864-759c-42c4-b435-a74ab694cd3b\csc SpecialBootStubDxe-Edk1_06-Pi0_9-Uefi2_1
:\fv4\6b2bff21-70c1-45a4-95b3-aca6b546647c\csc R5U230Dxe
:\fv4\6e8cd2b7-b636-4859-85ed-c637bdca5919\csc DellDaServiceTagSmm-Edk1_06-Pi1_0-Uefi2_1
:\fv4\6e9735e2-a871-43eb-b598-3e88def11ebb\csc DellTagsSmm-Edk1_06-Pi1_0-Uefi2_1
:\fv4\6eb15634-3f79-498b-bd38-86cb5ce8572d\csc DellBootLayerDxe-Edk1_06-Pi1_0-Uefi2_1
:\fv4\6fdf2ba1-f952-4748-bb6d-31a76d377a82\csc OdmDebugSmm
:\fv4\7122810d-ccd3-4b09-a0ab-8d107645c978\csc DellMonotonicCounterSmm
:\fv4\71287108-bf58-41ea-b71c-b3622debca9d\csc SmmSbGeneric
:\fv4\73adfbd4-0d9b-4161-bbcd-55fe94380aef\csc BattmanSmm-Edk1_06-Pi1_0-Uefi2_1
```

- **UEFI Firmware comes in a standard "firmware volume" which you can parse to quickly find relevant code.[1]**

[1] EFIPWN: https://github.com/G33KatWork/EFIPWN

Disturbing Trend



```
csc\fvsc\fv3\43172851-cf7e-4345-9fe0-d7012bb17b88\csc iFfsSmm
csc\fvsc\fv3\5552575a-7e00-4d61-a3a4-f7547351b49e\csc SmmBaseRuntime
csc\fvsc\fv3\59287178-59b2-49ca-bc63-532b12ea2c53\csc PchSmbusSmm
csc\fvsc\fv3\6869c5b3-ac8d-4973-8b37-e354dbf34add\csc CmosManagerSmm
csc\fvsc\fv3\753630c9-fae5-47a9-bbbf-88d621cd7282\csc SmmChildDispatcher
csc\fvsc\fv3\77a6009e-116e-464d-8ef8-b35201a022dd\csc DigitalThermalSensorSmm
csc\fvsc\fv3\7fed72ee-0170-4814-9878-a8fb1864dfaf\csc SmmRelocDxe
csc\fvsc\fv3\8d3be215-d6f6-4264-bea6-28073fb13aea\csc SmmThunk
csc\fvsc\fv3\921cd783-3e22-4579-a71f-00d74197fcc8\csc HeciSmm
csc\fvsc\fv3\9cc55d7d-fbff-431c-bc14-334eaea6052b\csc SmmDisp
csc\fvsc\fv3\a0bad9f7-ab78-491b-b583-c52b7f84b9e0\csc SmmControl
csc\fvsc\fv3\abb74f50-fd2d-4072-a321-cafc72977efa\csc SmmRelocPeim
csc\fvsc\fv3\acaeaa7a-c039-4424-88da-f42212ea0e55\csc PchPcieSmm
csc\fvsc\fv3\bc3245bd-b982-4f55-9f79-056ad7e987c5\csc AhciSmm
csc\fvsc\fv4\025b3ec4-28dc-44ae-8c94-d07563be743f\csc DellFnUsbEmulationSmm
csc\fvsc\fv4\0369cd67-fa74-45a3-bdcb-d25675d5ffde\csc DellOA30CtrlSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\08abe065-c359-4b95-8d59-c1b58eb657b5\csc IntelLomSmm
csc\fvsc\fv4\099fd87f-4b39-43f6-ab47-f801f99209f7\csc DellDcpRegisterSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\09d2cb46-c303-42c2-9726-5704a1fdfbbd\csc DellVariableSmmWrapper
csc\fvsc\fv4\0d28c529-87d4-4298-8a54-40f22a9fe24a\csc DellDaHddProtectionSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\0d81fdc5-cb98-4b9f-b93b-70a9c0663abe\csc DellDccsSmmDriver
csc\fvsc\fv4\0dde9636-8321-4edf-9f14-0bfca3b473f5\csc DellIntrusionDetectSmm
csc\fvsc\fv4\1137c217-b5bc-4e9a-b328-1e7bcd530520\csc DellThermalDebugSmmDriver
csc\fvsc\fv4\1181e16d-af11-4c52-847e-516dd09bd376\csc DellCenturyRolloverSmm
csc\fvsc\fv4\119f3764-a7c2-4329-b25c-e6305e743049\csc DellSecurityVaultSmm-Edk1_06-Pi1_0-Uefi2_1
csc\fvsc\fv4\12963e55-5826-469e-a934-a3cbb3076ec5\csc SmmSbAcpi
csc\fvsc\fv4\1478454a-4584-4cca-b0d2-120ace129dbb\csc DellMfgModeSmmDriver
csc\fvsc\fv4\166fd043-ea13-4848-bb3c-6fa295b94627\csc DellVariableSmm-Edk1_06-Pi0_9-Uefi2_1
csc\fvsc\fv4\16c368fe-f174-4881-92ce-388699d34d95\csc SmmGpioPolicy
csc\fvsc\fv4\1afe6bd0-c9c5-44d4-b7bd-8f5e7d0f2560\csc DellDiagsSbControlSmm
csc\fvsc\fv4\26c04cf3-f5fb-4968-8d57-c7fa0a932783\csc SbServicesSmm
csc\fvsc\fv4\2a502514-1e81-4cda-9b50-8970fa4ac311\csc R5U242Smm
csc\fvsc\fv4\2aeda0eb-1392-4232-a4f9-c57a3c2fa2d9\csc BindingsSmm
```

- **Lots and lots of code is getting put into SMM.**
- **An exploitable bug in any of this may lead to a firmware reflash bug.**

Unpatched Vulnerability

- Demo
- Reported to CERT
- Effects multiple vendors
- VU #291102

Conclusion

- OEM’s should be using the Protected Range registers, but many of them are not.
- Sole reliance on BIOS_CNTL for flash protection entangles the security of SMM with the security of the flash chip.
- Vulnerabilities lurking in OEM firmware update routines can allow arbitrary reflashes of the system firmware.
- These issues are UEFI/BIOS agnostic.
- There are still new UEFI systems shipping with unlocked flash chips!

MITRE

Acknowledgements

- **Rafal Wojtczuk**
  - Breaking ground in this area
  - Helpful email discussion on topic
- **Rick Martinez**
  - Dell contact for BIOS related security issues
  - Great vendor for researchers to work with
- **Bruce Monroe**
  - Intel contact
  - Strong interest in general BIOS/UEFI/Intel security issues!
  - Currently coordinating the release of other issues