Code Breakers Journal

Vol. 2, No. 1 (2005)
http://www.CodeBreakers-Journal.com

Award BIOS Code Injection

Mappatutu Salihun Darmawan*

E-Mail: mamanzip@yahoo.com

* Corresponding Author

Received: 07. Mar. 2005, Accepted: 10. Mar. 2005, Published: 20. Mar. 2005

Abstract

The possiblity of code injection in firmware, specifically Award Bios is not well explored yet up to this date. This is due to lack of documentation and understanding about how the whole system works. Code injection in Award Bios is made possible by the existence of the so called "POST Jump Table". This paper explores this possibility and provides a proof of concept for this idea.

Keywords: *Reverse Code Engineering; BIOS; Code Injection*

1. Introduction

Based on the Award Bios Reverse Engineering paper by Darmawan Mappatutu Salihun [1], it's very clear that there's a possibility for code injection in Award Bios. This opportunity is provided by the existence of the so called "POST jump table". We can do a code injection by patching this so called "POST jump table" to include a jump into our custom routine. There are several reasons why this method is a better approach to Award Bios hacking :

- In theory, this approach is much more safe compared to other bios hacking methods. In this technique, we are incorporating new functionality into the system bios (original.tmp in Award Bios) without replacing any functionality in the current system bios. In other words, it's safe to do it.
- There are lot of places in the "POST jump table" that are safe to patch, since they are only jump to "dummy" procedures.
- Incorporating an additional routine to bios, specifically Award Bios, as an isa option rom is not always guaranteed to be flawless. We have experienced a circumstance where this kind of approach is just unacceptable. When we implant an experimental expansion-rom based OS-kernel in a hacked Adaptec PCI SCSI controller card, the old isa option rom based bios patch causes the system to hang if the PCI slots are heavily populated. This is really unacceptable.
- Perhaps, one can add "cool" procedures to POST as cosmetics.

The following is the detail of the testbed used for this radical bios modification :

Processor	**:** Intel Celeron 300A, overclocked to 518 MHz by using ABIT Slotket II adapter
Mainboard	**:** Iwill VD133 (slot 1) with VIA693A northbridge and VIA596B southbridge
Videocard	**:** PowerColor Nvidia Riva TNT2 M64 32MB
RAM	**:** 256MB SDRAM with unknown chip
Soundcard	**:** Addonics Yamaha YMF724
Network Card	**:** Realtek RTL8139C
Harddrive	**:** Maxtor 20GB 5400RPM
CDROM	**:** Teac 40X
Monitor	**:** Samsung SyncMaster 551v (15')
Operating System	**:** - Windows 2000, used to run the modification tools - Real-mode DOS, used to flash the Bios

2. Tools Of The Trade

The tools needed to do the code injection as follows:

1. IDA Pro disassembler. We are using IDA Pro version 4.50. One can use their favourite interactive disassembler. We found IDA Pro is the most suitable for us. We need an interactive disassembler since the bios binary that we are going to disassemble is not a trivial code.
2. A good hex editor. We are using HexWorkshop ver. 3.02. The most beneficial feature of this hex editor is its capability to calculate checksums for the selected range of file that we open inside of it. We use this tool to edit the bios binary.
3. Nasm, the netwide assembler. Can be downloaded it at http://nasm.sourceforge.net. We use this to assemble the code that will be injected to the Bios.

4. A text editor, we use this to edit and write the injected x86 assembly language code. Anyway, notepad is enough.
5. Some bios modification tools i.e. :
    - CBROM, we are using version 2.08, 2.07 and 1.24. It's available at www.biosmods.com, in the download section
    - MODBIN, there are two types of modbin, modbin6 for Award Bios ver. 6 and modbin 4.50.xx for Award Bios ver. 4.5xPGNM. We need this tool to look at the bios components much more easily. It's available at www.biosmods.com, in the download section. This tool also used to ensure that the checksum of the modified bios is fine.
    - Awardbios editor version 1.0. Thanks to Mike Tedder a.k.a bpoint for providing us with this very nice tool. It is available at http://awdbedit.sourceforge.net/. We use this tool to replace the original system bios of our Award Bios (original.tmp) with a new one. Actually this can be accomplished using any LZH capable compressor such as LHA 2.55 along with a hexeditor. But, we haven't test the robustness of this method, and it's more easier to do it with Awardbios editor.
    - UNIFLASH or Awardflash. This is the tool we use to flash the modified Bios to the mainboard Bios chip. We won't explain how to use it. It's trivial, just read its manual. Awardflash can be obtained in many places on the web, including in the mainboard manufacturer website. Uniflash can be downloaded at http://www.uniflash.org. One can also use any windows based bios flashing tool that maybe available from the mainboard vendor.
6. Some chipset datasheets. This depends on the mainboard bios binary that you are going to dissect. Some datasheets available at www.rom.by in the PDF-s section. I'm dissecting a VIA693A-596B mainboard. We have all of the needed datasheets.

3. Prerequisite

There are some issues that won't be explained here and it's the reader task that should be carried out to comprehend this paper.

- The most important thing is you have to be able to program and understand x86 assembly language. We are using masm and nasm syntax throughout this article. Both of them are variant of Intel syntax.
- How to program in x86 real mode. The POST (Power On-Self Test) routine in the Bios is executed in real mode. So, if we want to inject code there, it should be executing in real mode.
- You have to be able to comprehend datasheets of mainboard chipsets, i.e. the northbridge and southbridge. This is not a must. But, if you intend to know how the sample "injected routine" works, you have to acquire this knowledge. In this article we will present an example routine that reprogram the mainboard chipset to tweak it to achieve better performance in it's memory subsystem. Basically, this routine reprogram the memory controller of the northbridge. This routine is injected to POST through the POST jump table.
- How to flash the bios binary into your mainboard. This is a trivial thing to do.
- We strongly encourage you to do at least preliminary reverse engineering on Award Bios. This is very useful to comprehend the explanation here. To begin with, you can read the Award Bios Reverse Engineering paper by Darmawan Mappatutu Salihun [1]. After doing this, if your Bios is Award Bios or it's variant, it's very possible that you will find the "POST jump table" location in its system bios (original.tmp) part.

Now, we proceed to some more hints and conventions that we have to agreed upon throughout this article. In this article we will explain how to inject your own code into Award Bios by patching the POST jump table. But, before that, let's clarify a few things:

- What we mean by POST is the **Power On-Self Test** part of the Bios. The routines in this part do the testing of the system equipment and other intialization tasks.
- POST routines is part of the system bios (i.e. original.tmp file in Award Bios).
- POST routines is executed by means of a "jump table" in Award Bios as explained in the Award Bios Reverse Engineering paper by Darmawan Mappatutu Salihun [1].
- Based on the the Award Bios Reverse Engineering paper by Darmawan Mappatutu Salihun [1], it's clear that not all of the "POST jump table" contents are functioning. Some of them are just "dummy" routines, i.e. doing nothing at all beside just signaling successful execution and returning. Below is an example :

```
Address    Hex Values       Mnemonic                  Comment
000:6276                   RAM_POST_TESTS proc near ; CODE XREF:
last_E000_POST+D
E000:6276                                           ; last_E000_POST+18 ...
E000:6276 8A C1               mov  al, cl           ; cl = 3
E000:6278 E6 80               out  80h, al          ; manufacture's diagnostic
checkpoint
E000:627A 68 00 F0            push 0F000h
E000:627D 0F A1               pop  fs               ; fs = F000h
E000:627F
E000:627F                  ;This is the beginning of the call into E000 segment
E000:627F                  ;POST function table
E000:627F                     assume fs:F000
E000:627F 2E 8B 05            mov  ax, cs:[di]      ; in the beginning :
E000:627F                                           ; di = 61C2h ; ax = cs:[di]
= 154Eh
E000:627F                                           ; called from E000:2489 w/
di=61FCh (dummy)
E000:6282 47                  inc  di               ; Increment by 1
E000:6283 47                  inc  di               ; di = di + 2
E000:6284 0B C0               or   ax, ax           ; Logical Inclusive OR
E000:6286 74 0B               jz   RAM_post_return  ; RAM Post Error
E000:6288 57                  push di               ; save di
E000:6289 51                  push cx               ; save cx
E000:628A FF D0               call ax               ; call [61C2h] = call 154Eh
E000:628A                                           ; (relative call addr),one
of this call
E000:628A                                           ; won't return in normal
condition
E000:628C 59                  pop  cx               ; restore all
E000:628D 5F                  pop  di
E000:628E 72 03               jb   RAM_post_return  ; Jump if Below (CF=1)
E000:6290 41                  inc  cx               ; Increment by 1
E000:6291 EB E3               jmp  short RAM_POST_TESTS ; Jump
E000:6293                  ; ---------------------------------------------------
-------------------------
E000:6293
E000:6293                  RAM_post_return:         ; CODE XREF:
RAM_POST_TESTS+10 j
E000:6293                                           ; RAM_POST_TESTS+18 j
E000:6293 C3                  retn                  ; Return Near from Procedure
E000:6293                  RAM_POST_TESTS endp
.........
E000:61C2                  E0_POST_TESTS_TABLE:
E000:61C2 4E 15               dw 154Eh              ; Restore boot flag
E000:61C4 6F 15               dw 156Fh              ; Chk_Mem_Refrsh_Toggle
E000:61C6 71 15               dw 1571h              ; keyboard (and its
controller) POST
E000:61C8 D2 16               dw 16D2h              ; chksum ROM, check EEPROM
```

```
E000:61C8                                       ; on error generate spkr
tone
E000:61CA 45 17             dw 1745h            ; Check CMOS circuitry
E000:61CC 8A 17             dw 178Ah            ; "chipset defaults"
initialization
E000:61CC                                       ; file: E0POST.ASM and
CT_TABLE.ASM
E000:61CE 98 17             dw 1798h            ; init CPU cache (both Cyrix
and Intel)
E000:61D0 B8 17             dw 17B8h            ; init interrupt vector,
also initialize
E000:61D0                                       ; "signatures" used for
Ext_Bios components
E000:61D0                                       ; decompression
E000:61D2 4B 19             dw 194Bh            ; Init_mainboard_equipment &
CPU microcode
E000:61D2                                       ; chk ISA CMOS chksum ?
E000:61D4 BC 1A             dw 1ABCh            ; Check checksum. Initialize
keyboard controller
E000:61D4                                       ; and set up all of the 40:
area data.
E000:61D6 08 1B             dw 1B08h            ; Relocate extended Bios
code
E000:61D6                                       ; init CPU MTRR, PCI
REGs(Video Bios ?)
E000:61D8 C8 1D             dw 1DC8h            ; Video_Init (including EPA
proc)
E000:61DA 42 23             dw 2342h
E000:61DC 4E 23             dw 234Eh
E000:61DE 53 23             dw 2353h            ; dummy
E000:61E0 55 23             dw 2355h            ; dummy
E000:61E2 57 23             dw 2357h            ; dummy
E000:61E4 59 23             dw 2359h            ; init Programmable Timer
(PIT)
E000:61E6 A5 23             dw 23A5h            ; init PIC_1 (programmable
Interrupt Ctlr)
E000:61E8 B6 23             dw 23B6h            ; same as above ?
E000:61EA F9 23             dw 23F9h            ; dummy
E000:61EC FB 23             dw 23FBh            ; init PIC_2
E000:61EE 78 24             dw 2478h            ; dummy
E000:61F0 7A 24             dw 247Ah            ; dummy
E000:61F2 7A 24             dw 247Ah
E000:61F4 7A 24             dw 247Ah
E000:61F6 7A 24             dw 247Ah
E000:61F8 7C 24             dw 247Ch            ; this will call
RAM_POST_tests again
E000:61F8                                       ; for values below(a.k.a ISA
POST)
E000:61FA 00 00             dw 0
E000:61FA                 END_E0_POST_TESTS_TABLE
.........
E000:2353 F8                clc                 ; Clear Carry Flag
E000:2354 C3                retn                ; Return Near from Procedure
E000:2355                 ; ---------------------------------------------------
-------------------------
E000:2355 F8                clc                 ; Clear Carry Flag
E000:2356 C3                retn                ; Return Near from Procedure
E000:2357                 ; ---------------------------------------------------
-------------------------
E000:2357 F8                clc                 ; Clear Carry Flag
E000:2358 C3                retn                ; Return Near from Procedure
E000:2359
.........
E000:247A                 sub_E000_247A proc near
E000:247A F8                clc                 ; Clear Carry Flag
E000:247B C3                retn                ; Return Near from Procedure
```

```
E000:247B                   sub_E000_247A endp
.........
```

The **clc** (clear carry flag) routine above is used to signal the caller of the POST routine that everything went OK.

4. Hacking the POST Jump Table

Now we've already known all the prerequisite knowledge to do the code injection. We'd like to formulate the steps that we need to do this :

- Reverse engineer the Bios to look where the "POST jump table" located in the system bios (original.tmp). We suggest to begin the reverse engineering process in the bootblock and proceed to system bios (original.tmp) accordingly.
- Analyze the "POST jump table", and try to find a jump to dummy procedure. If we find one, continue to next step, otherwise we stop here since it's not possible to carry out this method on the Bios.
- Assemble our custom procedure using nasm. Note the resulting binary size. Try to minimize the injected code size to ensure that the injected code will fit into the "free space" of the system bios.
- Extract the genuine system bios (original.tmp) from the bios binary file using AwardBios editor.
- Analyze the system bios using hexeditor to look for padding bytes, where we can inject our code. If we don't find any suitable area, then we're out of luck and cannot proceed. But this is a very seldom case.
- Inject our assembled custom procedure to the extracted system bios (original.tmp) by using hexeditor.
- Modify the "POST jump table" to include a jump to our procedure. Use hexeditor to edit the system bios "POST jump table".
- Replace the genuine system bios (original.tmp) with the modified system bios by using AwardBios editor.
- Ensure the checksum of the modified Bios is fine by opening it using modbin and cbrom. I suggest to change the Bios name string using modbin and saving the change, since sometimes in "weird" Award Bios there are false checksums that were failed to be patched by Awardbios editor. Do a double check using modbin and cbrom to ensure the validity of the hacked Bios binary.
- Flash the hacked bios binary to the mainboard.

By following the above guidelines, we will finally arrive at our modified Bios which incorporate the injected code.

4.1. Bios Reverse Engineering and Analysis

We have done this, the result can be seen in the Award Bios Reverse Engineering paper by Darmawan Mappatutu Salihun [1]. The "POST jump table" location is provided in the Prerequisite section. It's very clear there that we have several candidates of dummy procedure jumps that we can replace with our own procedure jump. They are highlighted with red color).

4.2. Assembling The Custom Procedure

The following is the source code of the procedure that is injected into the bios (using nasm syntax):

```
;---------------- BEGIN TWEAK.ASM ----------------------------------------
BITS 16 ;just to make sure nasm prefix 66 to 32 bit instructions, we're
assuming the uP
        ;is in 16 bits mode up to this point (from the boot state)

  section   .text

start:

        pushf
        push eax
        push dx

        mov eax,ioq_reg ;patch the ioq register of the chipset
        mov dx,in_port
        out dx,eax
        mov dx,out_port
        in  eax,dx
        or  eax,ioq_mask
        out dx,eax

        mov eax,dram_reg ;patch the DRAM controller of the chipset,
        mov dx,in_port  ;i.e. the interleaving part
        out dx,eax
        mov dx,out_port
        in  eax,dx
        or  eax,dram_mask
        out dx,eax

        mov eax,bank_reg ;Allow pages of different bank to be active
simultanoeusly
        mov dx,in_port
        out dx,eax
        mov dx,out_port
        in  eax,dx
        or  eax,bank_mask
        out dx,eax

        mov eax,tlb_reg ;Activate Fast TLB lookup
        mov dx,in_port
        out dx,eax
        mov dx,out_port
        in  eax,dx
        or  eax,tlb_mask
        out dx,eax

        pop dx
        pop eax
        popf

        clc             ;indicate that this POST routine successful
        retn            ;return near to the header of the rom file

  section .data
```

```
  in_port   equ 0cf8h
  out_port  equ 0cfch
  dram_mask equ 00020202h
  dram_reg  equ 80000064h
  ioq_mask  equ 00000080h
  ioq_reg   equ 80000050h
  bank_mask equ 20000840h
  bank_reg  equ 80000068h
  tlb_mask  equ 00000008h
  tlb_reg   equ 8000006ch
;---------------- END TWEAK.ASM ----------------------------------------
```

The code is assembled using nasm with the invocation syntax :

```
nasm -fbin tweak.asm -o tweak.bin
```

The resulting binary file is **tweak.bin**. The following is the hex-dump of this binary in hexworkshop v3.02

```
Address   Hexadecimal Values                         ASCII
00000000 9C66 5052 66B8 5000 0080 BAF8 0C66 EFBA .fPRf.P......f..
00000010 FC0C 66ED 660D 8000 0000 66EF 66B8 6400 ..f.f.....f.f.d.
00000020 0080 BAF8 0C66 EFBA FC0C 66ED 660D 0202 .....f....f.f...
00000030 0200 66EF 66B8 6800 0080 BAF8 0C66 EFBA ..f.f.h......f..
00000040 FC0C 66ED 660D 4008 0020 66EF 66B8 6C00 ..f.f.@.. f.f.l.
00000050 0080 BAF8 0C66 EFBA FC0C 66ED 660D 0800 .....f....f.f...
00000060 0000 66EF 5A66 589D F8C3                ..f.ZfX...
```

The dump above shows that we need **0x6A** bytes (106 bytes) free space to inject this code in system bios.

4.3. Injecting The Procedure

Now,extract the system bios by using AwardBios editor. It's very simple, just open the bios file then select the **System Bios** tree-item in the left pane, then click the **Action**|**Extract File** to save the system bios as a separate uncompressed binary file. *As convention in this paper, let's name it **original.tmp**.*

Then, open **original.tmp** using hexeditor. In this particular **original.tmp**, we found a lot of padding **FFh** bytes in the end of segment **E000h**. Perhaps, this quite confusing at first, an easier explanation: *In the the Award Bios Reverse Engineering paper by Darmawan Mappatutu Salihun [1], it's mentioned that the POST jump table resides in the **E000h** segment and the jump table contains addresses in Little-Endian 16 bit value. This means that the jump table is only for intra-segment jumps, hence, the injected procedure must reside in the same segment as the POST jump table itself, i.e. segment **E000h**. So, the "free space" that can be used for the injected procedure must reside in segment **E000h**. Most of the time this "free space" is padding bytes.*

If you still confused, let refresh your memory about the mapping between original.tmp in the real system address space and in the hexeditor that we use. Original.tmp size is 128KB, it uses the E000h and F000h segment during its execution. So, if you see address **0000 0000h** in your hexeditor for this file, it's basically address **E000:0000h** when original.tmp gets executed, and so forth. Due to this fact, we have to look for "free space", i.e. unused area or padding bytes below the address **0001 0000h** in the hexeditor.

Below is the snapshot of the beginning of the padding bytes in both IDA Pro 4.50 and Hexworkshop v3.02 for exactly the same address.

In IDA Pro 4.50:

```
Address   Hex Values          Mnemonic    Comment
E000:EFE0 C3                    db 0C3h ; +
E000:EFE1 00                    db    0 ;
E000:EFE2 00                    db    0 ;
E000:EFE3 00                    db    0 ;
E000:EFE4 00                    db    0 ;
E000:EFE5 00                    db    0 ;
E000:EFE6 00                    db    0 ;
E000:EFE7 00                    db    0 ;
E000:EFE8 00                    db    0 ;
E000:EFE9 00                    db    0 ;
E000:EFEA 00                    db    0 ;
E000:EFEB 00                    db    0 ;
E000:EFEC 00                    db    0 ;
E000:EFED 00                    db    0 ;
E000:EFEE 00                    db    0 ;
E000:EFEF 00                    db    0 ;
E000:EFF0 FF                    db 0FFh ;
E000:EFF1 FF                    db 0FFh ;
E000:EFF2 FF                    db 0FFh ;
E000:EFF3 FF                    db 0FFh ;
E000:EFF4 FF                    db 0FFh ;
E000:EFF5 FF                    db 0FFh ;
E000:EFF6 FF                    db 0FFh ;
E000:EFF7 FF                    db 0FFh ;
E000:EFF8 FF                    db 0FFh ;
E000:EFF9 FF                    db 0FFh ;
E000:EFFA FF                    db 0FFh ;
E000:EFFB FF                    db 0FFh ;
E000:EFFC FF                    db 0FFh ;
E000:EFFD FF                    db 0FFh ;
E000:EFFE FF                    db 0FFh ;
E000:EFFF FF                    db 0FFh ;
```

In Hexworkshop 3.02:

```
Address     Hex values                                ASCII
0000EFE0 C300 0000 0000 0000 0000 0000 0000 0000 ................
0000EFF0 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ................
```

Looking at the amount of padding bytes in original.tmp, we know that we have enough space to do the code injection. What we need to do is: use the hexeditor to replace 106 bytes beginning at E000:EFF0h (0000EFF0h) with the code that already assembled (in 16-bit x86 executable binary format) in the previous step. In hexworkshop, this step is trivial, just open **original.tmp** and **tweak.bin** in the same hexworkshop, then copy and paste **tweak.bin** contents to **original.bin**, that's it. The result in hexworkshop as follows (the hex-values highlighted in red is the injected code):

```
Address     Hex values                                      ASCII
0000EFD0 C300 0000 0000 0000 0000 0000 0000 0000 ................
0000EFE0 C300 0000 0000 0000 0000 0000 0000 0000 ................
0000EFF0 9C66 5052 66B8 5000 0080 BAF8 0C66 EFBA .fPRf.P......f..
0000F000 FC0C 66ED 660D 8000 0000 66EF 66B8 6400 ..f.f.....f.f.d.
0000F010 0080 BAF8 0C66 EFBA FC0C 66ED 660D 0202 .....f....f.f...
0000F020 0200 66EF 66B8 6800 0080 BAF8 0C66 EFBA ..f.f.h......f..
0000F030 FC0C 66ED 660D 4008 0020 66EF 66B8 6C00 ..f.f.@.. f.f.l.
0000F040 0080 BAF8 0C66 EFBA FC0C 66ED 660D 0800 .....f....f.f...
0000F050 0000 66EF 5A66 589D F8C3 FFFF FFFF FFFF ..f.ZfX.........
0000F060 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ................
0000F070 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF ................
```

If you eager to know what the code above accomplished, the snapshot of the chipset datasheet is provided below. Unfortunately, you still need know PCI protocol to make use of it. This is the snapshot for PCI device at address bus 0 - device 0 - function 0, i.e. the hostbridge of the corresponding mainboard.

Device 0 Configuration Registers - Host Bridge

These registers are normally programmed once at system initialization time.

Host CPU Control

Device 0 Offset 50 – Request Phase Control (00h) ..... RW

7 CPU Hardwired IOQ (In Order Queue) Size

Default per strap on pin MAB11#During reset. This register can be written 0 to restrict the chip to one level of IOQ.

0 1-Level

1 4-Level

6 Read-Around-Write

0 Disable .....default

1 Enable

5 Reserved ..... always reads 0

4 Defer Retry When HLOCK Active

0 Disable .....default

1 Enable

Note: always set this bit to 1

3-1 Reserved ..... always reads 0

0 CPU / PCI Master Read DRAM Timing

0 Start DRAM read after snoop complete ..... def

1 Start DRAM read before snoop complete

DRAM Control

These registers are normally set at system initialization time and not accessed after that during normal system operation. Some of these registers, however, may need to be programmed using specific sequences during power-up initialization to properly detect the type and size of installed memory (refer to the VIA Technologies VT82C693A Bios porting guide for details).

SDRAM Settings for Registers 67-64

7 Precharge Command to Active Command Period
0 TRP = 2T
1 TRP = 3T ..... default

6 Active Command to Precharge Command Period
0 TRAS = 5T
1 TRAS = 6T ..... default

5-4 CAS Latency
00 1T
01 2T
10 3T ..... default
11 reserved

3 DIMM Type
0 Standard
1 Registered ..... default

2 ACTIVE Command to CMD Command Period /
VCM Prefetch Read Latency
0 2T / 3T
1 3T / 4T ..... default

1-0 Bank Interleave
00 No Interleave ..... default
01 2-way
10 4-way
11 Reserved

Device 0 Offset 68 - DRAM Control (00h) ..... RW

7 SDRAM Open Page Control
0 Always precharge SDRAM banks when
accessing EDO/FPG DRAMs.....default
1 SDRAM banks remain active when accessing
EDO/FPG banks

6 Bank Page Control
0 Allow only pages of the same bank active.. def.
1 Allow pages of different banks to be active

5 Reserved ..... always reads 0

4 DRAM Data Latch Delay for EDO/FPG DRAM
0 Latch DRAM data at CCLK rising edge .... def.
1 Delay latch of DRAM data by ? CCLK

3 EDO Test Mode
0 Disable .....default
1 Enable

2 Burst Refresh
0 Disable .....default
1 Enable (burst 4 times)

1 System Frequency Divider .....RO
This bit is latched from MAB8# at the rising edge of RESET# (see table below).

0 System Frequency Divider .....RO
This bit is latched from MAB12# at the rising edge of RESET#.
00 CPU Frequency = 66 MHz
01 CPU Frequency = 100 MHz
10 CPU Frequency = 133 MHz
11 Reserved

Note: See also Rx69[7-6]
Note: MD0 is internally pulled up for EDO detection.

```
Device 0 Offset 6C - SDRAM Control (00h) ................... RW
7-5 Reserved ....................................... always reads 0
4 CKE Configuration
     0 Rx6B[4]=0 RASA = CSA, RASB = CSB,
                 CKE0=CKE0, CKE1 = CKE1
     x Rx6B[4]=1 RASA = CSA, RASB = Float,
                 CASB = Float, MAB = Float,
                 CKE0 = CKE0, CKE1 = CKE0
     1 Rx6B[4]=0 RASA = CSA, RASB = CSB,
                 CKE3-2 = CSA7-6
                 CKE5-4 = CSB7-6
                 CKE1 = GCKE (Global CKE)
                 CKE0 = FENA (FET Enable)
3 Fast TLB Lookup
     0 Disable ..................................................default
     1 Enable
2-0 SDRAM Operation Mode Select
     000 Normal SDRAM Mode .........................default
     001 NOP Command Enable
     010 All-Banks-Precharge Command Enable
         (CPU-to-DRAM cycles are converted
         to All-Banks-Precharge commands).
     011 MSR Enable
         CPU-to-DRAM cycles are converted to
         commands and the commands are driven on
         MA[14:0]. The Bios selects an appropriate
         host address for each row of memory such that
         the right commands are generated on
         MA[14:0].
     100 CBR Cycle Enable (if this code is selected,
         CAS-before-RAS refresh is used; if it is not
         selected, RAS-Only refresh is used)
     101 Reserved
     11x Reserved
```

After this step, we proceed to next step to patch the jump table.

4.4. Modifying The Jump Table

Modifying the POST jump table is just a trivial task after we do the reverse engineering in the bios binary. As presented above, in the prerequisite section, there are lots of jump table entries that points to "dummy" procedures.

We decided to redirect/replace the jump table entry at **E000:61DEh** to point to our injected procedure (at **E000:EFF0h**) instead to the previous "dummy" procedure. Below is the snapshot in both IDA Pro 4.50 and Hexworkshop, before the modification takes place :

In IDA Pro 4.50:

```
Address   Hex Values          Mnemonic    Comment
E000:61DC 4E 23                 dw 234Eh
E000:61DE 53 23                 dw 2353h            ; dummy
E000:61E0 55 23                 dw 2355h            ; dummy
E000:61E2 57 23                 dw 2357h            ; dummy
E000:61E4 59 23                 dw 2359h            ; init Programmable
Timer (PIT)
E000:61E6 A5 23                 dw 23A5h            ; init PIC_1
(programmable Interrupt Ctlr)
E000:61E8 B6 23                 dw 23B6h            ; same as above ?
E000:61EA F9 23                 dw 23F9h            ; dummy
E000:61EC FB 23                 dw 23FBh            ; init PIC_2
E000:61EE 78 24                 dw 2478h            ; dummy
E000:61F0 7A 24                 dw 247Ah            ; dummy
E000:61F2 7A 24                 dw 247Ah
E000:61F4 7A 24                 dw 247Ah
E000:61F6 7A 24                 dw 247Ah
.........
E000:2353 F8                    clc                 ; Clear Carry Flag
E000:2354 C3                    retn                ; Return Near from
Procedure
.........
```

In Hexworkshop 3.02:

```
Address     Hex values                              ASCII
........
000061D0 B817 4B19 BC1A 081B C81D 4223 4E23 5323 ..K.......B#N#S#
000061E0 5523 5723 5923 A523 B623 F923 FB23 7824 U#W#Y#.#.#.#.#x$
000061F0 7A24 7A24 7A24 7A24                     z$z$z$z$
........
```

Below is the snapshot in both IDA Pro 4.50 and Hexworkshop, after the modification takes place :

In IDA Pro 4.50:

```
Address   Hex Values                 Mnemonic    Comment
E000:61DC 4E 23                    dw 234Eh
E000:61DE F0 EF                    dw 0EFF0h       ;jump to our injected
code
E000:61E0 55 23                    dw 2355h
.........
E000:EFF0 9C                       pushf               ; Push Flags
Register onto the Stack
E000:EFF1 66 50                    push eax
E000:EFF3 52                       push dx
E000:EFF4 66 B8 50 00 00 80        mov  eax, 80000050h
E000:EFFA BA F8 0C                 mov  dx, 0CF8h
E000:EFFD 66 EF                    out  dx, eax
E000:EFFF BA FC 0C                 mov  dx, 0CFCh
E000:F002 66 ED                    in   eax, dx
E000:F004 66 0D 80 00 00 00        or   eax, 80h       ; Logical
Inclusive OR
E000:F00A 66 EF                    out  dx, eax
E000:F00C 66 B8 64 00 00 80        mov  eax, 80000064h
E000:F012 BA F8 0C                 mov  dx, 0CF8h
```

```
E000:F015 66 EF                     out  dx, eax
E000:F017
E000:F017                         loc_EF017:              ; DATA XREF:
E000:19975
E000:F017                                                 ; E000:1997A
E000:F017 BA FC 0C                  mov  dx, 0CFCh
E000:F01A
E000:F01A                         loc_EF01A:              ; DATA XREF:
E000:19962
E000:F01A 66 ED                     in   eax, dx
E000:F01C 66 0D 02 02 02 00         or   eax, 20202h      ; Logical
Inclusive OR
E000:F022 66 EF                     out  dx, eax
E000:F024 66 B8 68 00 00 80         mov  eax, 80000068h
E000:F02A BA F8 0C                  mov  dx, 0CF8h
E000:F02D 66 EF                     out  dx, eax
E000:F02F BA FC 0C                  mov  dx, 0CFCh
E000:F032 66 ED                     in   eax, dx
E000:F034 66 0D 40 08 00 20         or   eax, 20000840h   ; Logical
Inclusive OR
E000:F03A 66 EF                     out  dx, eax
E000:F03C 66 B8 6C 00 00 80         mov  eax, 8000006Ch
E000:F042 BA F8 0C                  mov  dx, 0CF8h
E000:F045 66 EF                     out  dx, eax
E000:F047 BA FC 0C                  mov  dx, 0CFCh
E000:F04A 66 ED                     in   eax, dx
E000:F04C 66 0D 08 00 00 00         or   eax, 8           ; Logical
Inclusive OR
E000:F052 66 EF                     out  dx, eax
E000:F054 5A                        pop  dx
E000:F055 66 58                     pop  eax
E000:F057 9D                        popf                  ; Pop Stack into
Flags Register
E000:F058 F8                        clc                   ; Clear Carry
Flag
E000:F059 C3                        retn                  ; Return Near
from Procedure
.........
```

In Hexworkshop 3.02:

```
Address     Hex values                              ASCII
........
000061D0 B817 4B19 BC1A 081B C81D 4223 4E23 F0EF ..K.......B#N#..
000061E0 5523 5723 5923 A523 B623 F923 FB23 7824 U#W#Y#.#.#.#.#x$
........
0000EFE0 C300 0000 0000 0000 0000 0000 0000 0000 ................
0000EFF0 9C66 5052 66B8 5000 0080 BAF8 0C66 EFBA .fPRf.P......f..
0000F000 FC0C 66ED 660D 8000 0000 66EF 66B8 6400 ..f.f.....f.f.d.
0000F010 0080 BAF8 0C66 EFBA FC0C 66ED 660D 0202 .....f....f.f...
0000F020 0200 66EF 66B8 6800 0080 BAF8 0C66 EFBA ..f.f.h......f..
0000F030 FC0C 66ED 660D 4008 0020 66EF 66B8 6C00 ..f.f.@.. f.f.l.
0000F040 0080 BAF8 0C66 EFBA FC0C 66ED 660D 0800 .....f....f.f...
0000F050 0000 66EF 5A66 589D F8C3 FFFF FFFF FFFF ..f.ZfX.........
........
```

By now, we have patched original.tmp to suit our need. The next thing to do is combining it back into one functional bios binary.

4.5. Recombining Bios Component and Fixing Checksums

This step is also trivial. Just open the previous bios binary from which we extract the original.tmp using awardbios editor. Then select the **System Bios** tree-item in the left pane, and proceed to click the **Action|Replace File** menu. After that select the modified original.tmp as the file used to replace the genuine original.tmp in that bios binary. Then save this change in awardbios editor.

Actually we are done at this point, but some "nasty" Award Bios sometimes causes awardbios editor failed to fix its checksum. To guard against this possible bug, open this modified bios binary using modbin, then do some minor changes, such as changing the bios string and then saving this change in modbin. This step, will causes modbin to recalculate all checksums and fix the possibly wrong checksums. That's all, voila' we're done :).

4.6. Testing The Modified Bios

Testing is also a trivial task, just flash the modified bios binary. We are using uniflash to do this in our testbed, since the awardflash is unable to handle my Atmel AT29C020C-90 backup-bios chip that were used in the testbed's mainboard, whereas uniflash v1.34 can handle flawlessly. Thanks to Ondrej Zary a.k.a Rainbow, who provide us with this great uniflash bios flashing utility.

5. Possible Downside and Its Workaround

During the experiment using this method to patch our bios, we encounter a weird situation that confusing at first. The bug that we encounter would hang my machine at boot, but it's very seldom and hard to reproduce, i.e. around 1 out of 30 tries. This bug is in effect if the following jump table modification is carried out.

Note : 1. The modification explained in the previous sections proved to be bug free after lots of testing and verifications.
2. The code is injected in the same place as explained in the previous sections.

The following is the jump table before the "buggy" patch incorporated :

```
Address     Hex Values          Mnemonic            Comment
E000:61DE 53 23                 dw 2353h            ; dummy
E000:61E0 55 23                 dw 2355h            ; dummy
E000:61E2 57 23                 dw 2357h            ; dummy
E000:61E4 59 23                 dw 2359h            ; init Programmable
Timer (PIT)
E000:61E6 A5 23                 dw 23A5h            ; init PIC_1
(programmable Interrupt Ctlr)
E000:61E8 B6 23                 dw 23B6h            ; same as above ?
E000:61EA F9 23                 dw 23F9h            ; dummy
E000:61EC FB 23                 dw 23FBh            ; init PIC_2
E000:61EE 78 24                 dw 2478h            ; dummy
E000:61F0 7A 24                 dw 247Ah            ; dummy
E000:61F2 7A 24                 dw 247Ah
E000:61F4 7A 24                 dw 247Ah
E000:61F6 7A 24                 dw 247Ah
.........
```

The following is the jump table after the "buggy" patch incorporated :

```
Address     Hex Values          Mnemonic            Comment
E000:61DE 53 23                 dw 2353h            ; dummy
E000:61E0 55 23                 dw 2355h            ; dummy
E000:61E2 57 23                 dw 2357h            ; dummy
E000:61E4 59 23                 dw 2359h            ; init Programmable
Timer (PIT)
E000:61E6 A5 23                 dw 23A5h            ; init PIC_1
(programmable Interrupt Ctlr)
E000:61E8 B6 23                 dw 23B6h            ; same as above ?
E000:61EA F9 23                 dw 23F9h            ; dummy
E000:61EC FB 23                 dw 23FBh            ; init PIC_2
E000:61EE 78 24                 dw 2478h            ; dummy
E000:61F0 F0 EF                 dw EFF0h            ; dummy
E000:61F2 7A 24                 dw 247Ah
E000:61F4 7A 24                 dw 247Ah
E000:61F6 7A 24                 dw 247Ah
.........
```

After further analysis, we conclude that this kind of bug very possibly related to timing issue and race condition during the code execution in POST. If we take a look closely at the jump table redirection, we see that this bug occur if we modify/redirect the jump table entry after the initialization of the Programmable Interrupt Controller (PIC) in the mainboard. Perhaps, the best way to avoid this is to place our jump table modification before the PIC initialization. Based on the testing result, doing so proved to be flawless and successfully eradicate the bug. We summarised some guidelines to avoid this bug in your jump table modification below :

- Analyze your code carefully and preserve the machine state during the execution of your code and don't forget to restore the machine state after execution of your code. The machine state we mean here is the registers affected by your code, such as the general purpose registers and the flag register. We've been bitten by this bug due to not preserving the flag register.
- Only save the registers and flags that are used/influenced by your routines as we already shown in our flawlessly executed example in the Assembling Our Custom Procedure section above.
- Don't forget to clear the carry flag (execute **clc**) prior to returning from your custom procedure. This is needed in Award Bioses to indicate that the POST procedure (in this case our injected custom procedure) is successfully executed.
- Patch/redirect the jump table entry only before the Programmable Interrupt Controller (PIC) initialization. This is perhaps a quite weird advice, but based on our experience, bios is a very strict software component in terms of timing. We don't guarantee that the assumption in this case is strictly right, but that's the best logical explanation to the bug that we encounter during the modification process. Also, we have to underline that the sample jump table modification in the Modifying The Jump Table section is flawless and have been tested thoroughly.

That's all about the possible downsides of this method and their workaround. It's possible that the explanation in this section is wrong. We really sorry about that, since we are still in the process of learning about this subject too.

6. Closing

Finally we are done. This bios code injection method is very possibly the most elegant trick to date. We haven't found any new elegant way to accomplish it. In this paper we have proved that the opportunity to carry out Award bios code injection is not only a mere possiblity, but also can be implemented flawlessly using tools widely available today.

References

1. Darmawan Mappatutu Salihun, *Award Bios Reverse Engineering*: The CodeBreakers-Journal, Vol. 1, No.2 (2004)