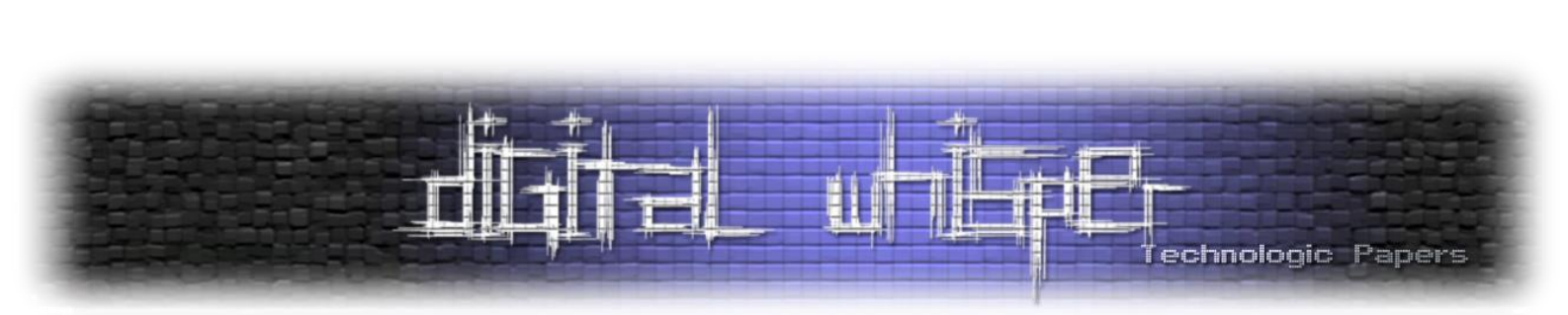


הנדסה-לאחור: שרשרת העלייה של Windows 7 חלק ראשון - MBR

מאת 0x3d5157636b525761

רקע: הנדסה לאחור

מהי הנדסה לאחור? הנדסה לאחור (Reverse Engineering) היא פעולה הפוכה (לכן מופיעה המילה "לאחור" במונח). אם "הנדסה" היא הרכבת מערכת שלמה ממרכיבים שונים לביצוע מטלה, אז "הנדסה לאחור" היא פירוק המערכת השלמה למרכיבים קטנים שאותם ניתן להבין ולנתח.

בהקשרי תוכנה, "הנדסה" היא תהליך בניית וחיבור מודולים זה לזה על מנת ליצור תוכנה ("המערכת השלמה"), בעוד ש-"הנדסה לאחור" היא פירוק התוכנה למודולים מובנים (רצוי בשפה עילית או פסאודו קוד), ניתוחם והבנתם. ספציפית נתייחס בדרך כלל להבנת קוד אסמבלי ונסיון להנדסו לאחור לקוד בשפה עילית כגון C, או לחילופין הבנת האלגוריתם הכללי.

הנדסה לאחור היא אמנות ולכן יכולה להתבצע בכל כלי שנבחר: הרצה דינאמית, כלי ניתוח סטטיים, חיפוש באינטרנט, קריאת ההוראות (!) ועוד.

הערת צד: ישנן בעיות חוקיות בכל הנוגע להנדסה לאחור. מדריך זה נועד ללימוד עצמי בלבד!

ידע וכלים נדרשים

מן הקורא מצופה להכיר:

- ארכיטקטורת מעבדי אינטל לדורותיהם (איך נראה instruction, מנגנון ה-ring-ים ועוד).
- קוד אסמבלי של אינטל במצבים שונים: real mode, virtual mode וכדומה. אני אעבוד ב-intel syntax כי לצערי אינני יכול לסבול את התחביר הנוראי של AT&T. המזוכיסטים מביניכם יתמודדו.
- עבודה עם IDA (עבודה ברמת האסמבלי, לא hex rays או החלפת jnz בשביל הצחוקים).

כלים דרושים:

- בעיקר IDA, כאמור.
- אשתמש גם ב-python, אבל זה יהיה רק לשבריר שנייה. למעשה, בסוף הכתבה תשכחו מזה לחלוטין.
- רפרנס טוב נוסף הוא Ralph Brown's Interrupt List, או בקיצור RBIL. כל מי שאי פעם תכנת ב-real mode assembly כנראה מכיר - בכל מקרה, RBIL הוא אחלה רפרנס ל-interrupts שונים. הכתובת היא: http://www.ctyme.com/rbrown.htm.
- intel instruction set, הנה אחד לדוגמא:

http://www.mathemainzel.info/files/x86asmref.html

רקע: תהליך עליית מחשב

תהליך העלייה של מחשב באופן כללי ארוך ומסורבל, וכולל מעברי מצב שונים של המעבד, קנפוגי חומרה שונים (PCI למשל), חלקי קוד שונים (BIOS \ Bootloader \ Kernel) ועוד.

הערת צד: בשנים האחרונות יש מעבר למנגנון חדש בשם UEFI. אנחנו נתעלם בינתיים מכך - אני כן מתכנן מתישהו לעשות כתבות על UEFI.

אנחנו נדלג לגמרי על החלק (המרתק!) של עליית המחשב ונתמקד בקוד שנטען על ידי ה-BIOS, אבל לפני כן חשוב להבין מה מצב המחשב לאחר עליית ה-BIOS.

אם כן, מה עושה ה-BIOS? המון! עם זאת, נציין highlights רלוונטיים:

- קנפוגים שונים של החומרה (עדכון ערוצי PCI, קנפוג ה-memory controller ועוד).
- מילוי טבלת פסיקות (שעליה נדבר בקרוב).
- זיהוי חומרה (חלק הידוע כ-POST).
- עבודה לפי קונפיגורציה (הידועה כ-CMOS ויושבת כ-nvram).
- אנומרציה של devices שונים במטרה למצוא bootable device, לפי סדר שנשמר ב-CMOS.

יש לשים לב שה-BIOS מתחיל לרוץ במצב של המעבד הנקרא real mode. הזיכרון שנוגעים בו הוא ישיר (אין זיכרון וירטואלי), הזיכרון segmented, משתמשים בדרך כלל רק באוגרים בגודל 16 ביט (AX וחברים).

כאשר ה-BIOS מוצא bootable device, הוא קורא 512 בתים ראשונים ממנו, טוען אותם לכתובת 0x7C00 ומעביר לשם את השליטה. כאן אנחנו מתחילים את המשחק שלנו. עוד מידע נוסף שה-BIOS מעביר: מספר הכונן שנטען יימצא באוגר DL. חשוב לזכור לאחר מכן.

הערת צד: אנשים שעובדים עם אמולטורים כגון QEMU או BOCHS יגלו הבדלים מזעריים בין מחשב פיזי לאמולטור - למשל, כל ה-RAM מאופס באמולטור (במחשב אמיתי ה-RAM יכיל מידע אקראי ולוא דווקא אפסים), ה-memory controller כבר יהיה במצב A20 (במחשב אמיתי זה לא מובטח) ועוד.

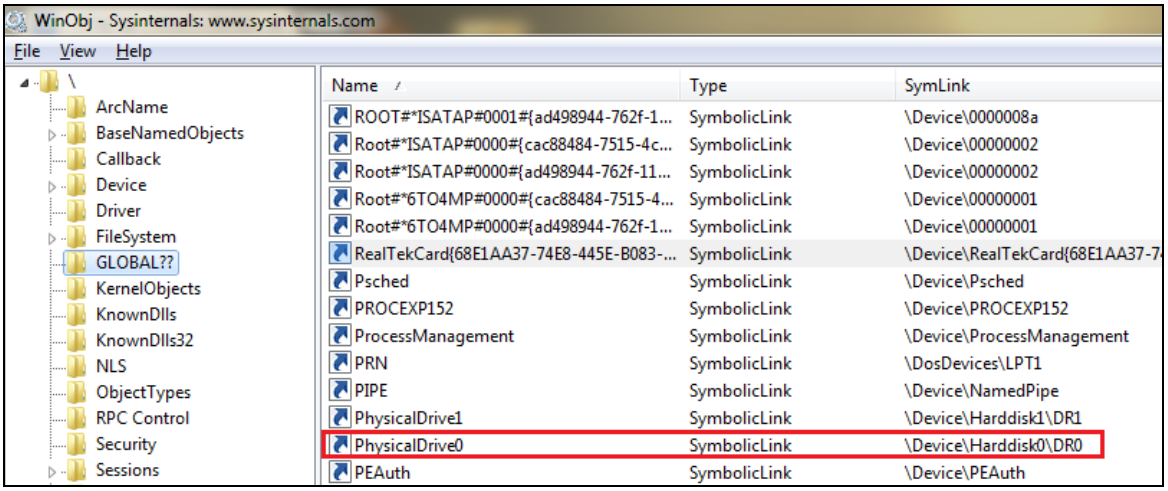
ה-Setup

במהלך המדריך אני אעבוד על Windows 7 SP1, x64. אני מניח שדברים עלולים להיות שונים קצת בין מחשב למחשב. בחלק זה נבצע רק ניתוח סטטי, אז לא אמור להגרם שום נזק.

חשיפת ה-MBR

ה-MBR (קיצור של Master Boot Record) הוא 512 הבתים הראשונים בדיסק הפיזי. יש להם פורמט מיוחד שאליו נתייחס בקרוב, אבל חשוב לציין שהפורמט הזה לא נדרש על ידי BIOS - מבחינת ה-BIOS, אם 512 הבתים הראשונים נגמרים ב-0xAA55 אז ה-device הוא bootable, ואחריות הקוד שנטען לדאוג לוידוא הפורמט של ה-MBR (הוא יכול גם לא לעשות את זה כמובן).

תחת לינוקס ניתן לקרוא את הדיסק הפיזי על ידי פנייה ל-device הרלוונטי וביצוע dd. ב-Windows זה סיפור שונה. בתור תומך python נלהב אציג הדרכה של חשיפת ה-MBR באמצעות python. כמובן שאתם יכולים לבצע זאת בכל דרך שבא לכם. ה-physical device ממופה ב-Windows כ-PhysicalDrive0. באמצעות Winobj של SysInternals ניתן לראות גם שזה סתם symbolic link, אבל נשתמש בו בכל זאת:



אם כן, כיצד חושפים את ה-MBR? פשוט קוראים 512 בתים מתוכו, כמו שהיינו עושים בלינוקס עם dd.

כפי שהבטחתי, python להמונים:

```
ActivePython 2.7.6.9 (ActiveState Software Inc.) based on
Python 2.7.6 (default, Feb 27 2014, 14:13:40) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> mbr = open('\\\\.\\PhysicalDrive0', 'rb').read(512)
>>> open('C:\\mbr.bin', 'wb').write(mbr)
>>>
```

כעת יש לנו MBR חשוף, נרצה לנתח אותו.

ניתוח ראשוני עם IDA

נפתח את הקובץ ב-IDA. כמובן, IDA לא יודעת לנתח את הקובץ ישירות כי מדובר בקוד טהור ולא בפורמט מוגדר כגון ELF או PE. לכן, IDA תציג הכל כ-DATA בהצלחה, וגם תשאל האם לנתח כקוד 32 ביט או 16 ביט. נבחר באפשרות ה-16 ביט, כי כאמור - אנחנו עובדים ב-Real mode.

לאחר שהקובץ נפתח, נראה רק DATA וסגמנט יחיד שמתחיל בכתובת 0. לא להיט, כי אנחנו יודעים שאמורים להתחיל ב-0x7C00. לכן, נבצע את שתי הפעולות הבאות:

- שינוי כתובת התוכנית (בתפריטים: Edit → Segment → Rebase program ולכתוב 0x7C00).
- סימון הבית הראשון ולחיצה על C (גורם ל-IDA לנתח קוד ולא DATA).

כעת התוכנית ברורה יותר, ואנחנו מוכנים להתחיל להבין מה הולך כאן. התהליך שנבצע כעת יעבוד בשלבים: ניתוח כל chunk של קוד עד להבנה מוחלטת. אגב, נשים לב ששני הבתים האחרונים הם אכן 0xAA55, כצפוי.

<u>חלק א': כל כך מעט קוד?</u>

אם אכן לחצתם C, כנראה שאתם רואים מעט מאד קוד אחרי הכל. מדוע? IDA עובדת בשיטת ניתוח בשם recursive descent, כלומר, כל חלק מנותח כקוד אם קופצים אליו או אם לפניו גם מופיעה שורת קוד. מהר מאד נבין למה אנחנו רואים מעט קוד, אבל בינתיים נתבונן במה שיש מולנו (את ההערות הוספתי בעצמי):

```
seg000:7C00
seg000:7C00                 xor     ax, ax
seg000:7C02 ;
seg000:7C02 ; Build stack (SS:SP = 00:7C00)
seg000:7C02 ;
seg000:7C02                 mov     ss, ax
seg000:7C04                 mov     sp, 7C00h
seg000:7C07 ;
seg000:7C07 ; Nullify ES and DS
seg000:7C07 ;
seg000:7C07                 mov     es, ax
seg000:7C09                 mov     ds, ax
seg000:7C0B ;
seg000:7C0B ; Self replicate to 0x600
seg000:7C0B ;
seg000:7C0B                 mov     si, 7C00h
seg000:7C0E                 mov     di, 600h
seg000:7C11                 mov     cx, 200h
seg000:7C14                 cld
seg000:7C15                 rep movsb
```

ניתוח:

- כתובת 7C00: איפוס של AX על ידי XOR עצמי. זה טריק ידוע מאד באסמבלי - הקסרה עצמית שקולה לאיפוס (מלבד side-effects של השפעה על הדגלים).
- בניית המחסנית: נשים לב שביצוע PUSH, POP, CALL ו-RET למיניהם בלתי אפשריים ללא מחסנית. אי אפשר להניח שה-BIOS השאיר לנו סגמנטים ורגיסטרים מסודרים ובטח שלא השאיר מחסנית יפהפייה, ולכן מאפסים את SS ושמים ב-SP את הערך 0x7C00.

- **נקודות עדינות:**
  - המחסנית גדלה לכתובות נמוכות. זה אומר שהמחסנית לעולם לא תעבור את 0x7C00, אלא אם כן כמובן נבצע יותר POP-ים מ-PUSH-ים, ואז מגיע לנו לקבל קריסות על פי מיטב הכללים של דרווין.
  - מדוע לא איפסנו את SS עם הקסרה עצמית? למה צריך את AX? התשובה היא שאין opcode רלוונטי להקסרה של SS, ובדרך כלל נגיעה באוגרי segment היא MOV.
  - אנקדוטה משעשעת: ל-BIOS גם אין מחסנית (ולמעשה הוא לא יודע לפני POST שיש לו RAM בכלל)! איך הוא מבצע קריאות ושומר מידע? התשובה היא ש-BIOS שומר את כל המידע הדרוש לו על ה-cache של המעבד.
- איפוס הסגמנטים האחרים: ES ו-DS, שישמשו אותנו אחר כך. אם כבר חוגגים על AX, אז עד הסוף.
- שכפול עצמי של ה-MBR את 0x600. מדוע עושים את זה? התשובה היא שיש צורך לחזות את העתיד: התפקיד של ה-MBR הוא למצוא bootable partition ולטעון מתוכו את ה-VBR, שהוא 512 הבתים הראשונים של המחיצה הרלוונטית ("מסורתית": כונן C שלכם). לאן ה-VBR אמור להטען? תשובה משעשעת: 0x7C00. זה אומר שמראש צריך לפנות לו מקום, וזה בדיוק מה שהולך לקרות בקוד הנוכחי. איך עושים זאת? שימוש בפקודות מחרוזת של אסמבלי:
  - אוגר SI אמור להצביע על כתובת המקור, במקרה שלנו, על 0x7C00.
  - אוגר DI אמור להצביע על כתובת היעד. מעתיקים אל 0x600, אז לשם.
  - אוגר CX מכיל את מספר הבתים שיש להעתיק, שזה 512 או 0x200 בקיצור.
  - פקודת CLD שמה 0 ב-direction flag. למי שלא מכיר, זהו דגל שמשפיע על הכיוון של פעולות מחרוזת כגון הפקודה הבאה. אם היה שם 1 אז היינו הולכים בכיוון ההפוך, כלומר, 512 בתים אחורה! כפי שאמרתי, אין להסתמך על זה שה-BIOS השאיר את ה-direction flag נקי.
  - ביצוע REP MOVSB. מה שהולך לקרות הוא ביצוע של MOVSB כמספר הפעמים שכתוב ב-CX. מה MOVSB עושה? מעתיק בית מ-SI אל DI ומגדיל את שניהם (במקרה שה-direction flag מאופס, כאמור).

- **נקודה עדינה:** מה שאמרתי היה קצת נאיבי. SI ו-DI הם מצביעים, ולכן הם נמצאים בסגמנט. למעשה, MOVSB מעתיק מ-DS:SI אל ES:DI. לכן היה חשוב גם לאפס את הסגמנטים לפני ההעתקה העצמית!

סיכום: קטע הקוד ביצע אתחול ראשוני של הסגמנטים, המחסנית וכן העתיק את ה-MBR אל 0x600. חשוב לשים לב שאף על פי שה-MBR הועתק אל 0x600, הקוד שלנו עדיין רץ ב-0x7C00! למעשה, אנחנו מצפים שהקוד שלנו יזוז "מספיק רחוק" מ-0x7C00 על מנת שיתפנה מקום ל-VBR. למעשה (ספוילר!), זה בדיוק מה שהולך לקרות עכשיו, וזו גם הסיבה ש-IDA לא הצליחה לנתח את המשך הקוד.

חלק ב': אז איך נוגעים ב-CS?

הנה נתבונן בקטע הקוד (הקצר!) הבא (ההערות שלי כבר בקוד):

```
seg000:7C17 ;
seg000:7C17 ; Long jump to 00:061C, changing CS and IP by RETF
seg000:7C17 ;
seg000:7C17                 push    ax
seg000:7C18                 push    61Ch
seg000:7C1B                 retf
seg000:7C1B ; ---------------------------------------------------------
```

ניתוח:

- יש פה טריק נחמד. חדי העין מביניכם שמו לב שאיפסנו את הסגמנטים המשומשים ביותר ב- real mode, אבל השמטנו את CS. שימו לב שבדומה לאוגר IP, לא ניתן לכתוב ישירות אל אוגר CS. בדרך כלל משתמשים ב-JMP כדי לשנות את אוגר IP, אבל מה עושים עם CS?
  - דוחפים את AX למחסנית. נזכור ש-AX מחזיק 0 בשלב זה בתכנית.
  - דוחפים את הקבוע 0x61C.
  - מבצעים RETF. הפקודה RETF היא קיצור של RETurn-Far, ולמעשה מבצעת חזרה מ- far call. ב-call רגיל נדחף ערכו של IP למחסנית. וכאשר הפונקציה מסתיימת מבוצע RET שלמעשה מבצע pop IP. ב-far call נדחפים CS ו-IP (בסדר זה), ולכן כאשר מבוצע RETF אז מבוצעות למעשה הפקודות pop IP ו-pop CS. זה אומר שלאחר ה-RETF, ערכו של CS יהיה 0 וערכו של IP יהיה 0x61C, כלומר, אנחנו נהיה בכתובת 00:061C.
- למה זה מעניין? כאן למעשה עברה השליטה אל העותק של ה-MBR, כפי שצפינו. בנוסף, הקוד ממשיך מאותו Offset! שימו לב שאילו ה-RETF היה NOP, היינו בכתובת 0x7C1C, כלומר, ב-offset של 0x1C מתחילת ה-MBR. זה מתאים בדיוק לכתובת 0x61C, שנמצאת באותו offset מתחילת העותק של ה-MBR.

- הסיבה לכך ש-IDA לא הצליחה להבין שלאחר ה-RETF יש קוד היא ש-IDA לא יכלה לצפות שביצוע RETF יקפוץ "כאילו" לשורה הבאה. למעשה, אפילו אם IDA הבינה ש-AX יכיל 0 תמיד ושתמיד ה-RETF יקפוץ לכתובת 00:061C, לא הייתה ל-IDA כל דרך לדעת ששם נמצא עותק ה-MBR.
- לצורך נוחות, נבצע rebase שוב לכתובת 0x600. זה יעזור לנו לראות דברים באופן נכון יותר, החל מרגע זה בתכנית. בנוסף, נמיר את החלק שמתחת ל-RETF לקוד (לחיצה על מקש C).

חלק ג': מחיצות או לא להיות

ספוילר: חלק זה של התכנית יתמקד בפרסור ראשוני של ה-partition table. כעת (באיחור רב!) נסביר על הפורמט המצופה מ-MBR. שוב נדגיש כי זה פורמט שמצופה מ-MBR, אבל בפועל כל דיסק שהסקטור הראשון בו נגמר ב-0xAA55 הוא bootable וזהו.

הערת צד: מאז 2010 ישנו תקן חדש שנקרא GPT (קיצור של GUID Partition Table), שלא משומש בגרסת מערכת ההפעלה שעליה מוצגת הכתבה. שווה לקרוא על זה באינטרנט בכל מקרה.

מבנה MBR קלאסי נראה כך:

Offset (HEX)	Description	Size (bytes)
0x0000	Bootstrap code area	**446**
0x01BE	Partition entry #1	**16**
0x01CE	Partition entry #2	**16**
0x01DE	Partition entry #3	**16**
0x01EE	Partition entry #4	**16**
0x01FE	Boot signature (0xAA55)	**2**

אפשר לראות שמגיעים בסוף ל-0x200 בתים ושאכן מסיימים ב-0xAA55. מצופה מ-MBR להחזיק ארבע רשומות עבור מחיצות, כאשר כל רשומה תופסת 16 בתים.

הערת צד: למעשה, יש פורמטים שונים ל-MBR, אבל אצל כולם מנוצל שטח מאזור הקוד עבור נתונים נוספים כגון timestamp, חתימת דיסק ועוד. בכל מקרה, בכל המבנים, מיקום ה-partition entries נשאר באותו offset, ולכן כולם compatible למבנה שהוצג כאן (שהוא ה-MBR ה-"קלאסי").

אם כן, כיצד נראית רשומה?

Offset (HEX)	Description	Size (bytes)
0x0000	Status byte (MSB = 1 means active, 0 means inactive, other options are invalid).	**1**
0x0001	CHS address of first absolute sector, by the order to Head, Sector, Cylinder.	**3**
0x0004	Partition type	**1**
0x0005	CHS address of last absolute sector	**3**
0x0008	LBA of first absolute sector	**4**
0x000C	Number of sectors in partition	**4**

לאחר שמובן לנו מבנה הרשומות, נתבונן סוף כל סוף בקוד:

```
seg000:061C                 sti
seg000:061D ;
seg000:061D ; Make BP point to the first partition entry
seg000:061D ; Partition table (which contains up to 4 entries) is located at offset 01BE from the MBR
seg000:061D ; Since MBR was loaded to 7C00 and relocated to 0600, the partition table is at 07BE
seg000:061D ; Make CX be the maximum number of partition entries
seg000:061D ;
seg000:061D                 mov     cx, 4
seg000:0620                 mov     bp, 7BEh
seg000:0623 ;
seg000:0623 ; Check out partitions
seg000:0623 ; First instruction is CMP because we would like to check the MSB as well as the zero flag
seg000:0623 ;
seg000:0623
seg000:0623 lblNextPartitionEntry:                  ; CODE XREF: seg000:0630↓j
seg000:0623                 cmp     byte ptr [bp+0], 0
seg000:0627                 jl      short lblFoundBootablePartition
seg000:0629 ;
seg000:0629 ; If the MSB is zero but the status byte isn't zero, the entry is corrupted
seg000:0629 ;
seg000:0629                 jnz     lblInvalidPartitionTable
seg000:062D ;
seg000:062D ; Move to the next entry (each entry is 0x10 bytes)
seg000:062D ;
seg000:062D                 add     bp, 10h
seg000:0630                 loop    lblNextPartitionEntry
seg000:0632 ;
seg000:0632 ; In this point, we've exhausted the partition table and didn't find a bootable partition
seg000:0632 ; Reboots the machine using INT18
seg000:0632 ;
seg000:0632                 int     18h             ; TRANSFER TO ROM BASIC
seg000:0632                                         ; causes transfer to ROM-based BASIC (IBM-PC)
seg000:0632                                         ; often reboots a compatible; often has no effect at all
```

ניתוח:

- בשורה הראשונה מבצעים STI, שמעלה את ה-interrupt flag. דגל זה קובע האם תתרחשנה פסיקות.
- לאחר מכן, שמים בתוך BP את הערך 0x07BE נזכור שה-MBR נמצא ב-0x0600, וזה אומר למעשה ש-BP מצביע על ה-Partition entry הראשון. בנוסף, ישנה השמה של 4 ל-CX. בדרך כלל CX משומש למנייה. המספר 4 צריך להיות מובן גם - הוא מספר ה-entries האפשריים.
- השוואה של הבית הראשון שמוצבע על ידי BP לאפס (ה-status byte). זאת דרך יעילה לבדוק במכה אחת גם את ה-MSB וגם האם זה אפס או לא, כי הדגלים המושפעים הם גם ZF, גם OF וגם SF. אם ה-MSB היה דלוק, אז ה-JL יקפוץ, ולכן נסיק שזה מקרה שבו מצאנו bootable partition.
- אם ה-JNZ קופץ זה אומר שגם ה-MSB היה כבוי וגם ה-status byte אינו אפס. כפי שצויין, זהו מצב לא חוקי ולכן נקפוץ לאזור שמתבכיין על זה שה-partition table לא תקין.
- שתי השורות הבאות מדלגות ל-partition entry הבא: הגדלת BP ב-0x10 (גודל entry) וביצוע LOOP, מה שמקטין את ערך CX וקופץ אם הוא איננו אפס. זה אומר שנוכל לבצע 4 פעמים את הסיפור הזה, מה שמתאים לחלוטין למספר הרשומות.
- השורה הבאה מבצעת INT 18. מכיוון ש-IDA הוא אחלה כלי - כתוב לנו כבר שמבוצע reset למכונה. עם זאת, לפעמים IDA לא יכולה לדעת מה יתבצע, למשל כאשר אוגר מסויים משפיע על מה הפסיקה

עושה. לכן, זה זמן מצויין להפנות אל Ralph Brown's Interrupt List או RBIL בקיצור. אין שם הכל, אבל רוב הדברים נמצאים.

סיכום: לאחר פרסור ה-partition table, אוגר BP אמור להצביע ל-entry המתאים ב-MBR המועתק. זה בדיוק המצב של התוכנית בשורה לאחר ה-INT 18 (האזור שציינתי בתור lblFoundBootablePartition).

חלק ד': LBA או CHS?

להלן הקוד (הערות שלי כבר בפנים):

```
seg000:0634 ;
seg000:0634 ; BP (which points to the bootable partition entry) is used to save various data:
seg000:0634 ;     Offset=0x00    The drive number (DL)
seg000:0634 ;     Offset=0x10    Whether READ extensions are supported or not
seg000:0634 ;     Offset=0x11    The number of read tries
seg000:0634 ; Backup BP on the stack since we'll use an interrupt, which doesn't save BP
seg000:0634 ;
seg000:0634
seg000:0634 lblFoundBootablePartition:              ; CODE XREF: seg000:0627↑j
seg000:0634                                         ; seg000:06AE↓j
seg000:0634                 mov     [bp+0], dl
seg000:0637                 push    bp
seg000:0638                 mov     byte ptr [bp+11h], 5
seg000:063C                 mov     byte ptr [bp+10h], 0
seg000:0640 ;
seg000:0640 ; Check for READ extension availability for the drive number in DL
seg000:0640 ;
seg000:0640                 mov     ah, 41h ; 'A'
seg000:0642                 mov     bx, 55AAh
seg000:0645                 int     13h             ; DISK - Check for INT 13h Extensions
seg000:0645                                         ; BX = 55AAh, DL = drive number
seg000:0645                                         ; Return: CF set if not supported
seg000:0645                                         ; AH = extensions version
seg000:0645                                         ; BX = AA55h
seg000:0645                                         ; CX = Interface support bit map
seg000:0647 ;
seg000:0647 ; Restore BP
seg000:0647 ;
seg000:0647                 pop     bp
```

ניתוח:

- ניתן לראות כי בארבע השורות הראשונות משתמשים ב-BP כבסיס לאחסון מידע. נשמור את מספר הכונן ב-offset של אפס, ונשמור את המספר 5 ואת המספר 0 ב-offset-ים 0x11 ו-0x10, בהתאמה. כבר ניתחתי מה המשמעות של ה-5 וה-0 האלה, אבל כרגע נזכור פשוט שזה מה שאוחסן, בלי להבין מדוע. בנוסף, דוחפים את BP למחסנית. מדוע? מי שיציץ אחר כך יגלה שמבצעים פסיקה - ולצערנו, פסיקות לא מתחייבות לשמר את אוגר BP. לכן, חשוב לשמור אותו במקום זמני, והמחסנית היא אחלה מקום לזה.
- שלוש השורות הבאות מבצעות הכנה לפסיקה ואת הפסיקה עצמה. IDA מבינה ויודעת לתאר לנו בדיוק מה הפסיקה עושה, אבל אפשר גם לחפש ב-RBIL כדי להבין לעומק. אציין במקרה זה שפסיקה זו (read extensions) למעשה בודקת האם הדיסק תומך בקריאה מסוג LBA או לא (ואז נקרא על ידי CHS).

- בסוף משחזרים את BP מהמחסנית.
- שווה לשים לב ש-IDA אומרת לנו מה אמור לחזור גם: AH יחזיק מספר גרסא, BX יחזיק 0xAA55, אוגר CX יחזיק דגלים שונים והכי חשוב - CF יהיה דלוק אם אין תמיכה (ואז צריך לדבר CHS).

הערת צד: CHS זה קיצור של Cylinder, Head, Sector וזוהי השיטה הסטנדרטית הישנה לבצע קריאה של דיסק. LBA היא שיטה חדשה יותר (Logical Block Addressing) ונותנת להתייחס לדיסק כאילו הוא מערך של בתים. ישנן דרכים להמרה בין שיטה אחת לשנייה - עוד בויקיפדיה על הנושא.

נמשיך לחלק הבא שבודק האם ניתן לקרוא CHS או LBA:

```
seg000:0648 ;
seg000:0648 ; Carry flag is set if READ extensions are not available
seg000:0648 ; BX should be 0xAA55 on a successful call
seg000:0648 ; LSB of CX should indicate that the support bitmap is suitable
seg000:0648 ; If all of these apply - mark that READ extensions are available in the (BP+0x10) byte
seg000:0648 ;
seg000:0648                 jb      short lblPerformReading
seg000:064A                 cmp     bx, 0AA55h
seg000:064E                 jnz     short lblPerformReading
seg000:0650                 test    cx, 1
seg000:0654                 jz      short lblPerformReading
seg000:0656                 inc     byte ptr [bp+10h]
seg000:0659 ;
seg000:0659 ; Backup all general-purpose registers
seg000:0659 ;
seg000:0659
seg000:0659 lblPerformReading:                      ; CODE XREF: seg000:0648↑j
seg000:0659                                         ; seg000:064E↑j ...
seg000:0659                 pushad
seg000:065B ;
seg000:065B ; Check if READ extensions are available
seg000:065B ; If not - we read using CHS
seg000:065B ; If we can use READ extensions - we use LBA
seg000:065B ;
seg000:065B                 cmp     byte ptr [bp+10h], 0
seg000:065F                 jz      short lblReadCHS
```

ניתוח:

- הבלוק הראשון בודק האם יש תמיכה או לא. הוא יציין זאת ב-offset של 0x10 מ-BP (זה היה הקטע בו התפקיד של ה-byte הזה מובן לנו). הוא עושה זאת על ידי הבדיקות הבאות:
  - בדיקת CF על ידי ביצוע JB.
  - בדיקה כי אכן BX מכיל 0xAA55.
  - בדיקת ה-LSB של CX. מוזמנים לבדוק ב-RBIL מה המשמעות שלו.
  - הבלוק הבא מבצע דחיפה של כל האוגרים הכלליים על ידי PUSHAD.
- לבסוף, קופצים אל lblReadCHS אם אי אפשר לקרוא LBA. אחרת, ממשיכים הלאה אל 0x0661 ושם נצפה לבצע קריאת LBA.

סיכום: הגענו למצב שבו אנחנו מוכנים לבצע את הקריאה הבאה - LBA או CHS.

חלק ה': קריאה

להלן קטע הקוד הבא:

```
seg000:0661 ;
seg000:0661 ; Build a disk address packet on the stack
seg000:0661 ;     00    BYTE     Size of packet (0x10 or 0x18)
seg000:0661 ;     01    BYTE     Reserved (0)
seg000:0661 ;     02    WORD     Number of blocks to transfer (1 block)
seg000:0661 ;     04    DWORD    Transfer buffer (00:7C00)
seg000:0661 ;     08    QWORD    Starting absolute block number (bp+8 is the LBA of first absolute sector)
seg000:0661 ;
seg000:0661                 push    large 0
seg000:0667                 push    large dword ptr [bp+8]
seg000:066B                 push    0
seg000:066E                 push    7C00h
seg000:0671                 push    1
seg000:0674                 push    10h
seg000:0677 ;
seg000:0677 ; Perform the extended READ in LBA form
seg000:0677 ; DL is the drive number
seg000:0677 ;
seg000:0677                 mov     ah, 42h ; 'B'
seg000:0679                 mov     dl, [bp+0]
seg000:067C                 mov     si, sp
seg000:067E                 int     13h             ; DISK - IBM/MS Extension - EXTENDED READ (DL - drive, DS:SI - disk address packe
seg000:0680 ;
seg000:0680 ; Backup the flags on AL
seg000:0680 ; Dispose of the disk address packet from the stack
seg000:0680 ; Restore flags from AL
seg000:0680 ;
seg000:0680                 lahf
seg000:0681                 add     sp, 10h
seg000:0684                 sahf
seg000:0685                 jmp     short lblPerformPostReadValidations
```

ניתוח:

- החלק הראשון מייצר disk address packet על המחסנית. מה זה disk address packet? אפשר לראות שאחר כך קוראים ל-int 13 עם AH=0x42. חיפוש ב-RBIL מראה שמדובר בקריאה מהדיסק, בת'כלס מדובר בקריאת LBA. תיעדתי את המבנה, אבל אתאר כאן לצורך השלמות:

Offset (HEX)	Description	Size (bytes)
0x0000	Size of packet (0x10 or 0x18)	**1**
0x0001	Reserved (0)	**1**
0x0002	Number of blocks to transfer	**2**
0x0004	Transfer buffer pointer	**4**
0x0008	Starting absolute address (LBA)	**8**

אפשר לראות שאכן דוחפים את המידע המתאים - אבל בסדר הפוך! למה? כי המחסנית גדלה לכיוון כתובות נמוכות. נראה שנבצע קריאה אל 00:7C00 של בלוק אחד של הכתובת המוצבעת על ידי ה-

DWORD שנמצא ב-BP+8. מכיוון ש-BP מצביע על ה-partition entry המתאים, זה בדיוק כתובת ההתחלה של ה-LBA.

- החלק השני מבצע ממש את הקריאה. אין המון מה לפרט כאן.
- החלק השלישי מנקה את המחסנית מה-disk address packet על ידי הוספה של 0x10 ל-SP. השימוש ב-LAHF ו-SAHF נעשה כדי לא לגרום לדגלים להשתנות לאחר פעולה ה-ADD. לבסוף, מתבצעת קפיצה בלתי מבוקרת (JMP), מכיוון שבכתובת הבאה (0x0687) ממומשת קריאת CHS.

בשלב זה ננתח גם את קריאת ה-CHS. שימו לב שהסיפור הולך להיות די דומה:

```
seg000:0687 ; ---------------------------------------------------------------------------
seg000:0687 ;
seg000:0687 ; Read CHS
seg000:0687 ; AL = 0x01 (the number of sectors to read)
seg000:0687 ; AH = 0x02 (read sectors)
seg000:0687 ; ES:BX = 00:7C00 (the buffer to fill)
seg000:0687 ; DL = drive number, the disk number previously saved in the (BP+0) byte
seg000:0687 ; DH = head, from the partition entry
seg000:0687 ; CL = sector, from the partition entry
seg000:0687 ; CH = cylinder, from the partition entry
seg000:0687 ;
seg000:0687
seg000:0687
seg000:0687 lblReadCHS:                                 ; CODE XREF: seg000:065F↑j
seg000:0687                 mov     ax, 201h
seg000:068A                 mov     bx, 7C00h
seg000:068D                 mov     dl, [bp+0]
seg000:0690                 mov     dh, [bp+1]
seg000:0693                 mov     cl, [bp+2]
seg000:0696                 mov     ch, [bp+3]
seg000:0699                 int     13h             ; DISK - READ SECTORS INTO MEMORY
seg000:0699                                         ; AL = number of sectors to read, CH = track, CL = sector
seg000:0699                                         ; DH = head, DL = drive, ES:BX -> buffer to fill
seg000:0699                                         ; Return: CF set on error, AH = status, AL = number of sectors read
```

ניתוח:

- מספר הסקטורים לקריאה הוא 1, לתוך כתובת 00:7C00. שימו לב לשימוש ב-partition entry - שואבים מהשדות בו את ה-head, cylinder וה-sector המתאימים. שימו לב גם לשימוש המחוכם באוגר AX כדי לשים ערכים גם ב-AH וגם ב-AL במכה אחת.
- דבר נחמד ששווה לשים לב אליו: בשתי הקריאות, CF עולה אם הייתה שגיאה. לכן, שני סוגי הקריאות מופנים לאותה הכתובת, שתבצע קריאה של ה-CF כדי לבדוק האם הייתה שגיאת קריאה או לא.

עכשיו נגיע אל החלק המנתח את הקריאה:

```
seg000:069B lblPerformPostReadValidations:          ; CODE XREF: seg000:0685↑j
seg000:069B                 popad
seg000:069D ;
seg000:069D ; Carry flag is set on error
seg000:069D ; If an error occured, decrease the number of tries in the (BP+0x11) byte
seg000:069D ;
seg000:069D                 jnb     short lblVbrLoadedSuccessfully
seg000:069F                 dec     byte ptr [bp+11h]
seg000:06A2                 jnz     short lblResetDiskAndRetryLoadingVBR
seg000:06A4 ;
seg000:06A4 ; We've exhaused the number of READ attempts
seg000:06A4 ; If DL was 0x80, then we present "error loading operating system" and quit
seg000:06A4 ; Otherwise, we try again one more time with DL=0x80
seg000:06A4 ;
seg000:06A4                 cmp     byte ptr [bp+0], 80h ; 'Ä'
seg000:06AB                 jz      lblErrorLoadingOperatingSystem
seg000:06AC                 mov     dl, 80h ; 'Ä'
seg000:06AE                 jmp     short lblFoundBootablePartition
seg000:06B0 ; ---------------------------------------------------------------------------
seg000:06B0 ;
seg000:06B0 ; Backup BP
seg000:06B0 ; Reset the disk number and restore BP
seg000:06B0 ; This is done because interrupts don't maintain BP
seg000:06B0 ; Then, we perform reading again
seg000:06B0 ;
seg000:06B0
seg000:06B0 lblResetDiskAndRetryLoadingVBR:         ; CODE XREF: seg000:06A2↑j
seg000:06B0                 push    bp
seg000:06B1                 xor     ah, ah
seg000:06B3                 mov     dl, [bp+0]
seg000:06B6                 int     13h             ; DISK - RESET DISK SYSTEM
seg000:06B6                                         ; DL = drive (if bit 7 is set both hard disks and floppy disks reset)
seg000:06B8                 pop     bp
seg000:06B9                 jmp     short lblPerformReading
```

ניתוח:

- מבצעים POPAD כדי לשחזר את הרגיסטרים ששמרנו לפני כן (ממש לפני תחילת הקריאה). יש לשים לב שזה לא משנה את אוגר הדגלים, אז CF יהיה דלוק עדיין אם הייתה שגיאה בקריאה.
- בודקים את CF על ידי JNB. במקרה ש-CF כבוי, נקפוץ את lblVbrLoadedSuccessfully. אחרת, נפחית את הערך של הבית ב-offset של 0x11 מ-BP ונבדוק האם הוא מתאפס. אם לא, נבצע ניסיון קריאה נוסף. זה בדיוק הבית ששמנו בו את הערך 5 בהתחלה, ולכן הוא מציין את מספר ניסיונות הקריאה המקסימאלי.
- אם מראש ניסינו לקרוא מ-device מספר 0x80 (שנשמר בבית הראשון המוצבע על ידי BP), אז נקפוץ לכתובת שלה קראתי lblErrorLoadingOperatingSystem. אחרת, נבצע ניסיון קריאה נוסף מתוך device מספר 0x80. מה זה ה-device הזה? לקח לי זמן למצוא את הסיבה לכך, אבל מסתבר שיש BIOS-ים באגיים שלא מעבירים נכון את ה-device ID לאוגר DL ב-boot, ולכן באופן hard-coded מבצעים ניסיון קריאה מההארד-דיסק הראשון.
- בחלק האחרון בקטע זה נבצע ריסט לדיסק. לכאן הגענו למעשה כחלק מניסיון הקריאה הנוסף (אחד מתוך חמישה, כאמור). אין כאן משהו מאד מיוחד, האסמבלי מדבר בעד עצמו.

חלק ו': המקלדת?

בחלק זה נסטה טיפה מהקוד הראשי ונקפוץ אל הפרוצדורה היחידה שקיימת (0x0756 לאחר ההעתקה). הסיבה לכך ש-IDA זיהתה שזו פרוצדורה היא שמבצעים call לשם (וכמו כן היא מסתיימת ב-retn).

```
seg000:0756
seg000:0756 ; =============== S U B R O U T I N E =======================================
seg000:0756
seg000:0756
seg000:0756 WaitForKeyboardInput proc near          ; CODE XREF: seg000:06C6↑p
seg000:0756                                         ; seg000:06D0↑p ...
seg000:0756                 sub     cx, cx
seg000:0758
seg000:0758 lblKeyboardPoll:                        ; CODE XREF: WaitForKeyboardInput+8↓j
seg000:0758                 in      al, 64h         ; 8042 keyboard controller status register
seg000:0758                                         ; 7:  PERR    1=parity error in data received from keyboard
seg000:0758                                         ;    +----------- AT Mode ----------+------------ PS/2 Mode ------------+
seg000:0758                                         ; 6: |RxTO    receive (Rx) timeout  | TO      general timeout (Rx or Tx)|
seg000:0758                                         ; 5: |TxTO    transmit (Tx) timeout | MOBF    mouse output buffer full  |
seg000:0758                                         ;    +-----------------------------+-----------------------------------+
seg000:0758                                         ; 4:  INH     0=keyboard communications inhibited
seg000:0758                                         ; 3:  A2      0=60h was the port last written to, 1=64h was last
seg000:0758                                         ; 2:  SYS     distinguishes reset types: 0=cold reboot, 1=warm reboot
seg000:0758                                         ; 1:  IBF     1=input buffer full (keyboard can't accept data)
seg000:0758                                         ; 0:  OBF     1=output buffer full (data from keyboard is available)
seg000:075A                 jmp     short $+2
seg000:075C                 and     al, 2
seg000:075E                 loopne  lblKeyboardPoll
seg000:0760                 and     al, 2
seg000:0762                 retn
seg000:0762 WaitForKeyboardInput endp
```

מה קורה בחלק הזה?

- בתחילת הפרוצדורה מבצעים איפוס של CX (נדבר על זה בקרוב). לאחר מכן יש לולאה:
  - השגת הסטטוס של ה-keyboard controller. למזלנו, IDA מכירה וידעה לפרט את סידור הביטים. הסבר נוסף כאן: http://www.computer-engineering.org/ps2keyboard.
  - קפיצה 2 בתים קדימה. גם על זה נדבר בקרוב.
  - בדיקה האם ביט מספר 1 (ה-IBF) דלוק. ביט זה מציין האם ניתן לבצע OUT על המקלדת.
  - מורידים את הערך של CX באחד ובודקים האם הוא אפס. אם הוא לא אפס ואם תוצאת החישוב הקודמת לא הייתה אפס, מבצעים איטרציה נוספת.
- בסוף (0x0760) מבצעים שוב and AL, 2 כדי להחזיר תוצאה באוגר AL. זה אומר שהתוצאה "טובה" אם ה-IBF היה 0, כלומר, הפרוצדורה מחזירה 0 בהצלחה וערך אחר בכשלון (בת'כלס: תחזיר 2).
- ביצוע retn מחזיר אותנו חזרה אל המקום שביצע call.
- מדוע מבצעים איפוס של CX בהתחלה? כדי לבצע את הלולאה 65536 פעמים! המתכנתת בנתה כאן על ה-wraparound של CX ועל כך ש-loopne מפחית קודם את CX ורק אז בודק את ערכו.
- מדוע מבצעים JMP $+2, פעולה שנראית חסרת תועלת לגמרי? ישנם שני הבדלים שעליהם חשבתי בין סתם NOP לבין JMP שכזה:
  - תזמון: לוקח יותר clock cycles לבצע JMP מאשר NOP.
  - ניקוי תור ה-prefetched instructions: ביצוע JMP אמור לנקות את כל ה-instructions שעברו prefetching ב-pipeline של המעבד, בעוד ל-NOP אין side effect כזה.

- ההשערה שלי (אני לא בטוח עד עכשיו שזה נכון) שמדובר בעניין timing. יכול להיות שהמטרה הייתה לבצע סוג של "sleep" מתוך מחשבה שאולי בזמן ביצוע sleep שכזה הבאפר של המקלדת יתרוקן.

עכשיו אפשר לחזור אל הקוד הראשי (0x06BB). אם ה-flow עד עכשיו היה תקין, אנחנו אמורים להיות במיקום הבא:

```
seg000:06BB ; ---------------------------------------------------------------------------
seg000:06BB ;
seg000:06BB ; The VBR is supposed to end with 0xAA55
seg000:06BB ; Since it's loaded to 00:7C00, the last word is at 00:7DFE
seg000:06BB ;
seg000:06BB
seg000:06BB lblVbrLoadedSuccessfully:                   ; CODE XREF: seg000:069D↑j
seg000:06BB                 cmp     word ptr ds:7DFEh, 0AA55h
seg000:06C1                 jnz     short lblMissingOperatingSystem
```

הסברים:

- ה-VBR נטען אל 00:7C00, ולפיכך הוא אמור להסתיים ב-0xAA55.
- אם לא, קופצים אל אזור אחר ("missing operating system").

לאחר מכן מבוצעות עוד כמה שורות פשוטות:

```
seg000:06C3 ;
seg000:06C3 ; Save [BP+0]
seg000:06C3 ;
seg000:06C3                 push    word ptr [bp+0]
seg000:06C6 ;
seg000:06C6 ; Wait for keyboard availability
seg000:06C6 ;
seg000:06C6                 call    WaitForKeyboardInput
seg000:06C9                 jnz     short lblKeyboardNotAvailable
```

הסברים:

- שומרים את הערך ב-[bp]. להזכירכם, שם נשמר הערך המקורי של DL, שהיה מספר ה-Drive.
- קוראים אל הפרוצדורה שלנו. היא מחזירה 0 במקרה של הצלחה (ה-input buffer פנוי).
- ביצוע JNZ אל מקום אחר (ת'כלס, עדיין happy flow).

מכאן נתחיל לבצע מניפולציות על המקלדת. אחד המקומות הטובים ביותר לקרוא על כך הוא בפרק 20 של Art Of Assembly, כאן:

https://courses.engr.illinois.edu/ece390/books/artofasm/CH20/CH20-2.html.

עבור אנשי ה-TLDR, אתמצת:

- ישנם 2 צ'יפים לעבודה עם מקלדת: אחד עם לוח האם והשני במקלדת.
- ניתן לדבר עם הצ'יפ שממוקם בלוח האם עם פורט 0x64, והוא גם ידוע כ-Control port.
- עם הצ'יפ של המקלדת מדברים בפורט 0x60, והוא ידוע כ-Data port.
- ספציפית, הפקודה שתעניין אותנו בעיקר היא 0xD1 על ה-control port.

הערת צד: נשים לב שחלק מהפסיקות שמולאו על ידי ה-BIOS "עוטפות" לנו עבודה מול מקלדת (למעשה, ראינו כבר אחת כזו בפרוצדורה שלנו). ישנן פסיקות ש-BIOS לא עוטף, ולכן צריך לדבר עם המקלדת ישירות.

חמושים בידע על מקלדות, אפשר לנתח את השורות הבאות:

```
seg000:06CB ;
seg000:06CB ; Keyboard is ready
seg000:06CB ; Write a data byte to the keyboard in order to enable A20 gate.
seg000:06CB ;
seg000:06CB                 cli
seg000:06CC                 mov     al, 0D1h ; '_'
seg000:06CE                 out     64h, al         ; 8042 keyboard controller command register.
seg000:06CE                                         ; Write output port (next byte to port 60h):
seg000:06CE                                         ; 7:  1=keyboard data line pulled low (inhibited)
seg000:06CE                                         ; 6:  1=keyboard clock line pulled low (inhibited)
seg000:06CE                                         ; 5:  enables IRQ 12 interrupt on mouse IBF
seg000:06CE                                         ; 4:  enables IRQ 1 interrupt on keyboard IBF
seg000:06CE                                         ; 3:  1=mouse clock line pulled low (inhibited)
seg000:06CE                                         ; 2:  1=mouse data line pulled low (inhibited)
seg000:06CE                                         ; 1:  A20 gate on/off
seg000:06CE                                         ; 0:  reset the PC (THIS BIT SHOULD ALWAYS BE SET TO 1)
seg000:06D0                 call    WaitForKeyboardInput
seg000:06D3                 mov     al, 0DFh ; '_'
seg000:06D5                 out     60h, al         ; 8042 keyboard controller data register.
seg000:06D7                 call    WaitForKeyboardInput
seg000:06DA                 mov     al, 0FFh
seg000:06DC                 out     64h, al         ; 8042 keyboard controller command register.
seg000:06DC                                         ; Pulse output port.
seg000:06DC                                         ; Bits 0-3 indicate ports to pulse.
seg000:06DE                 call    WaitForKeyboardInput
seg000:06E1                 sti
```

הסברים:

- ביצוע CLI למניעת פסיקות.
- כפי שהבטחתי, כתיבת 0xD1 אל ה-control port. הבית הבא שייכתב אל ה-data port יקבל את המשמעות המופיעה ב-IDA. נשים לב שלאחר מכן כותבים 0xDF על ה-data port. הייצוג הבינארי של 0xDF הוא 11011111, מה שאומר שמעלים את כל הדגלים מלבד IRQ 12 interrupt on mouse IBF.

ישנם כמה דברים שימושיים, כאשר גולת הכותרת היא ביט 1 (אם ה-LSB הוא 0), שמציין שמדליקים את ה-A20 gate.

- מהו אותו A20 gate? זהו דגל שממוען בכלל ל-memory controller (!) שקובע האם ניתן לפנות לכתובות זיכרון גבוהות (מעל 20 ביט).
- כידוע, 20 ביט של זיכרון שווי ערך למגה של מרחב-זיכרון. מכיוון שב-real mode פונים עם אוגר בסיס ו-offset (למשל: ES:DI) ומכיוון שכל אוגר הוא ברוחב 16 ביט, ניתן לגשת לכאורה לכתובות גבוהות יותר ממגה (הכתובת המקסימאלית שניתן להשיג כך היא 0x10FFEF, בעוד 20 מגה של מרחב זיכרון מאפשרים גישה עד 0xFFFFF).
- בעבר היו למחשב רק 20 קווים למיעון כתובת, ולכן מה שהיה קורה במקרה של כתובת גבוהה הוא wraparound (למשל, פנייה את כתובת 0x10FFEF הייתה זהה לכתובת 0xFFEF), ומתכנתים ניצלו את ה-wraparound הזה. לכן היה צורך לשמור על תמיכה לאחור כאשר נוספו כתובות זיכרון גבוהות (במחשבי 286), וה-wraparound נשמר אלא אם כן דגל ה-A20 דלוק.

- לאחר מכן כותבים 0xFF על ה-control port ומחזירים פסיקות. כתיבת 0xFF מסמנת למקלדת שהיא יכולה "לשתות" את הפקודות שנכתבו לה (דמיינו סוג של commit, אם תרצו).
- כמובן, לאחר כל פנייה למקלדת ממתינים שהבאפר של המקלדת יתרוקן.
- לאחר מכן ניתן לראות שקפצנו למקום שאליו היינו קופצים אם מראש הפרוצדורה שלנו הייתה נכשלת. מכאן ניתן ללמוד שהדלקת ה-A20 היא best-effort. לכן נשנה את השם lblKeyboardNotAvailable למשהו יותר מתאים.

הערת צד: מדוע דווקא ה-A20 gate עובר דרך המקלדת? התשובה היא שבמקרה למקלדות 8042 היה pin פנוי, אז "התעלקו" עליו לשליטה על שער ה-AND שהיה אחראי לסיפור.

הערת צד נוספת: BIOS-ים מודרניים יותר מממשים פסיקה 15 int עם AX=240X לשליטה ב- A20 gate. מידע נוסף ניתן למצוא ב-RBIL ובאינטרנט.

חלק ז': אל ה-VBR!

להלן הקוד (קצת ארוך הפעם):

```
seg000:06E2
seg000:06E2 lblLogTPM:                              ; CODE XREF: seg000:06C9↑j
seg000:06E2                 mov     ax, 0BB00h
seg000:06E5                 int     1Ah             ; Trusted Computing Group call - TCG_StatusCheck
seg000:06E5                                         ; Return: EAX = 0 if supported
seg000:06E5                                         ; EBX = 41504354h ('TCPA')
seg000:06E5                                         ; CH:CL = TCG BIOS Version
seg000:06E5                                         ; EDX = BIOS TCG Feature Flags
seg000:06E5                                         ; ESI = Pointer to Event Log
seg000:06E5                                         ;
seg000:06E7                 and     eax, eax
seg000:06EA                 jnz     short lblMoveControlToVBR
seg000:06EC                 cmp     ebx, 'APCT'
seg000:06F3                 jnz     short lblMoveControlToVBR
seg000:06F5                 cmp     cx, 102h
seg000:06F9                 jb      short lblMoveControlToVBR
seg000:06FB ;
seg000:06FB ; Change registers by pushing and then performing POPAD
seg000:06FB ; Order: EDI, ESI, EBP, <ignored>, EBX, EDX, ECX, EAX
seg000:06FB ;
seg000:06FB                 push    large 0BB07h    ; (EAX)
seg000:0701                 push    large 200h      ; (ECX)
seg000:0707                 push    large 8         ; (EDX)
seg000:070D                 push    ebx             ; (EBX)
seg000:070F                 push    ebx             ; (ignored)
seg000:0711                 push    ebp             ; (EBP)
seg000:0713                 push    large 0         ; (ESI)
seg000:0719                 push    large 7C00h     ; (EDI)
seg000:071F                 popad
seg000:0721 ;
seg000:0721 ; Change ES to 0
seg000:0721 ;
seg000:0721                 push    0
seg000:0724                 pop     es
seg000:0725 ;
seg000:0725 ; 0xBB07 = TCG_CompactHashLogExtendEvent
seg000:0725 ; ES:DI = Data buffer to be hashed
seg000:0725 ; ECX = Data buffer length
seg000:0725 ; EDX = Event number (PCR number)
seg000:0725 ;
seg000:0725                 int     1Ah             ; Trusted Computing Group call - TCG_StatusCheck
seg000:0725                                         ; Return: EAX = 0 if supported
seg000:0725                                         ; EBX = 41504354h ('TCPA')
seg000:0725                                         ; CH:CL = TCG BIOS Version
seg000:0725                                         ; EDX = BIOS TCG Feature Flags
seg000:0725                                         ; ESI = Pointer to Event Log
seg000:0725                                         ;
seg000:0727
seg000:0727 lblMoveControlToVBR:                    ; CODE XREF: seg000:06EA↑j
seg000:0727                                         ; seg000:06F3↑j ...
seg000:0727                 pop     dx
seg000:0728                 xor     dh, dh
seg000:072A                 jmp     far ptr 0:7C00h
seg000:072F ; ---------------------------------------------------------------------------
seg000:072F ;
seg000:072F ; Restart the machine
seg000:072F ;
seg000:072F                 int     18h             ; TRANSFER TO ROM BASIC
seg000:072F                                         ; causes transfer to ROM-based BASIC (IBM-PC)
seg000:072F                                         ; often reboots a compatible; often has no effect at all
```

הסברים:

- קריאה ל-int 1A עם AX=0xBB00. באופן כללי, כאשר AH=BB וקוראים ל-int 1A - תהיה זו קריאה ל-TPM. לוקח קצת זמן למצוא דברים באתר של ה-Trusted computing group, אבל בסוף מסתדרים. ספציפית, כאשר AL=00 אז בודקים האם יש תמיכה בכלל - אפשר לראות שמצפים לערכי חזרה תקינים ולמספר גרסא 1.2 (זה ה-0x0102 שמשווים אל CX). נחמד לראות שאם לא מסתדר - פשוט קופצים קדימה אל 0x0727.

- ביצוע מלא דחיפות ואז POPAD. אין כאן תחכום גדול מעבר ללקרוא את הסדר של האוגרים שמוציאים מהמחסנית עם הקריאה ל-POPAD.

- שינוי ES להיות 0 וקריאה אל int 1A עם AX=0xBB07. כאן מבאס לראות ש-IDA מטעה אותנו - מדובר בקריאה לפונקציה TCG_CompactHashLogExtendEvent בכלל, והסיבה להטעיה היא ש-IDA לא "מבינה" שאוגר AX השתנה ולכן היא מסתמכת על הערך הקודם שלו. פונקצייה זו מבצעת logging של event אל ה-TPM. מידע נוסף כאן:

https://www.trustedcomputinggroup.org/files/resource_files/CB0B2BFA-1A4B-B294-D0C3B9075B5AFF17/TCG_PCClientImplementation_1-21_1_00.pdf

- ב-0x0727 רואים "העברת אחריות" אל ה-VBR (שנכתב כבר אל 00:7C00): נזכור שדחפנו לפני כן את BP שהכיל בבית הראשון את ה-drive number, אז עכשיו מבצעים POP ומאפסים את DH. בכל מקרה, בתוך DL יהיה ה-drive number. מכאן מבצעים far jump אל ה-VBR.

- יש גם קוד שמבצע int 18 לאחר מכן. זה dead code, ולא מצאתי שום reference אליו.

חלק ח' - השלמות

למעשה סיימנו, אבל יש לנו כמה השלמות לעשות - למעשה, ההדפסה של הודעות השגיאה (במידה והן קרו):

```
seg000:0731
seg000:0731 lblMissingOperatingSystem:              ; CODE XREF: seg000:06C1↑j
seg000:0731                 mov     al, ds:gOffsetTable+2
seg000:0734                 jmp     short lblPrintMessageAndHang
seg000:0736 ; ---------------------------------------------------------------------------
seg000:0736
seg000:0736 lblErrorLoadingOperatingSystem:         ; CODE XREF: seg000:06A8↑j
seg000:0736                 mov     al, ds:gOffsetTable+1
seg000:0739                 jmp     short lblPrintMessageAndHang
seg000:073B ; ---------------------------------------------------------------------------
seg000:073B
seg000:073B lblInvalidPartitionTable:               ; CODE XREF: seg000:0629↑j
seg000:073B                 mov     al, ds:gOffsetTable
seg000:073E ;
seg000:073E ; AL is the offset from 0x0700
seg000:073E ; Make SI = 0x0700+AL
seg000:073E ;
seg000:073E
seg000:073E lblPrintMessageAndHang:                 ; CODE XREF: seg000:0734↑j
seg000:073E                                         ; seg000:0739↑j
seg000:073E                 xor     ah, ah
seg000:0740                 add     ax, 700h
seg000:0743                 mov     si, ax
seg000:0745 ;
seg000:0745 ; Load the next character to AL and increase SI
seg000:0745 ;
seg000:0745
seg000:0745 lblPrintNextChar:                       ; CODE XREF: seg000:0751↓j
seg000:0745                 lodsb
seg000:0746 ;
seg000:0746 ; If a NUL character is found - finish
seg000:0746 ;
seg000:0746                 cmp     al, 0
seg000:0748                 jz      short lblHang
seg000:074A ;
seg000:074A ; Perform TTY write
seg000:074A ; BX = 7 (gray color)
seg000:074A ;
seg000:074A                 mov     bx, 7
seg000:074D                 mov     ah, 0Eh
seg000:074F                 int     10h             ; - VIDEO - WRITE CHARACTER AND ADVANCE CURSOR (TTY WRITE)
seg000:074F                                         ; AL = character, BH = display page (alpha modes)
seg000:074F                                         ; BL = foreground color (graphics modes)
seg000:0751                 jmp     short lblPrintNextChar
seg000:0753 ; ---------------------------------------------------------------------------
seg000:0753 ;
seg000:0753 ; Hang machine (HLT + self JMP)
seg000:0753 ;
seg000:0753
seg000:0753 lblHang:                                ; CODE XREF: seg000:0748↑j
seg000:0753                                         ; seg000:0754↓j
seg000:0753                 hlt
seg000:0754 ; ---------------------------------------------------------------------------
seg000:0754                 jmp     short lblHang
```

הסברים:

- ניתן לראות שיש 3 מקומות שבהם AL מקבל מספר מתוך טבלה שלה קראתי gOffsetTable ולאחר מכן מבצעים קפיצה אל lblPrintMessageAndHang. אפשר לראות את שלושת הערכים מתוך הטבלה הזו - הערכים הם 0x63, 0x7B, 0x9A.
- בתוך lblPrintMessageAndHang מוסיפים 0x700 אל הערך של AL - ושמים בתוך SI. למשל, עבור האינדקס הראשון בטבלה, SI יקבל 0x763. לאחר מכן יש הדפסת TTY (המשמעות של TTY בהקשר שלנו היא שמבוצעת התקדמות של ה-cursor לאחר כל הדפסה, למשל). בכל שלב מבוצע lodsb (העברת הבית שמוצבע על ידי SI אל AL ואז קידום SI באחד), השוואה לאפס (null terminartor) והדפסה בצבע אפור.
- לאחר שהגענו אל lblHang מבוצע HLT, שאמור למעשה לכבות את ה-CPU עד שתתבצע שוב פסיקה חיצונית. בלי קשר קופצים בלולאה אינסופית, כך שהמחשב "תקוע" (ידוע כ-hang).
- ניתן להמיר ב-IDA את המקומות המוצבעים (0x763, 0x77B, 0x79A למחרוזות ANSI) ולראות מה כתוב בהן. כצפוי, מדובר בהודעות שגיאה כגון Invalid partition table או Missing operating system.

סיכום

- סקרנו את תהליך העלייה (מאוד ב-high level) עד ה-MBR וה-VBR.
- ניתחנו את ה-MBR של Windows 7, ובמיוחד התעמקנו בנושאים הבאים:
  - העתקה עצמית אל 0x600 כדי לפנות מקום ל-VBR.
  - בדיקת תמיכה ב-TPM ועבודה מעטה איתו.
  - הדלקת ה-A20 gate והמשמעות של ההדלקה.
  - פרסור ה-partition table.
  - קריאה מהדיסק ב-LBA או CHS.
- אשתדל לספק סקירה דומה על ה-VBR ולאחר מכן על המשך תהליך העלייה.
- הוספתי נספח של הקוד השלם. הייתי מוסיף IDB, אבל גרסאות שונות של IDA לא תומכות בהכרח בכל IDB וגם נחמד שזה יגיע יחד עם המסמך. מדובר בקוד הסופי, כולל ההערות, כמובן.

על המחבר

0x3d5157636b525761, עושה Reversing ופיתוח Low Level למחייתו.

ניתן ליצור איתי קשר ב:

0x3d5157636b525761@gmail.com

נספח א': הקוד המלא כולל הערות

```
seg000:0600 ;
seg000:0600 ;
seg000:0600 ; Input MD5   : A866B1FE21333A151E05EAB96E17C465
seg000:0600 ; Input CRC32 : F51EBC70
seg000:0600
seg000:0600 ; ---------------------------------------------------------------------------
seg000:0600 ; File Name   : mbr.bin
seg000:0600 ; Format      : Binary file
seg000:0600 ; Base Address: 0000h Range: 0000h - 0200h Loaded length: 0200h
seg000:0600
seg000:0600                 .686p
seg000:0600                 .mmx
seg000:0600                 .model flat
seg000:0600
seg000:0600 ; ===========================================================================
seg000:0600
seg000:0600 ; Segment type: Pure code
seg000:0600 seg000          segment byte public 'CODE' use16
seg000:0600                 assume cs:seg000
seg000:0600                 ;org 600h
seg000:0600                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:0600                 xor     ax, ax
seg000:0602 ;
seg000:0602 ; Build stack (SS:SP = 00:7C00)
seg000:0602 ;
seg000:0602                 mov     ss, ax
seg000:0604                 mov     sp, 7C00h
seg000:0607 ;
seg000:0607 ; Nullify ES and DS
seg000:0607 ;
seg000:0607                 mov     es, ax
seg000:0609                 mov     ds, ax
seg000:060B ;
seg000:060B ; Self replicate to 0x600
seg000:060B ;
seg000:060B                 mov     si, 7C00h
seg000:060E                 mov     di, 600h
seg000:0611                 mov     cx, 200h
seg000:0614                 cld
```

```
seg000:0615                 rep movsb
seg000:0617 ;
seg000:0617 ; Long jump to 00:061C, changing CS and IP by RETF
seg000:0617 ;
seg000:0617                 push    ax
seg000:0618                 push    61Ch
seg000:061B                 retf
seg000:061C ; ---------------------------------------------------------------------------
seg000:061C                 sti
seg000:061D ;
seg000:061D ; Make BP point to the first partition entry
seg000:061D ; Partition table (which contains up to 4 entries) is located at offset 01BE from
the MBR
seg000:061D ; Since MBR was loaded to 7C00 and relocated to 0600, the partition table is at
07BE
seg000:061D ; Make CX be the maximum number of partition entries
seg000:061D ;
seg000:061D                 mov     cx, 4
seg000:0620                 mov     bp, 7BEh
seg000:0623 ;
seg000:0623 ; Check out partitions
seg000:0623 ; First instruction is CMP because we would like to check the MSB as well as the
zero flag
seg000:0623 ;
seg000:0623
seg000:0623 lblNextPartitionEntry:                  ; CODE XREF: seg000:0630j
seg000:0623                 cmp     byte ptr [bp+0], 0
seg000:0627                 jl      short lblFoundBootablePartition
seg000:0629 ;
seg000:0629 ; If the MSB is zero but the status byte isn't zero, the entry is corrupted
seg000:0629 ;
seg000:0629                 jnz     lblInvalidPartitionTable
seg000:062D ;
seg000:062D ; Move to the next entry (each entry is 0x10 bytes)
seg000:062D ;
seg000:062D                 add     bp, 10h
seg000:0630                 loop    lblNextPartitionEntry
seg000:0632 ;
seg000:0632 ; In this point, we've exhausted the partition table and didn't find a bootable
partition
seg000:0632 ; Reboots the machine using INT18
seg000:0632 ;
seg000:0632                 int     18h             ; TRANSFER TO ROM BASIC
seg000:0632                                         ; causes transfer to ROM-based BASIC
(IBM-PC)
seg000:0632                                         ; often reboots a compatible; often has
no effect at all
seg000:0634 ;
seg000:0634 ; BP (which points to the bootable partition entry) is used to save various data:
seg000:0634 ;     Offset=0x00    The drive number (DL)
seg000:0634 ;     Offset=0x10    Whether READ extensions are supported or not
seg000:0634 ;     Offset=0x11    The number of read tries
seg000:0634 ; Backup BP on the stack since we'll use an interrupt, which doesn't save BP
seg000:0634 ;
seg000:0634
seg000:0634 lblFoundBootablePartition:              ; CODE XREF: seg000:0627j
seg000:0634                                         ; seg000:06AEj
seg000:0634                 mov     [bp+0], dl
seg000:0637                 push    bp
seg000:0638                 mov     byte ptr [bp+11h], 5
seg000:063C                 mov     byte ptr [bp+10h], 0
seg000:0640 ;
seg000:0640 ; Check for READ extension availability for the drive number in DL
```

```
seg000:0640 ;
seg000:0640                 mov     ah, 41h ; 'A'
seg000:0642                 mov     bx, 55AAh
seg000:0645                 int     13h             ; DISK - Check for INT 13h Extensions
seg000:0645                                         ; BX = 55AAh, DL = drive number
seg000:0645                                         ; Return: CF set if not supported
seg000:0645                                         ; AH = extensions version
seg000:0645                                         ; BX = AA55h
seg000:0645                                         ; CX = Interface support bit map
seg000:0647 ;
seg000:0647 ; Restore BP
seg000:0647 ;
seg000:0647                 pop     bp
seg000:0648 ;
seg000:0648 ; Carry flag is set if READ extensions are not available
seg000:0648 ; BX should be 0xAA55 on a successful call
seg000:0648 ; LSB of CX should indicate that the support bitmap is suitable
seg000:0648 ; If all of these apply - mark that READ extensions are available in the
(BP+0x10) byte
seg000:0648 ;
seg000:0648                 jb      short lblPerformReading
seg000:064A                 cmp     bx, 0AA55h
seg000:064E                 jnz     short lblPerformReading
seg000:0650                 test    cx, 1
seg000:0654                 jz      short lblPerformReading
seg000:0656                 inc     byte ptr [bp+10h]
seg000:0659 ;
seg000:0659 ; Backup all general-purpose registers
seg000:0659 ;
seg000:0659
seg000:0659 lblPerformReading:                      ; CODE XREF: seg000:0648j
seg000:0659                                         ; seg000:064Ej ...
seg000:0659                 pushad
seg000:065B ;
seg000:065B ; Check if READ extensions are available
seg000:065B ; If not - we read using CHS
seg000:065B ; If we can use READ extensions - we use LBA
seg000:065B ;
seg000:065B                 cmp     byte ptr [bp+10h], 0
seg000:065F                 jz      short lblReadCHS
seg000:0661 ;
seg000:0661 ; Build a disk address packet on the stack
seg000:0661 ;     00    BYTE     Size of packet (0x10 or 0x18)
seg000:0661 ;     01    BYTE     Reserved (0)
seg000:0661 ;     02    WORD     Number of blocks to transfer (1 block)
seg000:0661 ;     04    DWORD    Transfer buffer (00:7C00)
seg000:0661 ;     08    QWORD    Starting absolute block number (bp+8 is the LBA of first
absolute sector)
seg000:0661 ;
seg000:0661                 push    large 0
seg000:0667                 push    large dword ptr [bp+8]
seg000:066B                 push    0
seg000:066E                 push    7C00h
seg000:0671                 push    1
seg000:0674                 push    10h
seg000:0677 ;
seg000:0677 ; Perform the extended READ in LBA form
seg000:0677 ; DL is the drive number
seg000:0677 ;
seg000:0677                 mov     ah, 42h ; 'B'
seg000:0679                 mov     dl, [bp+0]
seg000:067C                 mov     si, sp
seg000:067E                 int     13h             ; DISK - IBM/MS Extension - EXTENDED READ
```

```
(DL - drive, DS:SI - disk address packet)
seg000:0680 ;
seg000:0680 ; Backup the flags on AL
seg000:0680 ; Dispose of the disk address packet from the stack
seg000:0680 ; Restore flags from AL
seg000:0680 ;
seg000:0680                 lahf
seg000:0681                 add     sp, 10h
seg000:0684                 sahf
seg000:0685                 jmp     short lblPerformPostReadValidations
seg000:0687 ; ---------------------------------------------------------------------------
seg000:0687 ;
seg000:0687 ; Read CHS
seg000:0687 ; AL = 0x01 (the number of sectors to read)
seg000:0687 ; AH = 0x02 (read sectors)
seg000:0687 ; ES:BX = 00:7C00 (the buffer to fill)
seg000:0687 ; DL = drive number, the disk number previously saved in the (BP+0) byte
seg000:0687 ; DH = head, from the partition entry
seg000:0687 ; CL = sector, from the partition entry
seg000:0687 ; CH = cylinder, from the partition entry
seg000:0687 ;
seg000:0687
seg000:0687
seg000:0687 lblReadCHS:                             ; CODE XREF: seg000:065Fj
seg000:0687                 mov     ax, 201h
seg000:068A                 mov     bx, 7C00h
seg000:068D                 mov     dl, [bp+0]
seg000:0690                 mov     dh, [bp+1]
seg000:0693                 mov     cl, [bp+2]
seg000:0696                 mov     ch, [bp+3]
seg000:0699                 int     13h             ; DISK - READ SECTORS INTO MEMORY
seg000:0699                                         ; AL = number of sectors to read, CH =
track, CL = sector
seg000:0699                                         ; DH = head, DL = drive, ES:BX -> buffer
to fill
seg000:0699                                         ; Return: CF set on error, AH = status,
AL = number of sectors read
seg000:069B ;
seg000:069B ; Restore general purpose registers
seg000:069B ;
seg000:069B
seg000:069B lblPerformPostReadValidations:          ; CODE XREF: seg000:0685j
seg000:069B                 popad
seg000:069D ;
seg000:069D ; Carry flag is set on error
seg000:069D ; If an error occured, decrease the number of tries in the (BP+0x11) byte
seg000:069D ;
seg000:069D                 jnb     short lblVbrLoadedSuccessfully
seg000:069F                 dec     byte ptr [bp+11h]
seg000:06A2                 jnz     short lblResetDiskAndRetryLoadingVBR
seg000:06A4 ;
seg000:06A4 ; We've exhaused the number of READ attempts
seg000:06A4 ; If DL was 0x80, then we present "error loading operating system" and quit
seg000:06A4 ; Otherwise, we try again one more time with DL=0x80
seg000:06A4 ;
seg000:06A4                 cmp     byte ptr [bp+0], 80h ; 'à'
seg000:06A8                 jz      lblErrorLoadingOperatingSystem
seg000:06AC                 mov     dl, 80h ; 'à'
seg000:06AE                 jmp     short lblFoundBootablePartition
seg000:06B0 ; ---------------------------------------------------------------------------
seg000:06B0 ;
seg000:06B0 ; Backup BP
seg000:06B0 ; Reset the disk number and restore BP
```

```
seg000:06B0 ; This is done because interrupts don't maintain BP
seg000:06B0 ; Then, we perform reading again
seg000:06B0 ;
seg000:06B0
seg000:06B0 lblResetDiskAndRetryLoadingVBR:         ; CODE XREF: seg000:06A2j
seg000:06B0                 push    bp
seg000:06B1                 xor     ah, ah
seg000:06B3                 mov     dl, [bp+0]
seg000:06B6                 int     13h             ; DISK - RESET DISK SYSTEM
seg000:06B6                                         ; DL = drive (if bit 7 is set both hard
disks and floppy disks reset)
seg000:06B8                 pop     bp
seg000:06B9                 jmp     short lblPerformReading
seg000:06BB ; ---------------------------------------------------------------------------
seg000:06BB ;
seg000:06BB ; The VBR is supposed to end with 0xAA55
seg000:06BB ; Since it's loaded to 00:7C00, the last word is at 00:7DFE
seg000:06BB ;
seg000:06BB
seg000:06BB lblVbrLoadedSuccessfully:               ; CODE XREF: seg000:069Dj
seg000:06BB                 cmp     word ptr ds:7DFEh, 0AA55h
seg000:06C1                 jnz     short lblMissingOperatingSystem
seg000:06C3 ;
seg000:06C3 ; Save [BP+0]
seg000:06C3 ;
seg000:06C3                 push    word ptr [bp+0]
seg000:06C6 ;
seg000:06C6 ; Wait for keyboard availability
seg000:06C6 ;
seg000:06C6                 call    WaitForKeyboardInput
seg000:06C9                 jnz     short lblLogTPM
seg000:06CB ;
seg000:06CB ; Keyboard is ready
seg000:06CB ; Write a data byte to the keyboard in order to enable A20 gate.
seg000:06CB ;
seg000:06CB                 cli
seg000:06CC                 mov     al, 0D1h ; '_'
seg000:06CE                 out     64h, al         ; 8042 keyboard controller command
register.
seg000:06CE                                         ; Write output port (next byte to port
60h):
seg000:06CE                                         ; 7:  1=keyboard data line pulled low
(inhibited)
seg000:06CE                                         ; 6:  1=keyboard clock line pulled low
(inhibited)
seg000:06CE                                         ; 5:  enables IRQ 12 interrupt on mouse
IBF
seg000:06CE                                         ; 4:  enables IRQ 1 interrupt on keyboard
IBF
seg000:06CE                                         ; 3:  1=mouse clock line pulled low
(inhibited)
seg000:06CE                                         ; 2:  1=mouse data line pulled low
(inhibited)
seg000:06CE                                         ; 1:  A20 gate on/off
seg000:06CE                                         ; 0:  reset the PC (THIS BIT SHOULD
ALWAYS BE SET TO 1)
seg000:06D0                 call    WaitForKeyboardInput
seg000:06D3                 mov     al, 0DFh ; '_'
seg000:06D5                 out     60h, al         ; 8042 keyboard controller data register.
seg000:06D7                 call    WaitForKeyboardInput
seg000:06DA                 mov     al, 0FFh
seg000:06DC                 out     64h, al         ; 8042 keyboard controller command
register.
```

```
seg000:06DC                                         ; Pulse output port.
seg000:06DC                                         ; Bits 0-3 indicate ports to pulse.
seg000:06DE                 call    WaitForKeyboardInput
seg000:06E1                 sti
seg000:06E2
seg000:06E2 lblLogTPM:                              ; CODE XREF: seg000:06C9j
seg000:06E2                 mov     ax, 0BB00h
seg000:06E5                 int     1Ah             ; Trusted Computing Group call -
TCG_StatusCheck
seg000:06E5                                         ; Return: EAX = 0 if supported
seg000:06E5                                         ; EBX = 41504354h ('TCPA')
seg000:06E5                                         ; CH:CL = TCG BIOS Version
seg000:06E5                                         ; EDX = BIOS TCG Feature Flags
seg000:06E5                                         ; ESI = Pointer to Event Log
seg000:06E5                                         ;
seg000:06E7                 and     eax, eax
seg000:06EA                 jnz     short lblMoveControlToVBR
seg000:06EC                 cmp     ebx, 'APCT'
seg000:06F3                 jnz     short lblMoveControlToVBR
seg000:06F5                 cmp     cx, 102h
seg000:06F9                 jb      short lblMoveControlToVBR
seg000:06FB ;
seg000:06FB ; Change registers by pushing and then performing POPAD
seg000:06FB ; Order: EDI, ESI, EBP, <ignored>, EBX, EDX, ECX, EAX
seg000:06FB ;
seg000:06FB                 push    large 0BB07h    ; (EAX)
seg000:0701                 push    large 200h      ; (ECX)
seg000:0707                 push    large 8         ; (EDX)
seg000:070D                 push    ebx             ; (EBX)
seg000:070F                 push    ebx             ; (ignored)
seg000:0711                 push    ebp             ; (EBP)
seg000:0713                 push    large 0         ; (ESI)
seg000:0719                 push    large 7C00h     ; (EDI)
seg000:071F                 popad
seg000:0721 ;
seg000:0721 ; Change ES to 0
seg000:0721 ;
seg000:0721                 push    0
seg000:0724                 pop     es
seg000:0725 ;
seg000:0725 ; 0xBB07 = TCG_CompactHashLogExtendEvent
seg000:0725 ; ES:DI = Data buffer to be hashed
seg000:0725 ; ECX = Data buffer length
seg000:0725 ; EDX = Event number (PCR number)
seg000:0725 ;
seg000:0725                 int     1Ah             ; Trusted Computing Group call -
TCG_StatusCheck
seg000:0725                                         ; Return: EAX = 0 if supported
seg000:0725                                         ; EBX = 41504354h ('TCPA')
seg000:0725                                         ; CH:CL = TCG BIOS Version
seg000:0725                                         ; EDX = BIOS TCG Feature Flags
seg000:0725                                         ; ESI = Pointer to Event Log
seg000:0725                                         ;
seg000:0727
seg000:0727 lblMoveControlToVBR:                    ; CODE XREF: seg000:06EAj
seg000:0727                                         ; seg000:06F3j ...
seg000:0727                 pop     dx
seg000:0728                 xor     dh, dh
seg000:072A                 jmp     far ptr 0:7C00h
seg000:072F ; ---------------------------------------------------------------------------
seg000:072F ;
seg000:072F ; Restart the machine
seg000:072F ;
```

```
seg000:072F                 int     18h             ; TRANSFER TO ROM BASIC
seg000:072F                                         ; causes transfer to ROM-based BASIC
(IBM-PC)
seg000:072F                                         ; often reboots a compatible; often has
no effect at all
seg000:0731
seg000:0731 lblMissingOperatingSystem:              ; CODE XREF: seg000:06C1j
seg000:0731                 mov     al, ds:gOffsetTable+2
seg000:0734                 jmp     short lblPrintMessageAndHang
seg000:0736 ; ---------------------------------------------------------------------------
seg000:0736
seg000:0736 lblErrorLoadingOperatingSystem:         ; CODE XREF: seg000:06A8j
seg000:0736                 mov     al, ds:gOffsetTable+1
seg000:0739                 jmp     short lblPrintMessageAndHang
seg000:073B ; ---------------------------------------------------------------------------
seg000:073B
seg000:073B lblInvalidPartitionTable:               ; CODE XREF: seg000:0629j
seg000:073B                 mov     al, ds:gOffsetTable
seg000:073E ;
seg000:073E ; AL is the offset from 0x0700
seg000:073E ; Make SI = 0x0700+AL
seg000:073E ;
seg000:073E
seg000:073E lblPrintMessageAndHang:                 ; CODE XREF: seg000:0734j
seg000:073E                                         ; seg000:0739j
seg000:073E                 xor     ah, ah
seg000:0740                 add     ax, 700h
seg000:0743                 mov     si, ax
seg000:0745 ;
seg000:0745 ; Load the next character to AL and increase SI
seg000:0745 ;
seg000:0745
seg000:0745 lblPrintNextChar:                       ; CODE XREF: seg000:0751j
seg000:0745                 lodsb
seg000:0746 ;
seg000:0746 ; If a NUL character is found - finish
seg000:0746 ;
seg000:0746                 cmp     al, 0
seg000:0748                 jz      short lblHang
seg000:074A ;
seg000:074A ; Perform TTY write
seg000:074A ; BX = 7 (gray color)
seg000:074A ;
seg000:074A                 mov     bx, 7
seg000:074D                 mov     ah, 0Eh
seg000:074F                 int     10h             ; - VIDEO - WRITE CHARACTER AND ADVANCE
CURSOR (TTY WRITE)
seg000:074F                                         ; AL = character, BH = display page
(alpha modes)
seg000:074F                                         ; BL = foreground color (graphics modes)
seg000:0751                 jmp     short lblPrintNextChar
seg000:0753 ; ---------------------------------------------------------------------------
seg000:0753 ;
seg000:0753 ; Hang machine (HLT + self JMP)
seg000:0753 ;
seg000:0753
seg000:0753 lblHang:                                ; CODE XREF: seg000:0748j
seg000:0753                                         ; seg000:0754j
seg000:0753                 hlt
seg000:0754 ; ---------------------------------------------------------------------------
seg000:0754                 jmp     short lblHang
seg000:0756
seg000:0756 ; =============== S U B R O U T I N E =======================================
```

```
seg000:0756
seg000:0756
seg000:0756 WaitForKeyboardInput proc near          ; CODE XREF: seg000:06C6p
seg000:0756                                         ; seg000:06D0p ...
seg000:0756                 sub     cx, cx
seg000:0758
seg000:0758 lblKeyboardPoll:                        ; CODE XREF: WaitForKeyboardInput+8j
seg000:0758                 in      al, 64h         ; 8042 keyboard controller status
register
seg000:0758                                         ; 7:  PERR    1=parity error in data
received from keyboard
seg000:0758                                         ;    +----------- AT Mode ----------+----
-------- PS/2 Mode ------------+
seg000:0758                                         ; 6: |RxTO    receive (Rx) timeout  | TO
general timeout (Rx or Tx)|
seg000:0758                                         ; 5: |TxTO    transmit (Tx) timeout |
MOBF    mouse output buffer full  |
seg000:0758                                         ;    +------------------------------+----
------------------------------+
seg000:0758                                         ; 4:  INH     0=keyboard communications
inhibited
seg000:0758                                         ; 3:  A2      0=60h was the port last
written to, 1=64h was last
seg000:0758                                         ; 2:  SYS     distinguishes reset types:
0=cold reboot, 1=warm reboot
seg000:0758                                         ; 1:  IBF     1=input buffer full
(keyboard can't accept data)
seg000:0758                                         ; 0:  OBF     1=output buffer full (data
from keyboard is available)
seg000:075A                 jmp     short $+2
seg000:075C                 and     al, 2
seg000:075E                 loopne  lblKeyboardPoll
seg000:0760                 and     al, 2
seg000:0762                 retn
seg000:0762 WaitForKeyboardInput endp
seg000:0762
seg000:0762 ; ---------------------------------------------------------------------------
seg000:0763 aInvalidPartitionTable db 'Invalid partition table',0
seg000:077B aErrorLoadingOperatingSystem db 'Error loading operating system',0
seg000:079A aMissingOperatingSystem db 'Missing operating system',0
seg000:07B3                 db    0
seg000:07B4                 db    0
seg000:07B5 gOffsetTable    db 63h, 7Bh, 9Ah        ; 0
seg000:07B5                                         ; DATA XREF:
seg000:lblInvalidPartitionTabler
seg000:07B5                                         ; seg000:lblErrorLoadingOperatingSystemr
...
seg000:07B8                 db  12h
seg000:07B9                 db 0D3h ; _
seg000:07BA                 db 0C3h ; _
seg000:07BB                 db 0A0h ; _
seg000:07BC                 db    0
seg000:07BD                 db    0
seg000:07BE                 db  80h ; à
seg000:07BF                 db  20h
seg000:07C0                 db  21h ; !
seg000:07C1                 db    0
seg000:07C2                 db    7
seg000:07C3                 db 0DFh ; _
seg000:07C4                 db  13h
seg000:07C5                 db  0Ch
seg000:07C6                 db    0
seg000:07C7                 db    8
```

```
seg000:07C8                 db    0
seg000:07C9                 db    0
seg000:07CA                 db    0
seg000:07CB                 db  20h
seg000:07CC                 db    3
seg000:07CD                 db    0
seg000:07CE                 db    0
seg000:07CF                 db 0DFh ; _
seg000:07D0                 db  14h
seg000:07D1                 db  0Ch
seg000:07D2                 db    7
seg000:07D3                 db 0FEh ; _
seg000:07D4                 db 0FFh
seg000:07D5                 db 0FFh
seg000:07D6                 db    0
seg000:07D7                 db  28h ; (
seg000:07D8                 db    3
seg000:07D9                 db    0
seg000:07DA                 db    0
seg000:07DB                 db 0D8h ; _
seg000:07DC                 db  66h ; f
seg000:07DD                 db  18h
seg000:07DE                 db    0
seg000:07DF                 db 0FEh ; _
seg000:07E0                 db 0FFh
seg000:07E1                 db 0FFh
seg000:07E2                 db    7
seg000:07E3                 db 0FEh ; _
seg000:07E4                 db 0FFh
seg000:07E5                 db 0FFh
seg000:07E6                 db    0
seg000:07E7                 db    0
seg000:07E8                 db  6Ah ; j
seg000:07E9                 db  18h
seg000:07EA                 db    0
seg000:07EB                 db  58h ; X
seg000:07EC                 db 0CEh ; _
seg000:07ED                 db  21h ; !
seg000:07EE                 db    0
seg000:07EF                 db    0
seg000:07F0                 db    0
seg000:07F1                 db    0
seg000:07F2                 db    0
seg000:07F3                 db    0
seg000:07F4                 db    0
seg000:07F5                 db    0
seg000:07F6                 db    0
seg000:07F7                 db    0
seg000:07F8                 db    0
seg000:07F9                 db    0
seg000:07FA                 db    0
seg000:07FB                 db    0
seg000:07FC                 db    0
seg000:07FD                 db    0
seg000:07FE                 dw 0AA55h
seg000:07FE seg000          ends
seg000:07FE
seg000:07FE
seg000:07FE                 end
```