Technologic Papers

פיתוח מערכות הפעלה – חלק א'

נכתב ע"י עידן פסט

הקדמה

מהי מערכת הפעלה?

"מערכת הפעלה היא תוכנה המגשרת בין המשתמש, החומרה ויישומי התוכנה. זו התוכנה הראשונה שעולה עם הדלקת המחשב והיא זו המאפשרת לו לפעול. מערכת ההפעלה מספקת שלושה ממשקים: ממשק משתמש (User Interface), ממשק עבור החומרה על ידי מנהלי התקנים וממשק תכנות היישומים (API). מערכת ההפעלה היא רכיב חיוני בכל מחשב. תהליך טעינתה של מערכת ההפעלה, המתבצע עם הדלקת המחשב, קרוי אתחול." - ויקיפדיה

בשביל מה צריך את מערכת ההפעלה?

מערכת ההפעלה מהווה במה משותפת שעליה יכולות לרוץ תוכנות שישרתו את משתמש הקצה. לדוגמא, כשאני כותב את המאמר הזה, המפתח של Office לא היה צריך לכתוב רכיב שמתממשק עם הלוח אם כדי לקבל את הקשות המקלדת שלי, ולא היה צריך לדבר ישירות עם כרטיס המסך כדי שהאותיות יופיעו לי עליו. במקום זאת, המפתח דאג להנחות את המחשב: "תגיב בצורה א' למקש כזה ובצורה ב' למקש כזה" ו"תצייר לי את האות ג'". מערכת ההפעלה עשתה בשביל המפתח את העבודה והיוותה את הגשר בין החומרה לבין התוכנה.

מעבר להתממשקות עם החומרה, מערכת ההפעלה גם אחראית לנהל את זיכרון המחשב, זמן המעבד, חלוקת משאבים ועוד אלפי רכיבים אחרים. ככל שעובר הזמן מערכות ההפעלה הנפוצות לוקחות על עצמן יותר ויותר אחריות, כאשר פונקציונאליות שהייתה פעם ממומשת על ידי אפליקציות רגילות, כיום ממומשות על ידי מערכת ההפעלה.

ככל שעובר הזמן, התלות שלנו במערכת ההפעלה גדלה, ואיתה הצורך להבין לעומק את מהות מערכת ההפעלה ודרך פעולתה. מעבר לזה, תכנות מערכות הפעלה זה תחביב שיוכיח לכל החברים שלכם שאתם Hardcore כמו שאתם טוענים שאתם (למרות שאם באמת הייתם Hardcore לא היה לכם חברים, אז על מי אתם עובדים בדיוק?)

מערכות ההפעלה הנפוצות כיום

כיום בשוק קיימות מספר מערכות הפעלה נפוצות, שנבדלות ב"התמחות" שלהן:

1. **Windows** - פותחה על ידי Microsoft, ובעלת נתח השוק הכי גדול. רוב המשתמשים במחשב האישי בוחרים במערכת ההפעלה הזו, בעיקר בגלל תאימותה עם הרוב המוחלט של האפליקציות כיום, תמיכה טכנית נפוצה ושימוש בכוח המונופולי של Microsoft כדי למנוע משחקנים נוספים להיכנס לשוק.

2. **Linux** - כיום Linux מהווה לרוב את הבסיס למספר הפצות של לינוקס כדוגמת Ubuntu, שנפוצה בקרב מחשבים אישיים ו-CentOS, שנפוצה בשרתים. Linux פותחה במקור על ידי לינוס טורבאלדס, אבל היום מאורגנת סביבה קהילה פעילה של מפתחים. Linux היא מערכת הפעלה חינמית, חופשית ובקוד פתוח ומהווה את הדוגמא הגדולה ביותר לפרויקטים בקוד פתוח. מיועדת בעיקר לפריקים של קוד פתוח, מחשבי על ושרתים.

3. **Macintosh** - פותחה על ידי Apple ומותקנת ביחד עם מחשבי ה-Mac של Apple. היום היא עונה לשם OS X, ומהווה את המתחרה השנייה בגודלה בשוק המחשבים האישיים. מבוססת Unix (בדומה ל-Linux), ונפוצה אצל מעצבים גרפיים.

4. **Android**- מערכת ההפעלה הכי נפוצה לסמארטפונים. מבוססת על Linux ופותחה על ידי קבוצה של כמה חברות, כשהמובילה בהן היא Google. המערכת מפותחת בקוד פתוח ועליה רצים בד"כ תוכנות צד שלישי המפותחות ב-Java.

5. **iOS** - מערכת ההפעלה למכשירים הניידים של Apple. המערכת היא התאמה של OS X למכשירים ניידים, הכוללים את ה-iPad, ה-iPhone ועוד. המערכת מריצה תוכנות צד שלישי שמפותחות ב-Objective C.

אלו רק כמה ממערכות ההפעלה המרכזיות שיוצא למשתמש ממוצע להתקל בהן במודע. האדם הממוצע משתמש במערכות הפעלה נוספות כדוגמת מערכת הפעלה בכספומט או במכשיר מיקרו, אך השימוש נעשה בדרך כלל לא במודע.

מה אנחנו נעשה?

החלק הראשון של המאמר יתמקד בתיאוריה שמאחורי תכנות מערכות הפעלה, ונתחיל לבנות מערכת הפעלה **בסיסית** (ברמת מסך שחור עם כמה פקודות בסיסיות, שעליו תוכלו להמשיך ולבנות מה שרק תחלמו עליו). מערכת ההפעלה שנבנה בפרק זה תהיה מאוד מינימאלית, ותרוץ כמערכת הפעלה Live - משמע, שום דבר לא נשמר באופן קבוע לדיסק, והמערכת תתחיל מחדש בכל הפעלה של המחשב.

מערכת ההפעלה, סימן 1

מה נצטרך?

ידע קודם ויכולות:

- שליטה סבירה ב-Assembly x86 וב-C.
- יכולת חיפוש בגוגל - יש המון בעיות שיכולות לצוץ, והרבה מאוד אנשים בעולם נתקלו באותה הבעיה ובכך ניתן למצוא פתרונות לבעיות שצצות.
- סבלנות. והרבה ממנה.

כלים:

- VMWare, QEMU או כל פיתרון וירטואליזציה אחר.
- מומלץ בחום: מחשב שמריץ Linux. בדוגמאות שאציג אני אשתמש בכלים נפוצים הקיימים ב-Linux (בעיקר GCC). אם אתם בוחרים לנסות לפתח על Windows, אני מאחל לכם הרבה הצלחה.
- Cross-Compiler. זהו מהדר שיכול לקמפל קבצים לפורמטים וארכיטקטורות ששונות מהמחשב שעליו הוא רץ. גם אם אתם הולכים לתכנת לאותה סביבה, שימוש ב-Cross-Compiler ימנע בעיות של Dependency בספריות או חוסר תאימות בין קבצי PE ל-ELF.
- Assembler, ומומלץ NASM - מספר מאקרוים הם בעלי סינטקס ייחודי ל-NASM. כמובן שניתן למצוא להם תחליף ב-Assemblers אחרים, אך הדוגמאות שיובאו במדריך נבדקו ב-NASM בלבד.

הקוד

נכתוב את מערכת ההפעלה הראשונה שלנו, שתעשה... כלום:

```
[BITS 16]
[ORG 0x7c00]

jmp $

TIMES 510-($-$$) db 0
dw 0xaa55
```

בוא נעבור על כל שורה:

```
[BITS 16]
```

כשאנו מתחילים את הקוד בהוראה [BITS 16] אנו מנחים את המרכיב לתרגם את הקוד לארכיטקטורת 16 ביט ב-Real Mode.

CPU

רכיב ה-CPU (Central Processing Unit, מעבד) הוא המוח של המחשב - הוא האחראי העיקרי לביצוע של הוראות. מעבדים יודעים לדבר בשפה מאוד מוגדרת, ואוסף כל ה"מילים" שהם יודעים נקרא Instruction set (סט פקודות). את המילים המעבד מקבל בצורת קוד מכונה - אותות חשמליים שמייצגים את סט הפקודות. אנו יכולים לתקשר עם המעבד דרך שפה מתווכת בשם Assembly (שפת סף). שפה זו היא בעלת התאמה חד-חד ערכית לקוד המכונה, ורכיב בשם Assembler (מרכיב) מתרגם את שפת הסף לקוד המכונה. כפי שציינתי, המרכיב שאנחנו נשתמש בו הוא NASM (Netwide Assembler), והוא מסוגל לייצר מגוון רחב של סוגי קבצים.

RAM

RAM (Random Access Memory) הוא סוג זיכרון שנגיש באופן ישיר למעבד. זיכרונות מתאפיינים בכמה תכונות: ראשית, הם נדיפים, משמע הזיכרון תלוי באספקת החשמל אליהם, וכאשר המחשב כבה הזיכרון הנוכחי נמחק איתו. שנית, הם מתאפיינים במהירות גבוהות בכמה סדרי גודל מזיכרונות לא נדיפים. אוסף כל כתובות הזיכרון שהמעבד יכול לגשת אליהן נקרא Address Space. ה-Address Space של מחשב שרץ עם 16 ביט הוא 2^{16} בתים (או 64 KB). ה-Address Space של מחשב שרץ עם 32 ביט הוא 2^{32} בתים (או 4GB). היום קיים גם מצב נוסף בעל 64 ביט, שמאפשר להרחיב את הAddress Space מ-4 ג'יגהבייט ל16- אקסהבייט, שהם, לחלופין, 16,000,000,000 GB, (שידוע בכינוי "פ**ינג המוווווווון".) אבל, בדיוק כמו שפעם 4GB נחשב כהרבה, זה רק עניין של זמן עד שגודלי זיכרון כאלה יהיו קיימים.

אופני פעולה של המעבד

למעבדים מודרניים (מאז שחרור סט הפקודות הידוע בשם 80286 ב-1982) קיימים מספר אופני פעולה שונים. הראשון נקרא Real Mode - מצב זה מקביל לאופן הפעולה של כל מעבד שנוצר לפני סדרת ה-80286. במצב זה המעבד עובד ללא שום מנגנוני הגנה - הגישה לחומרה נעשית באופן ישיר וללא שום בקרה. מנגד, Protected Mode מאפשר להשתמש בתכונות כמו זיכרון וירטואלי, ריבוי משימות והגדרת רמות שונות של הרשאות בעת הרצת קוד.

מערכות הפעלה מודרניות משתמשות ב-Protected Mode, אך בכדי לאפשר תאימות לתוכניות שנכתבו ל-Real Mode קיים אופן פעולה נוסף בשם Virtual Mode, המאפשר שימוש בתכונות שקיימות ב-Real Mode מתוך Protected Mode, וזאת משום שלא ניתן לעבור מ-Protected Mode ל-Real Mode ללא ביצוע אתחול מחדש למעבד (דבר שמן הסתם ממש לא רצוי בעת שימוש רגיל במחשב).

הבדל עיקרי נוסף הוא ש-Protected Mode מאפשר לעבור משימוש ב-16 ביט לשימוש ב-32 ביט.
ההבדל בין 16 ביט ל-32 ביט נעוץ בשני שינויים עיקריים. ראשית, ה-Registers (אוגרים) במעבד גדלים ל-32 ביט, אך ההבדל השני, והעיקרי יותר, הוא שגודל הזיכרון שהמעבד יכול לגשת אליו גדל. מאז הגדילה בנפחי הזיכרון בשוק קיים אופן פעולה נוסף בשם Long Mode שמאפשר להשתמש ב-64 ביט. כיום, כדי לאפשר תאימות עם מערכות הפעלה ישנות, מחשבים נדלקים כברירת מחדל למצב Real Mode, ולכן אנו חייבים לכתוב את הקוד הראשוני במצב זה.

אופן	Real Mode	Protected Mode	Virtual Mode	Long Mode
Address Space	20 ביט	32 ביט	20 ביט	64 ביט
גודל אוגרים	16 ביט	32 ביט	16 ביט	64 ביט
גישה לחומרה	ישירה	מוגנת	מוגנת	מוגנת
קיים מאז	תמיד	80286	80386	Opteron
גישה לזיכרון	סגמנטים	דפדוף	סגמנטים	דפדוף

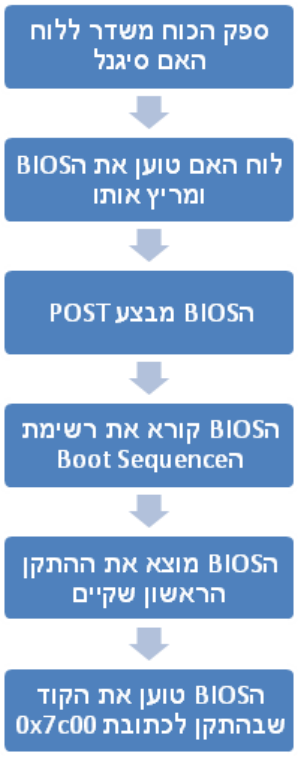
השורה הבאה בקוד, היא:

```
[ORG 0x7c00]
```

גם פקודה זו מנחה את המרכיב, ולא את המעבד. הפקודה אומרת ל-NASM שהקוד שלנו נטען מהכתובת 0x7c00, בניגוד ל-0x0000. סיבה זו נעוצה בכך שה-BIOS טוען את התכנית לכתובת זו.

BIOS

ה-BIOS (Basic Input Output System) הוא הרכיב הראשון שמופעל כשאנחנו מדליקים את המחשב. ה-BIOS אחראי על ביצוע בדיקה ראשונית למחשב המכונה POST ("בדיקה עצמית לאחר הפעלה"). בדיקה זו אחראית לוודא שכל רכיבי המחשב קיימים.



כמו כן, ה-BIOS מהווה ממשק אוניברסאלי לרכיבי החומרה, שמאפשר גישה אליה. אנו ננצל יכולת זו בתחילת הדרך של כתיבת מערכת ההפעלה. כאשר אנו מדליקים את המחשב, ספק הכוח מקבל מתח ולאחר שהוא מצליח לייצב אותו הוא מעביר סיגנל ללוח האם שאומר שהכל תקין. לאחר מכן הלוח אם קורא מצ'יפ CMOS שמכיל את קוד ה-BIOS ומריץ אותו. ה-BIOS נטען, ומבצע את ה-POST. אם בדיקת ה-POST לא הצביעה על אף בעיה, ה-BIOS עובר על רשימה בשם Boot Sequence, שמגדירה את הסדר שבו ה-BIOS יחפש התקני אחסון בכדי לבצע מהם את האתחול. ה-BIOS עובר מתחילת הרשימה, ובודק אם ההתקנים שכתובים לו אכן מחוברים למחשב ומכילים מידע שאפשר לקרוא אותו. כשה-BIOS מגלה את ההתקן הראשון שאכן קיים, הוא קורא אותו, וטוען את הקוד אל הכתובת 0x7c00. לאחר מכן ה-BIOS מעביר את השליטה אל הקוד.

והשורה הבאה שלנו:

```
jmp $
```

הסימן $ הוא סימן שאינו חלק מההגדרה הכללית של שפת הסף, אבל הוא תוסף נחמד שהוסיפו למרכיב שאנחנו נשתמש בו - NASM. הסימן בעצם אומר "פה" - המרכיב מחליף את הסימן בכתובת של תחילת השורה שבה הסימן כתוב. קפיצה לכתובת של אותו שורה בעצם תפעיל את הפקודה הזאת שוב ושוב, והתוכנה בעצם לא תעשה כלום. דבר זה שקול לכתיבת הקוד הבא:

```
loop:
      jmp loop
```

השורה הבאה:

```
TIMES 510-($-$$) db 0
```

בשורה זו לא קיימות שום פקודות ששייכות לשפת הסף הכללית, והיא מכילה רק סימונים ופקודות מיוחדים ש-NASM מבין. הראשון זה פקודת ה-TIMES - פקודה זו ("מאקרו") פשוט משכפלת את קוד שפת הסף שנותנים לה מספר פעמים נתון. במקרה זה הפקודה שאנו רוצים לשכפל היא "db 0" ומספר הפעמים הוא ($$-$)-510. הסימן המיוחד השני הוא $$ - בדומה ל-$, שמסמן את הכתובת של תחילת השורה, הסימן $$ מסמן את הכתובת של תחילת ה-section או במילים אחרות הסימן אומר "ההתחלה" (במקרה שלנו, קיים רק section אחד ולכן הסימן מייצג את הכתובת של תחילת הקוד). לכן, הביטוי $$-$ בעצם נותן לנו את האורך של הקוד שכתבנו עד עכשיו.

ה-MBR אמור להיות באורך של 512 בתים, כששני הבתים האחרונים הם חתימה ייחודית של ה-MBR. לכן אנו צריכים למלא את 510 הבתים האחרים במשהו - וכל מה שלא חלק מהקוד שלנו אנחנו צריכים למלא באפסים.

MBR

ה-MBR(Master Boot Record) הוא חלק בהארד דיסק שמשמש שתי מטרות:

1. הוא מכיל רשימה של כל המחיצות בדיסק. מחיצות הן חלוקות לוגיות של הדיסק שמקלות על גישה וחלוקה של הדיסק. מבחינת מערכת ההפעלה, כל מחיצה היא דיסק בפני עצמה.
2. ה-MBR יכול להכיל קוד בסיסי שירוץ כאשר ה-BIOS קורא אותו. אנו ננצל תכונה זו בתחילת כתיבת המערכת.

באופן היסטורי החלוקה לאזורים בהתקן כזה נעשתה בשיטה הקרויה CHS (Cylinder-Head-Sector), שתאמה את החלוקה הפיזית שקיימת בהתקן. כאשר תוכנה רצתה לגשת להארד דיסק היא העבירה אוסף של שלושה מספרים שייצגו את הצילינדר, ראש וסקטור שהמידע יושב עליו. שיטת גישה זו מקבילה לדוגמא לכתובת, בה מסמנים את העיר (יחידת החלוקה הכי גדולה), הרחוב ומספר הבית (יחידת החלוקה הכי קטנה). היום הגישה נעשית בשיטה בשם LBA (Linear Block Addressing) שמייצגת באופן ליניארי כל אזור (שנקרא Block).

ה-MBR יושב על הסקטור הראשון בהארד דיסק (לחלופין, זה גם יכול להיות דיסק און קי או כל התקן אחסון משני אחר). הסקטור הראשון יושב בכתובת CHS 1,0,0, או לפי שיטת ה-LBA - בבלוק הראשון (שימו לב שסקטורים מתחילים במספר 1 ולא ב-0). ה-MBR נכתב בפורמט הבא:

טווח בתים	מידע
1-440	איזור קוד. איזור זה יכול להכיל קוד שירוץ לאחר שה-BIOS קורא את הMBR.
441-444	חתימת הדיסק.
445-446	ממולא ב-NULL.
447-510	טבלת המחיצות. מחולקת ל-4 רשומות של 16 בתים, שמייצגות את מספר המחיצות הפיזיות האפשרויות על הארד דיסק יחיד. היום ניתן גם ליצור מחיצות לוגיות שיכילו מספר תתי מחיצות.
511	0x55 - בית ראשון של חתימת הMBR
512	0xAA - בית שני של חתימת הMBR

היום כל רכיב BIOS מודרני יודע להתמודד עם MBR שלא מכיל את החתימה (0x55AA) הרגילה, וכיום גם אין צורך למלא את הבתים 441 עד 446 במידע לא שימושי וניתן למלא אותם בקוד. רכיבי BIOS רבים גם ירשו לנו לכתוב קוד על אזור טבלת המחיצות, כך שבעצם יש לנו פה 512 בתים חופשיים לכתיבת קוד.

בשביל זה קיימת הפקודה "db" - פקודה זו היא חלק ממשפחת פקודות שמכריזות על אזור בזיכרון לשימוש מיוחד והן כתובות כ- "dX" כאשר X מסמל את גודל אזור הזיכרון. במקרה שלנו, אנו מכריזים על אזור של byte. בד"כ משתמשים בפקודות אלו בכדי להכריז על משתנים, אבל אנחנו ננצל אותה כרגע בכדי למלא את האזור החסר באפסים.

הפקודה הבאה:

```
dw 0xaa55
```

כפי שצוין, פה אנו מכריזים על אזור בגודל של מילה (שני בתים) שיכיל את הערך 55aa, חתימה הייחודית של ה-MBR. אם שמתם לב שהחתימה של ה-MBR היא 55aa אך בקוד שלנו כתוב aa55, זו לא טעות - זה בגלל ה-Endianness (סדר בתים) של המחשב.

Endianness

Endianness (סדר בתים) הוא תכונה של מחשבים שמגדירה את הסדר שבה ערכים בגודל העולה על בית אחד נשמרים בזיכרון. לדוגמא, כאשר מחשב שומר את הערך 0x0A0B, הייצוג בזיכרון יכול להראות כך (כשהתא השמאלי הוא הקודם בזיכרון):

0A	0B

או כך:

0B	0A

לשיטה שבה הבית המשמעותי יותר נמצא קודם בזיכרון קוראים Big-Endian, והיא מתאימה לצורה שבה אנחנו קוראים מספרים - 32 כ"שלושים" (החלק המשמעותי) ו"שתיים". לשיטה השנייה קוראים Little-Endian והיא מתאימה לצורה שבה אנחנו מחברים מספרים - בכדי לבצע 32 ועוד 27 קודם מחברים 7 ל-2 (החלק הפחות משמעותי) ובודקים אם יש העברת תוצאות לספרת העשרות ואם לא אז מחברים 2 עם 3 ומקבלים 59.

לכל אחת מהשיטות יש יתרון משלה, אך הארכיטקטורות הנפוצות היום (x86 ו-x86-64) משתמשות בשיטת Little-Endian בגלל שתי סיבות עיקריות: ראשית, בגלל שבעת חיבור העברת תוצאה עוברת מהחלקים הפחות משמעותיים לחלקים היותר משמעותיים כך שחיבור יכול להתבצע בסדר קריאה רגיל מהזיכרון. שנית, שיטת ה-Little-Endian מאפשרת לקרוא אזור בזיכרון ללא ידע מוקדם על הגודל שלו.

בשביל הידע הכללי, השמות Little-Endian ו-Big-Endianו מגיעים מ"מסעי גוליבר", שבו מתוארת מלחמה בין שתי ממלכות שיצאו למלחמה כי אחת מהן סירבה לשבור את הביצה על צדה הצר, ולכן כונו תושביה Big-Endians. האדם אשר טבע את המונח הזה היה דני כהן מהטכניון. אחח, גאווה ישראלית.

הרצה

עתה, אחרי שהבנו את הקוד, נלמד איך להפוך אותו למערכת הפעלה שניתן להריצה.

סידור תיקיות

על מנת להקל על ההבנה, אני מציע את הסידור הבא לתיקיות שלכם:

```
OS
├──────src
├──────bin
```

בתוך תיקיית src הכניסו את הקוד שלכם, בתיקיית bin שימו את כל הקבצים הבינאריים שלכם. אני יצרתי גם תת-תיקיות בכל אחת מהתיקיות בכדי לסדר את הפרויקט עוד יותר - במקרה שלנו, יצרתי תיקיית boot בכל אחת מן התיקיות. אני מניח שבכל אחת מהתיקיות קיימת תיקיית boot, והקוד שיצרתם נקרא boot.asm.

הרכבה

תחילה, נקרא ל-NASM שיהפוך לנו את קוד שפת הסף לקוד מכונה:

```
nasm -f bin -o bin/boot/boot.bin src/boot/boot.asm
```

פקודה זו אומרת ל-NASM לקחת את הקוד שכתוב בקובץ שקיים ב-src/boot/boot.asm, להפוך אותו לקובץ bin, ולכתוב אותו לקובץ bin/boot/boot.bin. יש לציין שבגלל שהנחנו את NASM להפיק את הקובץ בפורמט bin, שמשומש בעיקר לBootloaders ול-MS-DOS, NASM יודע אוטומטית להפיק את הקובץ ב-16 ביט Real Mode, כך שהיינו יכולים לוותר על השורה הראשונה בקוד.

הפיכה לדיסק

עתה קיבלנו קובץ בינארי וניצור ממנו תמונת דיסק (ISO) שנוכל לבצע ממנה אתחול:

```
mkisofs -R -input-charset utf8 -b boot/boot.bin -no-emul-boot -boot-
load-size 4 -o os.iso bin
```

פקודה זו יוצרת קובץ ISO. קובץ ISO הוא קובץ שמאגד בתוכו מערכת קבצים שלמה מסוג ISO9660, הידועה גם כ-CDFS. מערכת קבצים זו משמשת בעיקר כמערכת הקבצים הנפוצה ב-CD-ROM ו-DVD. אנו מכילים את כל המערכת קבצים הזו בתוך קובץ ISO, כך שנוכל אחרי זה לצרוב אותו לדיסק או להריץ מתוכו את מערכת ההפעלה דרך תוכנת וירטואליזציה שיודעת להתייחס אליו ככונן דיסקים לכל דבר. הפרמטרים שמועברים בפקודה קובעים שהדיסק ייוצר כ-El-Torito במצב No Emulation, וגודל ה- Boot Sector יהיה 4 סקטורים (זאת בגלל ש-ISO9660 הוא בעל סקטורים בגודל של 2048 בתים, שהם פי 4 מהגודל של סקטור רגיל של הארד דיסק).

El-Torito

בכדי להבין כיצד דיסק El-Torito מתפקד יש להבין את הרקע ההיסטורי - כונני פלופי שהכילו סך הכל 1.44MB. באותה תקופה, פלופי היה הדבר היחיד ש-BIOS-ים ידעו לקרוא ממנו את מערכת ההפעלה. כאשר הגיע ה-CD והשתלט על השוק, רצו לאפשר ל-BIOS לקרוא גם את הדיסקים החדשים, אך לא רצו לשנות לחלוטין את הקוד של ה-BIOS כך שיוכל להבין גם את מערכת הקבצים, ולכן נוצר ה-El-Torito. הרעיון - כל דיסק יכיל קובץ בשם boot.catalog שמכיל מצביע לקובץ אחר שממנו ה-BIOS יכול לקרוא את מערכת ההפעלה.

שאריות של תקופות חשוכות יותר נשארו בכפתור השמירה

הקובץ הזה הוא העתק של פלופי, כך שה-BIOS מבחינתו טוען פלופי, ואנחנו יכולים לבצע Boot מה-CD. היום, כשנדיר כבר להשתמש בפלופי, ניתן ליצור דיסק El-Torito שמכיל בתוכו הארד דיסק שלם במקום פלופי. מצב זה נקרא "No Emulation". בגלל שאנו משתמשים ב-No Emulation, אנו צריכים לדאוג שהמידע שאנו כותבים ל-CD נראה כמו הארד דיסק רגיל, ולכן אנו צריכים לכתוב MBR.

הרצה

ועתה נתחיל את QEMU (אם אתם משתמשים ב-VMWare, פשוט תגדירו את הקובץ כדיסק שממנו תעשה התכונה Boot):

```
qemu -cdrom os.iso
```

שימו לב שניתן גם פשוט לצרוב את מערכת ההפעלה שכתבתם על דיסק ולהפעיל אותה על מחשב אמיתי, אך בינתיים חבל לבזבז דיסק על מערכת הפעלה שלא עושה כלום.

אם אתם לא רואים שה-BIOS של המכונה הוירטואלית מדווח לכם על בעיה בהפעלת מערכת ההפעלה מהדיסק, הכול הלך כמו שצריך. אם כן, גוגל הוא חברכם הטוב והמקום למציאת פתרון לבעיה.

מערכת ההפעלה, סימן 2

כעת ניקח את מערכת ההפעלה לשלב הבא: הדפסה של מחרוזות על המסך.

קוד

```
[BITS 16]
[ORG 0x7c00]

mov si,msg
call prints
jmp $

prints:
      mov ah, 0x0e
      printc:
            lodsb
            cmp al,0
            jz return
            int 0x10
            jmp printc
      return:
            ret


msg db "Hello World!",0

TIMES 510-($-$$) db 0
dw 0xaa55
```

בוא נעבור על השורות שנוספו, השורה הראשונה:

```
msg db "Hello World!",0
```

להזכירכם, הפקודה db מכריזה על אזור מיוחד בזיכרון. בשורה זו אנו יוצרים אזור בזיכרון שיכיל את המחרוזת "Hello World!" ונוכל לפנות אליו בשם msg. את המחרוזת אנו נסיים בבית שיכיל את הערך 0. סוג המחרוזות שמסתיימות בבית הזה נקרא Null Terminated Strings. הסיבה לסיום המחרוזות ב-Null Byte (בית המכיל את הערך 0) היא בכדי לסמן את סוף המחרוזת. שימו לב שקיימת אפשרות נוספת לסמן את סוף המחרוזת - לשמור מראש את האורך שלה. לכל שיטה יתרונות וחסרונות משלה, אך מאחר ואנו נשתמש בשלבים מאוחרים יותר ב-C, אשר מסיימת מחרוזות ב-Null Byte, אנו נשתמש בשיטה זו.

השורה הבאה:

```
mov si,msg
```

הפקודה מעבירה את הערך של המשתנה msg (הערך שלו הוא אזור בזיכרון שמכיל את מה שהכנסנו אליו) אל האוגר SI. האוגר SI (Source Index) משמע בעיקר להעתקות של מחרוזות - הוא מסמל את האזור בזיכרון שממנו יועתק הזיכרון.

השורה הבאה:

```
lodsb
cmp al,0
jz return
```

סדרת פקודות אלו אחראית לטעון את התו הבא לתוך אוגר. הפקודה lodsb מקבילה לקוד:

```
mov al,[si]
inc si
```

לאחר ביצוע הפקודה, האוגר al מכיל את הערך של התו הקרוב במחרוזת שהאוגר si מצביע עליה, והערך של האוגר מועלה ב-1 כך שהוא מצביע לתו הבא במחרוזת.

לאחר מכן אנו משווים את al ל-0. אם al אכן 0 הדגל ZF (Zero Flag) יודלק. להזכירכם, al יהיה אפס רק כאשר נגיע לבית האחרון במחרוזת - ה-Null Byte, וכך נדע שאנו הגענו לסופה. לאחר מכן אנו מבצעים קפיצה מותנית - אם דגל ה-ZF דלוק, קפוץ ל-return.

הפקודות הבאות:

```
mov ah, 0x0e
…
int 0x10
```

פקודת int 0x10 שולחת את פסיקה מספר 0x10 ל-BIOS. פסיקה זו אחראית לטיפול בוידיאו. בכדי להשתמש בפסיקה זו יש להעביר לה את מספר התת-פסיקה שאנו רוצים ליצור באוגר ah. הערך 0x0e מגדיר את תת-הפסיקה כהדפסת תו על מסך המחשב. תת פסיקה זו מקבלת את ערך ה-ASCII של התו באוגר al, אותו קבענו בפקודה lodsb.

BIOS Interrupts

Interrupt (פסיקה) היא אות הנשלח לרכיב בכדי לבצע פעולה שעוקפת את שגרת הביצוע הרגילה. בעת קבלת פסיקה, הרכיב מפסיק כל פעולה אחרת בכדי לטפל בפסיקה שהתקבלה. בReal Mode ניתן לבצע פסיקות ישירות לרכיב ה-BIOS ודבר זה ממומש על ידי IVT (Interrupt Vector Table) זוהי טבלה שיושבת ב-1024 הבתים הראשונים בזיכרון, ומורכבת מ-255 ערכים של 4 בתים שמייצגים 255 סוגי פסיקות שניתן ליצור. כל ערך בטבלה מכיל מצביע לאזור קוד שיכול לטפל באותו סוג של פסיקה. קיימים מספר סוגי פסיקות לשימוש על ידי התוכנה (חלק גדול מהפסיקות משומש רק על ידי החומרה), ובטבלה ניתן לראות מספר מצומצם שלהן.

מספר פסיקה	שימוש
0x00-0x07	נוצרת על ידי המעבד בעת שגיאות למיניהן.
0x08-0x0f	נוצרת על ידי רכיבי חומרה שונים, ביניהם שעון המחשב, כונני דיסקים ופלופי והמקלדת.
0x10	שירותי וידיאו. יש להעביר באוגר ah את המספר של התת-הפסיקה.
0x11	מחזיר את רשימת ההתקנים במחשב.
0x12	מחזיר את כמות הזיכרון במחשב.
0x13	דרך פסיקה זו ניתן לקרוא ולכתוב לדיסק. יש להעביר את תת הפסיקה באוגר ah.
0x16	דרך פסיקה זו ניתן לקרוא הקשות מקלדת. יש להעביר את תת הפסיקה באוגר ah.
0x19	פסיקה זו נוצרת על ידי הBIOS לאחר הPOST, ומתחילה את טעינת מערכת ההפעלה. קריאה לפסיקה זו תטען את מערכת ההפעלה שכתבתם מחדש.

הרצה

כעת ניתן לבצע בדיוק את אותן הפעולות שביצעתם בכדי להריץ את הקוד הקודם, ומערכת ההפעלה אמורה להדפיס לכם את המחרוזת "Hello World!" למסך.

מערכת ההפעלה, סימן 3

מניסיוני האישי עם תכנות מערכות הפעלה, אנשים שלא מבינים את הידע הנדרש בכדי לבצע זאת לא בדיוק מתלהבים ממסך שחור שכותב להם "שלום" בשחור ולבן. לכן, נקנח בקטע קוד שיבצע בשבילכם משימה - להפיק את השיר "99 Bottles of Beer on the Wall" (המקביל באנגלית ל-"99 תפוחים היו על העץ") בשפת סף טהורה.

הקוד מבוסס אך ורק על מושגים שנלמדו קודם, כך שניתן לוותר על הסבר:

```
[BITS 16]
[ORG 0x7C00]

call print_all
mov si,msg6
call prints
jmp $

print_all:
call print_first_half
dec byte [msg5+1]
cmp byte [msg5+1],'0'
je decten
call print_second_half
jmp print_all
decten:
cmp byte [msg5],'0'
je return
                  call print_second_half
                  call print_first_half
mov byte [msg5+1],'9'
dec byte [msg5]
call print_second_half
jmp print_all
print_first_half:
mov si,msg5 ;99
call prints
mov si,msg1 ;bootles of beer on the wall,
call prints
mov si,msg5 ;99
call prints
mov si,msg2 ;bottles of beer.
call prints
mov si,msg3 ;Take one down, pass it around
call prints
ret
print_second_half:
mov si,msg5 ;98
call prints
mov si,msg4 ;bootles of beer on the wall \r\n
call prints
ret
prints:
mov ah,0x0e
printc:
lodsb
cmp al,0
jz return
int 0x10
jmp printc
return:
ret
```

```
msg1 db " bottles of beer on the wall,",0
msg2 db " bottles of beer.",0
msg3 db " Take one down, pass it around, ",0
msg4 db " bottles of beer on the wall.",13,10,0
msg5 db "99",0
msg6 db " no bottles are left on the wall.",0

TIMES 510-($-$$) db 0
dw 0xaa55
```

השלכות על אבטחת המידע

כמה פנים לנושאים שהובאו בפרק בעלי השלכות לאבטחת מידע:

MBR

ה-MBR הוא רכיב פגיע ביותר ובעייתי מבחינת אבטחת מידע. ראשית, הוא אינו נמצא תחת מערכת הפעלה מסויימת, כך שגם מערכת ההפעלה המוקשחת ביותר לא יכולה להגן או לשחזר שינויים כאשר היא אינה הרכיב הראשון שמופעל. שנית, שינויים בו קשים לזיהוי על ידי משתמש הקצה - אין קבצים מוזרים שנשארים ולא קיימים תהליכים שהמשתמש לא מודע להם.

ניתן לבצע שני שינויים עיקריים ל-MBR, שהמטרה משתנה בין הרס לביון:

- וירוס בעל הראשות כתיבה ל-MBR יכול פשוט למחוק את כל הרשומות ולמנוע ממערכת ההפעלה להיטען. בסבירות גבוהה ביותר המשתמש (וגם הרבה טכנאי מחשב) יניח שכל הדברים שלו נמחקו, ויבצע פרמוט ללא בדיקה לגבי קיום תבניות שיזהו מחיצות של מערכות הקבצים על הדיסק. מתקפה זו לא דורשת ידע טכני רב או תחכום בקוד - פשוט צריך למלא את הסקטור הראשון בדיסק בזבל.

- מתקפה מתוחכמת יותר יכולה לבצע שתי מניפולציות עיקריות על ה-MBR - שינוי של הקוד שמוכל ב-MBR או הוספת מערכת הפעלה ראשונית שתופעל ורק אז תפעיל את מערכת ההפעלה העיקרית. ניתן לנצל שני שינויים אפשריים אלו בכדי לבצע מודיפיקציה מקודמת של הזיכרון או ה-Registry, אך אפשרות מעניינת במיוחד היא הפעלת מערכת הפעלה ראשונית שתתפקד כ-Hypervisor - באופן פשוט, רכיב שיוצר מכונה וירטואלית, ותפעיל את מערכת ההפעלה העיקרית בתוך הסביבה הוירטואלית הזו. ממצב זה ניתן לזייף ולשנות כל רכיב אפשרי, מהבקשות והתשובות שחוזרות מרכיבי החומרה עד זמן המחשב. באופן תיאורטי לא ניתן לזהות מתקפה זו מתוך מכונה נגועה כל עוד המתקפה ממומשת כראוי (קרי, לא מכילה חתימות ידועות שניתן לזהות או שגיאות שניתן ליצור). סוג מתקפה זה הודגם על ידי רוטקיט בשם המאוד מתאים - Blue Pill.

BIOS Interrupts

בדומה ל-IDT שממומש ב-Protected Mode, גם טבלת הפסיקות שמנוצלת על ידי ה-BIOS ניתנת לשינוי. בייחוד בגלל הפגיעות שאינרנטית ל-Real Mode, בו כל קוד יכול לגשת לחומרה באופן ישיר, כל תהליך יכול לקרוא ולשנות את טבלת הפסיקות שיושבת ב-1024 הבתים הראשונים בזיכרון. תוקף יכול לשנות כל רשומה בטבלה זו, ולבצע קפיצה לקוד שלו, במקום קוד של הBIOS. בדרך זו ניתן לממש Hook לכל פונקציה שנחשפת על ידי ה-BIOS Interrupts. אוסף המתקפות שניתן לנצל בדרך זו הוא גדול ביותר, אך גם ללא שינוי טבלת הפסיקות ניתן לממש אלפי מתקפות אחרות ב-Real Mode, כך שחוסר הגנה על אזור זה בזיכרון הוא האחרון שיש לדאוג לגביו.

BIOS Flashing

לאור העובדה שרכיבי BIOS צריכים לדעת לקרוא התקנים חדשים שנכנסים לשוק ולתמוך בפרוטוקולים חדשים שיוצאים באופן תדיר, יצרני רכיבי BIOS אפשרו לבצע תהליך שנקרא BIOS Flashing. בתהליך זה הקוד שקיים בצ'יפ ה-BIOS נמחק, וגרסה חדשה נכתבת אל הצ'יפ במקומו. ניתן לנצל מתקפה זו באופן דומה למתקפה שתוארה על הMBR, רק שבניגוד לMBR מתקפה זו תהיה כמעט בלתי אפשרית לזיהוי (מספיק להוסיף jmp קטן לאזור אחר בזיכרון, או להגדיר Boot Sequence קבוע) וניתן לנצל אותה בכדי להשיג שליטה כמעט מוחלטת על המחשב. יש לציין שה-ROM של ה-BIOS אינו הצ'יפ היחיד שמכיל מערכות בסיסיות עם גישה לחומרה שניתן לשכתב אותם. גם כרטיסי מסך מודרניים מכילים רכיבים כאלו, ואפילו כרטיסי רשת מסוימים ונתבים ניתנים לניצול דרך Flashing.

סיכום

תכנות מערכות הפעלה הוא עולם מעניין ומסובך. בכדי להגיע למערכות הפעלה ברמה תפקודית גבוהה יש להשקיע אלפי שעות ולצבור ידע רב. מצד שני, זהו תחום מרתק ששליטה בו נותנת כלים רבים. רק לאחר התעסקות מעמיקה עם תכנות מערכות הפעלה ניתן להבין מדוע ההערכה היא שלקח 14,000 **שנות** אדם בכדי לפתח את Debian 2.2 (שיצאה ב-2000), ומדוע מערכת ההפעלה Windows XP, שבמונחים מודרניים נחשבת ישנה, מכילה למעלה מ-45 מיליון שורות קוד. אין גבול לרמת העומק שניתן לרדת אליו בנושא זה, וכל המרבה, הרי זה משובח. בפרק זה סקרנו את הצעד הראשון בדרך למערכת הפעלה מתפקדת - יצירת מערכת הפעלה בסיסית ב-Real Mode ושימוש ב-BIOS Interrupts בכדי להדפיס מחרוזות למסך.

על המחבר

שמי עידן פסט, בן 17 מרעות, ואני עובד כבודק חוסן בחברה לאבטחת מידע לאחר שסיימתי את לימודי התיכון שנה שעברה. המגזין הזה היווה לי מקור מידע מעולה ונגיש, והרגשתי צורך לתרום חזרה בכדי לקדם אנשים אחרים בתחום. זהו המאמר הראשון שאני מפרסם במגזין זה ובכלל, ואשמח לכל תגובה אפשרית. ניתן ליצור איתי קשר בכתובת idanfast AT gmail.com.

קישורים לקריאה נוספת:

- מדריכים של אינטל. כבד מאוד, אבל בהחלט שווה:
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

- דוקיומנטציה על מספר רב של פקודות x86:
http://faydoc.tripod.com/cpu/index.htm

- רשימת האינטרפטים של ראלף בראון - מומלץ:
http://ctyme.com/rbrown.htm

- מדריך למערכת הפעלה דמויית UNIX:

http://www.jamesmolloy.co.uk/tutorial_html/index.html

- פרוייקט Linux From Scratch. מעולה ללמידה מעמיקה על הארכיטקטורה של Linux:

http://www.linuxfromscratch.org

- הקוד של הגרסה הראשונה של הקרנל של לינוקס. אם רוצים אפשר לקרוא את הקוד של גרסאות חדשות יותר, אבל הוא נהייה שם יותר מסובך בגלל אופטימיזציות וכו':

ftp://ftp.kernel.org/pub/linux/kernel/v1.0/

- מקור מדהים של מידע בנושא, בעיקר הקהילה:

http://wiki.osdev.org

- MSDN של מיקרוסופט - מאגר ידע עצום. מכיל הרבה רעיונות ותיעודים:

http://msdn.microsoft.com/en-us/library/ms123401.aspx

- ללא ספק הויקי המקיפה ביותר לגבי הפצה מסוימת של לינוקס, ומכילה המון מידע אודות כלים למינהם וכו'. עוזר בצורה עקיפה:

https://wiki.archlinux.org/index.php/Main_Page

- רשימה של פרוייקטים שאנשים עשו ב-OSDEV. מביא רעיונות מגניבים:

http://wiki.osdev.org/Projects