digital whisper
Technologic Papers

Booty Call - When the Firmware Answers the Wrong Ring

מאת רן אברהם

הקדמה

אני מניח שכל מי שקורא את המאמר הזה מכיר שקיים תהליך קסום שהופך כל רכיב טכנולוגי מסוים מקופסא מלאה בחוטים לפלא שאני יכול לשחק בו פורטנייט.

התהליך נקרא - Booting, ובו המחשב שלנו מעלה את מערכת ההפעלה בכדי שנוכל להשתמש בה. אף פעם לא התעמקתי יותר מדי בתהליך הזה, פשוט לחצתי על כפתור ההדלקה והמשכתי עם חיי כרגיל, אבל תמיד תהיתי לעצמי מה באמת הולך שם מאחורי הקלעים, והתחשק לי לבחון כמה עמוק אני יכול להיכנס ל- rabbit hole הזה שנקרא boot.

מרגיש לי שעם הזמן יש יותר ויותר הבנה על כמה תהליך ה-boot פגיע וחשוב להגן עליו, אבל עדיין לדעתי יש קהל שלא מבין את זה, אפילו יצרני משחקי מחשב (לדוגמא Riot) מחייבים במנגנון ה-anti cheat שלהם הפעלה של פיצ'ר הגנתי בשם secure boot ושיהיה קיים צ'יפ בשם TPM (ארחיב על שניהם בהמשך). זה רק בא להמחיש כמה חשוב להבין את הסיכונים, וכמה הם רלוונטיים גם לפעילות השוטפת שלנו על המחשב, שמתרחשת הרבה אחרי העלייה של המחשב.

במאמר זה נסקור בצורה מלאה את תהליך העלייה של המחשב שלנו, מהלחיצה על כפתור ההדלקה עד לשלב בו אנחנו נדרשים להכניס את שם המשתמש והסיסמא שלנו. בנוסף, ניגע בפיצ'רים הגנתיים שקיימים עבור התהליך ונראה איך אנחנו יכולים לשבור אותם. אסביר כיצד באמצעות תמונה ניתן לעקוף כל פיצ'ר הגנתי שקיים ואציג מתווה תקיפה שמנצל חולשה שרלוונטית לכולם.

Firmware - BIOS & UEFI

What Is Firmware? And Where Is It Saved?

Firmware היא תוכנה שמוטמעת בתוך רכיבי חומרה, בכל מכשיר טכנולוגי יש firmware - ממצלמות לטלוויזיות ולמדפסות. Firmware מהווה הגורם המגשר בין החומרה של המכשיר הטכנולוגי לחלקים יותר high level כמו מערכת ההפעלה. בעזרת firmware ניתן להעלות את המערכת בשלבים המאוד מוקדמים שלה כששום דבר לא רץ עדיין ולהתממשק עם רכיבי החומרה. שני הסוגים העיקריים של firmware שנדבר עליהם פה הם BIOS ו-UEFI.

ה-firmware חייב להיות מותקן איפשהו על המערכת שלנו, בעבר הוא היה מותקן על צ'יפ בשם ROM (read only memory) אבל בגלל שהבינו שיש צורך בצ'יפ יותר מתקדם שיתמוך בעדכונים של ה-firmware, כיום המצב מעט שונה.

ישנו צ'יפ ייעודי מסוג NVRAM, כלומר RAM ששומר את המידע שבתוכו גם לאחר שהמחשב נכבה, כלומר כאשר היה איבוד של זרם. ניתן גם לכתוב לתוכו, בניגוד ל-ROM שרק אפשר לקרוא ממנו. סוגים נפוצים של NVRAM הם SPI flash (serial peripheral interface) ו-EEPROM (electrically erasable programmable read-only memory). בדרך כלל ה-firmware ישמר על אחד מאלה עם עדיפות ל-SPI flash. לי פחות חשוב להתייחס להבדלים ביניהם ולאיך בדיוק הם עובדים, אבל חשוב להכיר את הקונספט, כי בהמשך המאמר וגם במאמרים אחרים יתייחסו למקום שבו נשמר ה-firmware בתור SPI flash וכדומה.

Different CPU States

במהלך הריצה של ה-firmware (ובכללי גם במערכת ההפעלה) יש שימוש במצבים שונים של ה-CPU, חשוב לי להבהיר את ההבדלים המרכזיים בין שני סוגי המצבים העיקריים ש-CPU רץ בהם. בהמשך נראה כיצד נעשה המעבר בין המצבים ומתי.

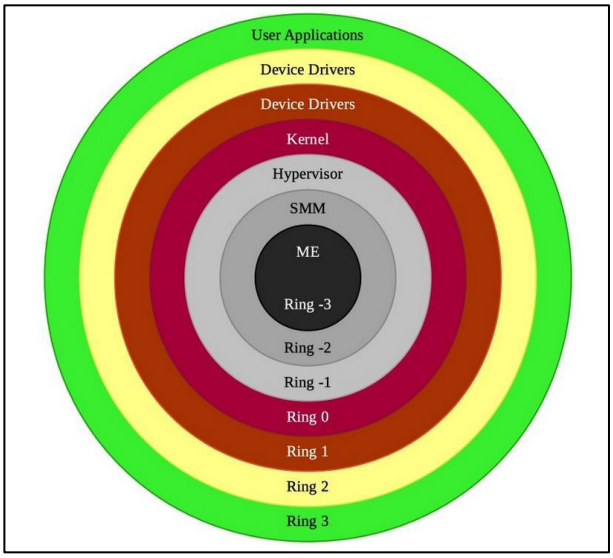
המצבים שאליהם אני מתייחס הם real mode ו-protected/long mode:

Real mode	Protected mode/long mode
16 Bit	32 / 64 Bit
אין שום זיכרון וירטואלי, כלומר גישה לזיכרון אומרת גישה למיקום האמיתי בזיכרון	יש זיכרון וירטואלי לא ניתן לגשת לכתובת הפיזית ישירות

מרחב הכתובות מוגבל ל-Bit20 כלומר MB1	ב-Bit32 מרחב הכתובות הוא בגודל GB4
אין שום הגבלה לאיזה כתובות ניתן לגשת ולאיזה לא, אין שום הגנה	ישנה הגבלה לאיזה כתובות ניתן לגשת, לא ניתן לגשת לכל כתובת שנרצה

SMM - System Management Mode

עד עכשיו בטח הכרתם את ring 0 ו-ring 3 שהמעבד שלנו מיחצן ושמערכת הפעלה שלנו משתמשת בהם, תהליך ה-boot חושף אותנו ל-ring-ים קונספטואליים שליליים. אחד מהם הוא ring -2 שנקרא SMM.



[תרשים מתוך מאמר ב-Medium]

זהו מצב ריצה מיוחד שנועד לטיפול במשימות שצורכות הרשאות גבוהות במערכת, כלומר גישה ושליטה ישירה בחומרה ברמה הגבוהה ביותר, לדוגמא ניהול רכיב ה-TPM (ארחיב עליו בהמשך), ניהול של הזרם שמגיע לרכיבים במערכת, וניהול פונקציות אבטחה של המעבד כמו כיבוי בעת התחממות יתר.

SMM נועד לשימוש רק על ידי ה-firmware של המערכת, לא על ידי כל אפליקציה או תוכנה. היתרון העיקרי של מצב זה הוא ש-SMM מציע סביבת הרצה מבודדת לגמרי שפועלת על המערכת שלנו. כאשר המעבד נכנס למצב SMM כל התהליכים שרצו עד עכשיו מפסיקים את ריצתם עד שהמעבד מסיים את ריצתו במצב SMM.

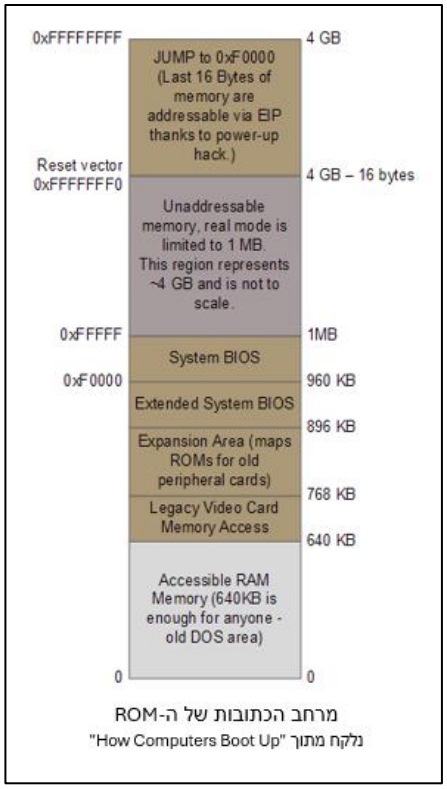
ניתן להיכנס למצב SMM על ידי קריאה ל-SMI - System management interrupt. ניתן לקרוא ל-SMI בעזרת פין פיזי של ה-CPU, כלומר אחד מהפינים הפיזיים של CPU מעיד על שליחה של SMI, וברגע שהוא נשלח ניכנס למצב SMM. ברגע שה-CPU נמצא במצב SMM כל interrupt אחר מבוטל. בכדי לצאת ממצב SMM

המעבד משתמש בהוראת RSM (resume from SMM) וניתן לקרוא לה רק מתוך מצב SMM, הוראה זו אומרת למעבד לטעון מחדש את המצב שהיה קודם ולחזור להרצה שהפסיק. קוד SMM רץ בזיכרון מוגן בשם SMRAM, כלומר SMM רץ בזיכרון בצורה שקופה לכל תהליך אחר - כמו מערכת ההפעלה לדוגמא. מרחב הכתובות הזה מכיל את הקוד ל-SMI handler ואת המידע שהקוד צריך. SMRAM ניתן לנעילה ככה ששום תהליך אחר לא יכול לגשת לזיכרון הזה.

SMRAM מכיל כמה רכיבים מרכזיים:

- SMBASE - זו כתובת הבסיס של ה-SMRAM, אוגר ב-CPU מכיל את הכתובת הזו.
- SMI Handler code - זה הקוד הראשון שרץ בעת קבלת SMI.
- State saved area - כאשר מצב ה-CPU עובר ל-SMM המצב הקודם של ה-CPU נשמר פה.

הגודל הדיפולטיבי של SMRAM הוא 64 KB, ערך ה-SMBASE הדיפולטי בעת עליית המערכת הוא 0x30000. המעבד מחפש את ההוראה הראשונה של ה-SMI handler בכתובת [SMBASE + 0x8000]. הוא שומר את ה-state הקודם של המעבד בין הכתובות: [SMBASE + 0xFFFF] - [SMBASE + 0xFE00]. תוקפים בדרך כלל ירצו לנצל את ה-SMM בגלל ההרשאות האינסופיות שיש לו. הם ינסו להגיע למצב של הרצת קוד ב-SMM בעזרת מציאת חולשה ב-firmware, שהוא זה שיכול להריץ בהרשאות אלה.



מרחב הכתובות של ה-ROM
נלקח מתוך "How Computers Boot Up"

לדוגמא ה-NSA (national security agency של ארה"ב) השתמש בחולשות zero day שמצאו על ניצול ה-SMM בכדי להשיג persistence על המערכת, הם עשו זאת במגוון מבצעים שלהם כגון: DEITYBOUNCE, ו-IRATEMONK.

פחות אתייחס למתקפות על ה-SMM במהלך המאמר, אבל היה חשוב לי להסביר על המצב הזה כי הוא מהווה חלק חשוב מהתהליך.

BIOS

סוג ה-firmware הראשון שאדבר עליו הוא Basic input output system, התפקיד החשוב ביותר של ה-BIOS הוא לטעון את מערכת ההפעלה. BIOS מהווה גורם מגשר בין כל רכיבי החומרה שמחוברים למחשב. מערכת הפעלה חייבת שיהיה כזו תוכנה שתגשר בשבילה, מכיוון שיש כל כך הרבה סוגי חומרה שונים, אז היא בעצמה לא יכולה לבצע את הגישור הזה בצורה כללית לכל סוגי החומרה, היא חייבת תוכנה ייעודית לכל חומרה. אחלק את ההריצה של BIOS לכמה שלבים:

Reset vector

כאשר המחשב נדלק אין ל-CPU שום instructions לבצע, אבל האוגר EIP שומר בתוכו את ה-reset vector שזו כתובת שנמצאת 16 bytes לפי סוף מרחב הכתובות של ה-flash memory, בכתובות זו תהיה הוראת JMP למיקום האמיתי. בכללי האוגר EIP הוא ה-Instruction Pointer, שמייצג את הכתובת לפעולה הבאה שיש ל-CPU לבצע. ניתן לראות בהמשך דיאגרמה שמציגה דוגמא למבנה של ROM שמכיל את הקוד של ה-BIOS.

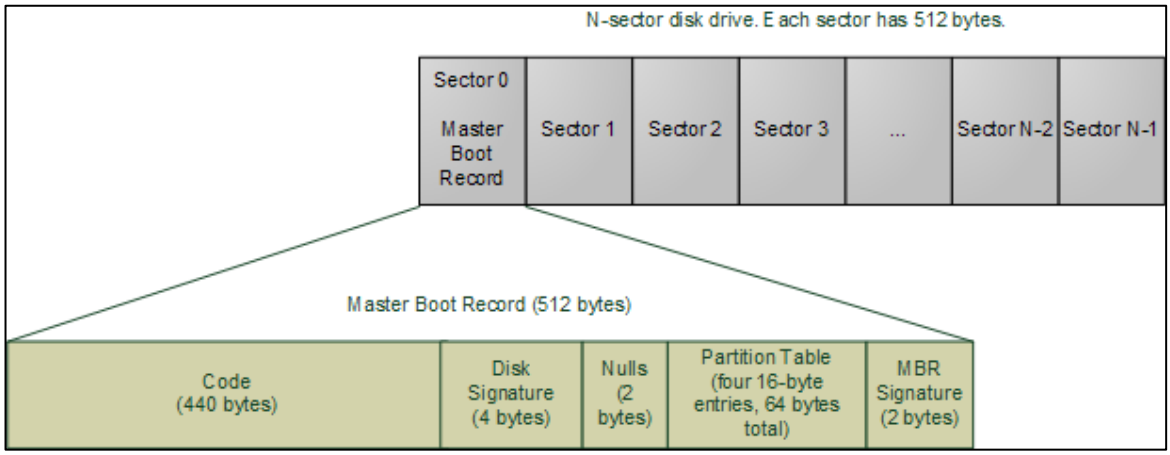
POST

לאחר מכן ה-CPU מתחיל להריץ את הקוד של ה-BIOS שמיד מתחיל לבצע בדיקות על החומרה של המחשב ורכיבים שמחוברים אליו, בדיקה זו נקראת POST (Power on self test). ראשית נבדק אם הכרטיס הגרפי מתפקד כראוי - אם ישנה בעיה יושמע צליל והתהליך יקטע. לאחר מכן יופיע הלוגו של חברת ה-motherboard ויתבצעו שאר הבדיקות ובמקרה תהיה בעיה תופיע הודעה בהתאם על המסך - לדוגמא אם אין מקלדת מחוברת. BIOS-ים מודרניים יוצרים טבלאות שמתארות את החומרה במחשב לפי פורמט שנקרא: Advanced Configuration and Power Interface. טבלאות אלו משומשות על ידי ה-kernel.

MBR

לאחר ה-POST ה-BIOS רוצה להעלות את מערכת ההפעלה, אז הוא מחפש boot device שעליו תימצא מערכת ההפעלה הרלוונטית. היא יכולה להימצא במגוון רכיבים במחשב: HD, CDs, USBs וכו'. סדר הבדיקה של הרכיבים ניתן לקסטומיזציה על ידי המשתמש. ברגע שנמצא רכיב מתאים ה-BIOS קורא את ה-sector הראשון של האחסון, כלומר את ה-512 bytes הראשונים. Sector זה נקרא גם ה-MBR (master boot record) והוא מאופיין בעזרת חתימה של שני ה-bytes האחרונים שלו עם הערך - 0xaa55.

ה-BIOS טוען את ה-MBR לזיכרון לכתובת 0x7c00. שני החלקים העיקריים של ה-MBR הם הקוד הספציפי למערכת ההפעלה, שיכול להיות loader של windows או של linux כמו GRUB. החלק המרכזי השני של ה-MBR הוא Partition Table, המכיל טבלה שמראה כיצד הדיסק חולק - תיאור של ה-partitions השונים, זאת בכדי שנוכל להריץ כמה מערכות הפעלה על אותו הדיסק או אפילו סתם לחלק אותו לוגית לכמה volumes. הגודל של הקוד ב-MBR הוא 446 bytes והגודל של ה-partition table הוא 64 bytes (היא יכולה להכיל רק ארבע partitions).



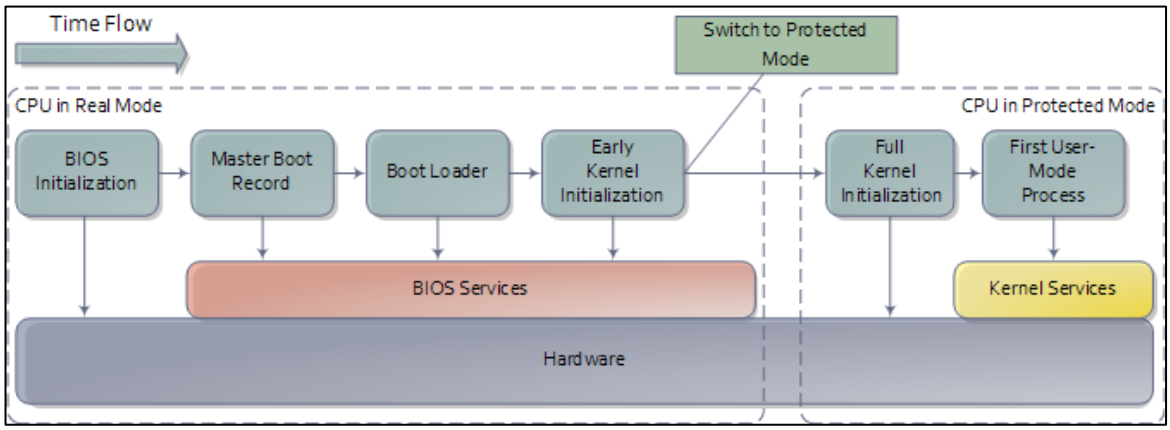
["How Computers Boot Up" נלקח מתוך]

VBR & IPL

בגלל הגודל הקטן של ה-MBR יש צורך לקרוא ל-stage 2 loader שנקרא גם IPL (initial program load), בדרך כלל זה יהיה boot sector שימצא בתחילת active partition (לעומת MBR שנמצא בתחילת ה-HD) . ה boot sector- נקרא גם ה-VBR (volume boot record). אבל גם יכולה להיות קפיצה ל-sector שהכתובת שלו מוטמעת (hardcoded) בתוך ה-MBR.

לאחר מכן קוד ה-stage 2 קורא מאותו ה-partition את מערכת ההפעלה ומתחיל אותה, בשלב זה בכל מערכת הפעלה התהליך מעט שונה. מערכת ההפעלה ניגשת ל-BIOS ושולפת ממנו את המידע שהשיג בתהליך ה-POST על הרכיבים השונים במערכת, ונעשית בדיקה האם יש את ה-driver-ים המתאימים לכל רכיב. אם הכל נבדק ואושר אז מערכת ההפעלה תמשיך בתהליך נפרד משלה עד שתגיע לתפקוד מלא. יש לציין כי מכיוון שה-BIOS הינו מוצר legacy אז כל הריצה שלו תהיה ב-real mode ואילו רק בשלב מאוחר יותר כשמתחילה ההתערבות של מערכת ההפעלה ב-stage 2 נעשה מעבר ל-protected mode. ניתן לעבור ל-protected mode בעזרת שינוי הערך של האוגר CRO[PE] ל-1, במקום 0 ל-real mode. חשוב לנו לעבור ל-protected mode מכיוון שב-real mode אנחנו מוגבלים רק ל-1MB של זיכרון, אז כשנרצה לטעון את ה-kernel נתקל בבעיה.

בכדי להתגבר על הבעיה שצריך גם פונקציות של ה-BIOS וגם לטעון משהו ששוקל יותר מ-1MB אנחנו נעבור בין protected mode לבין real mode מספר פעמים ורק כאשר נצטרך (נקרא גם unreal mode). ולבסוף נעבור לגמרי ל-protected mode.



[תיאור מלא של תהליך העלייה עם BIOS, נלקח מתוך "How Computers Boot Up"]

UEFI

UEFI (Unified Extensible Firmware Interface) הינו המחליף המודרני של BIOS, הוא מהווה שדרוג משמעותי ל-BIOS גם מבחינה תפעולית וגם אבטחתית.

כיום UEFI הינו המוצר הנפוץ מבין השניים שנמצא כמעט בכל מחשב. הגרסה הקודמת שלו נקראה EFI, לכן אולי נתקל הרבה פעמים במילה EFI (אותם ראשי תיבות רק בלי unified). אפשר ממש לחשוב על UEFI בתור מיני מערכת הפעלה, ל-UEFI יש שירותים שהוא מספק, driver-ים שנטענים אליו והוא גם יודע להתמודד עם מערכת קבצים ולקרוא ממנה.

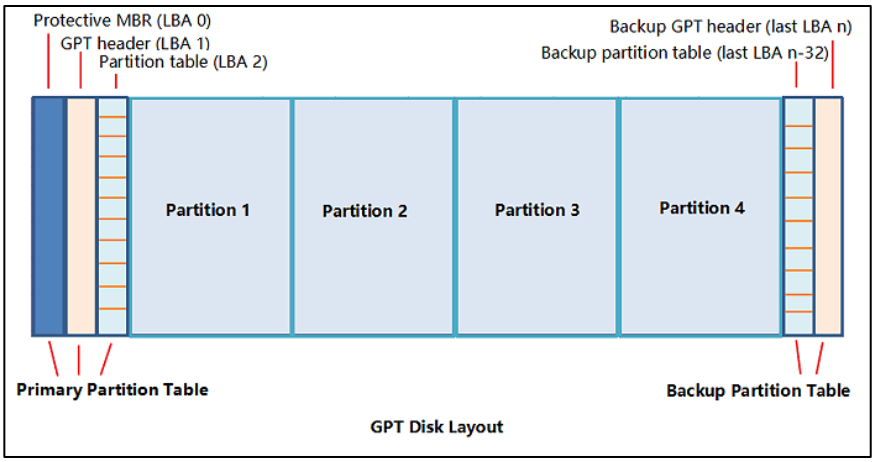
תהליך ה-BOOT עם UEFI הוא מעט שונה מאשר עם BIOS וגם יותר מורכב, ההבדל המשמעותי בין השניים הוא ש-UEFI לא צריך את ה-MBR או את ה-VBR. אלא הוא משתמש בקוד שמוטמע בתוך ה-NVRAM ו-GUID partition table (GPT).

GPT

UEFI משתמש ב-GPT בכדי לחלק את הדיסק בצורה מסודרת. כלומר לכל partition יש GUID שמייחד אותה ו-UEFI נעזר ב-GPT בכדי למצוא את הקבצים הרלוונטיים שהוא צריך בכדי להעלות את מערכת ההפעלה.

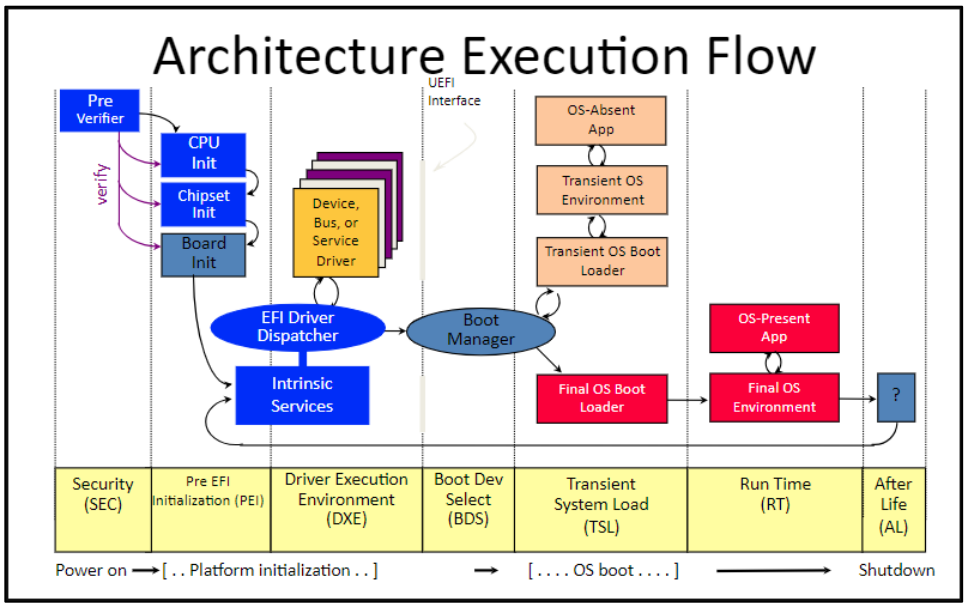
החלקים המרכזיים ב-GPT שחשוב להכיר:

- Protective MBR - בגלל ש-BIOS הינו firmware לא מאוד מתוחכם הוא יודע לעבור רק עם MBR ואם לא קיים MBR על דיסק הוא במקרים מסוימים יכול למחוק את כל התוכן של הדיסק במטרה לנסות לקרוא אותו. לכן ב-GPT קיים Protective MBR, שמקרה בו מישהו ירצה להשתמש ב-GPT ביחד עם BIOS ה-firmware ידע להתמודד עם הדיסק.
- GPT Header - זהו header שמכיל מידע על כל הדיסק. מידע כמו: כמות ה-partitions, GUID של הדיסק ועוד.
- Partition Table - רשימה של ה-partitions שקיימים על הדיסק, הנתונים העיקריים שיהיו על partition מסוים הם ה-GUID שלו, מתי והוא מתחיל ומתי הוא נגמר - כלומר באילו sectors.
- Backup - ישנו שכפול של ה-header ושל ה-partition table בסוף הדיסק. זאת בכדי לשמש כ-backup ובמקרה של כשל.



["What Is GPT Disk - EaseUS" נלקח מתוך]

כאשר נדליק את המחשב ה-CPU יפנה לכתובת של ה-reset vector (16 bytes מסוף ה-flash chip). שם יופיע קוד ה-initialization של ה-UEFI. בשלב זה נתחיל את תהליך העלייה של UEFI. תהליך העלייה בעזרת UEFI נקרא platform initialization. באתר של UEFI מצורף תרשים שמסכם את כל התהליך בצורה מאוד מפורטת. אציג אותו ואז אפרט על כל שלב בקצרה.



[נלקח מהדוקומנטציה של UEFI]

Security (SEC)

לקוד זה יש כמה מטרות עיקריות: להחליף את מצב הריצה של ה-CPU מ-real mode ל-protected mode, להשתמש בזיכרון שפנוי ב-CPU בכדי ליצור RAM זמני, להוות כ-root of trust למערכת - כלומר שיהיה ניתן לסמוך עליו, ולבסוף להעביר את המידע הנדרש לשלב הבא - PEI.

Pre EFI Initialization (PEI)

שלב ה-PEI משתמש רק במשאבים של ה-CPU בכדי להריץ EFI Initialization Modules. אלה Modules מאפשרים לבצע אתחול ראשוני למערכת כמו לדוגמא memory initialization. לבסוף PEI מריץ את סביבת השלב הבא - DXE.

Driver eXecution Environment (DXE)

זה השלב שבו רוב העלייה של המערכת נעשית, שלב ה-PEI נועד בכדי "להכין את השטח" לשלב ה-DXE. שלושת החלקים העיקריים בשלב ה-DXE הם DXE Core, DXE Dispatcher, DXE Drivers.

ה-DXE core אחראי על הרצה של שירותי boot ו-runtime. ה-dispatcher נועד בכדי למצוא ולהריץ DXE drivers בסדר הנכון. לבסוף ה-DXE drivers הם חלקי הקוד שמספקים את השירותים, דוגמא לשירותים ש-drivers מספקים הם: network access, thermal management ועוד.

עוד אחריות של ה-core היא למלא את ה-EFI System Table שמכילה pointers לשירותים ש-UEFI מספק ול-EFI Configuration Table. השירותים ש-UEFI מספק מתחלקים לשני סוגים: Boot Services & Runtime Services, שירותי ה-boot יכבו בעת סיום הריצה של UEFI, שירותי ה-runtime ימשיכו לרוץ גם לאחר סיום הריצה של UEFI וכשמערכת ההפעלה כבר השתלטה על העלייה של המחשב.

אפשר לחשוב על שלב ה-DXE כשלב העיקרי של UEFI, בזכותו UEFI הוא firmware כזה מתקדם.

BDS

השלב הבא הוא ה-BDS (Boot Device Selection), לאחר ההרצה של כל ה-DXE drivers השליטה מועברת ל-BDS והמטרה בשלב זה היא למצוא boot application (קובץ efi.), זאת מכיוון שהמערכת מוכנה להריץ מערכת הפעלה ורק צריכה למצוא את המיקום שלה. לכן פה יש שימוש ב-GPT בכדי למצוא את ה-ESP (EFI system partition), ה-ESP הוא partition נפרד עם מערכת קבצים מסוג FAT32. הוא מכיל בתוכו את הקבצים הרלוונטיים לעלייה של מערכת ההפעלה. הנתיבים לקבצים הרלוונטיים נשמרים כ-EFI variables.

TSL

השלב הבא נקרא TSL (Transient System Load), בו ישנה קריאה ל-boot loader , שירים את שאר מערכת ההפעלה ויעצור את השירותים של ה-UEFI boot.

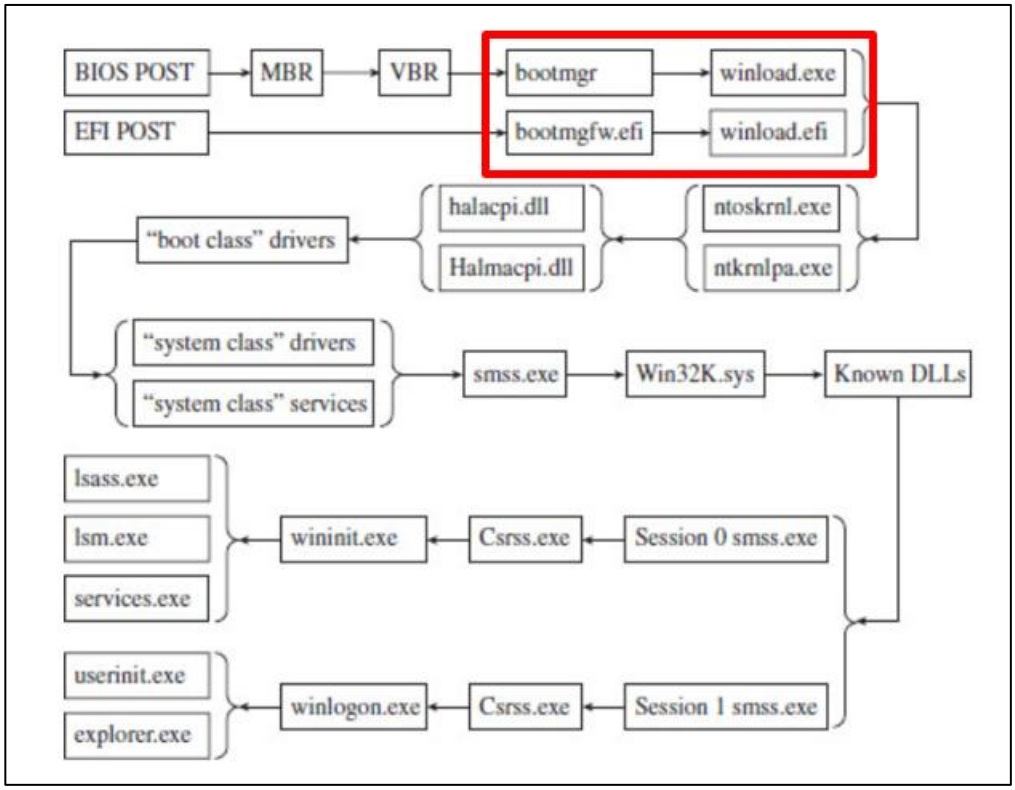
RT (Runtime)

לבסוף בשלב ה-RT (Runtime) סופית תעבור השליטה על המערכת למערכת ההפעלה. חלק מהשירותים של ה-UEFI יישארו זמינים למערכת ההפעלה כמו כתיבה של variables ל-NVRAM, אבל כעת מערכת ההפעלה לא תהיה חייבת לעבור דרך ה-UEFI בכדי לגשת לחומרה, בהרבה מקרים היא תעשה זאת בעצמה ישירות.

OS Level Booting

לאחר שה-Firmware סיים את עבודתו הוא יעביר את הריצה למערכת ההפעלה הרלוונטית, אני בחרתי להתייחס רק ל-Windows. בעיקר משום שאם הייתי מוסיף התייחסות גם ל-Linux המאמר היה יותר מדי ארוך.

Windows Boot



Boot Manager

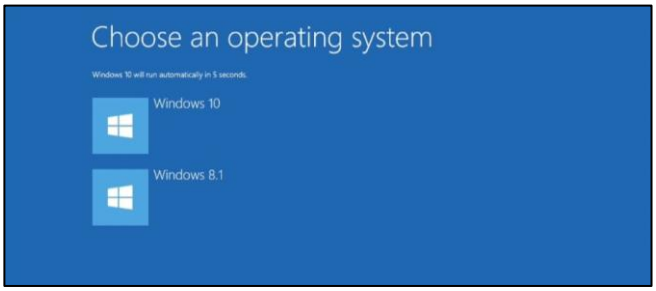
בכדי לתמוך גם ב-BIOS boot וגם ב-UEFI boot, windows מספקת ארכיטקטורת boot שמתאימה לשני הסוגים - בעיקר בכדי שיהיה ניתן להשתמש במערכת ההפעלה עם כמה שיותר סוגי חומרה. ב-BIOS לאחר הרצה של קוד ה-VBR, נגיע לקובץ bootmgr.exe. ב-UEFI לאחר שנפתח את ה-ESP partition ירוץ הקובץ bootmgfw.efi שנמצא בנתיב \EFI\Microsoft\Boot\bootmgfw.efi. כעת ה-boot manager לוקח בעלות על התהליך ומעלה את מערכת ההפעלה.

BCD

ה-boot manager משתמש במידע שנשמר ב-registry hive בכדי להתחיל את המערכת. ה-hive file נקרא BCD (Boot Configuration Data).

ניתן לראות את ה-BCD תחת נתיב הבא ב-registry: HKLM\BCD00000000, או אפילו בעזרת הקובץ bcdedit.exe שקיים דיפולטית במערכת ההפעלה.

BCD store תמיד יכיל אובייקט boot manager יחיד או כמה אובייקטים של boot loader. בעזרת ה-BCD ה-boot manager בוחר איזה boot loader להריץ, או לפי החלטה של המשתמש. במקרים בהם יהיה יותר מ-bootloader אחד על העמדה יוצג למשתמש צג לבחירת bootloader - פיצ'ר זה נקרא גם dual boot.



Hibernation

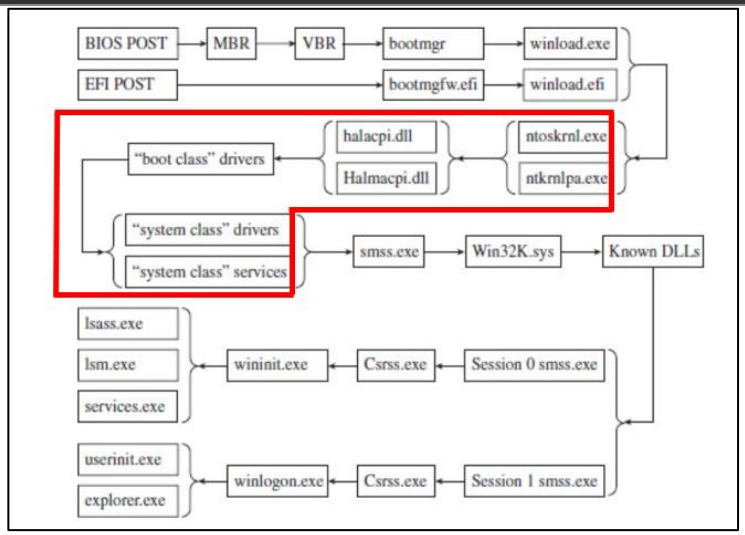
במקרה בו נשאיר את המחשב דלוק אבל לא ניגע בו הרבה זמן, המסך שלו "יהפוך לשחור" וניכנס למצב של hibernation - אפשר לחשוב על זה כמעין תנומת חורף. בגלל שלא באמת כיבינו את המחשב, כאשר נרצה לחזור לאותו מצב שבו עזבנו, תהליך ה-boot לא יהיה אותו הדבר. במצב הזה תהיה הגדרה ב-BCD על עליית מערכת מ-hibernation state, אז ירוץ Winresume.exe/Winresume.efi שמטרתו היא לגשת לקובץ Hiberfil.sys ולטעון את התוכן שלו לתוך הזיכרון ולהעלות את המערכת בדיוק כפי שהיית לפני ה-hibernation. כלומר נעביר את השליטה ל-kernel שיעלה את מערכת ההפעלה בדיוק באותה צורה שהיא עצרה ממקודם.

Boot Loader

אם אנחנו לא עולים מ-hibernation state ה-boot manager יריץ את ה-windows boot loader (או ייתן את הבחירה למשתמש במקרה של כמה boot loader קיימים), במערכות BIOS - winload.exe (המחליף של NTDLR- ה-bootloader הישן מאוד של windows) ובמערכות UEFI - Winload.efi. המיקום של קובץ ה-loader יצוין באובייקט בBCD.

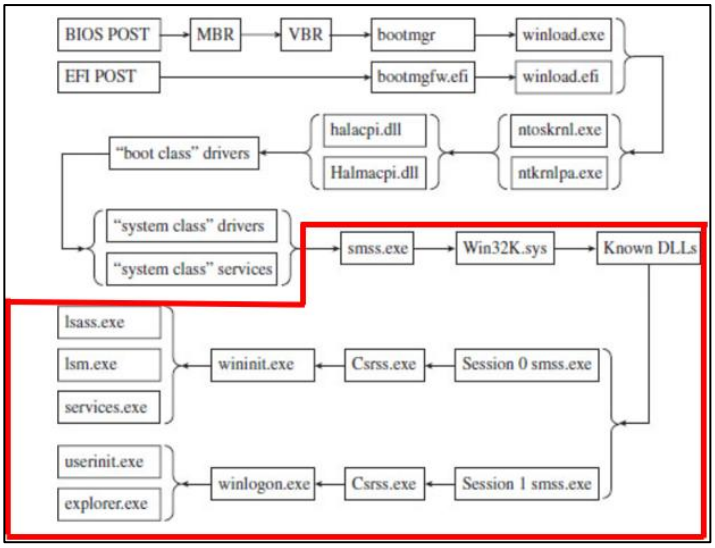
ה-windows boot loader הוא באמת הגורם שמעלה את מערכת ההפעלה במלואה, הוא מתחיל בלטעון את ה-registry hive - HKLM\SYSTEM שנמצא ראשית תחת system32. לאחר מכן הוא מוודא את החתימה של עצמו (nt5.cat) שנמצא תחת:

```
%SystemRoot%\System32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}
```



Kernel

לאחר וידוא החתימה, ה-boot loader יטען את ntoskrnl.exe לזיכרון. Ntoskrnl הוא ה-kernel של windows, שאחריותו לטעון את ה-registry, לטעון drivers ו-services רלוונטים לפי ערך ה-registry - SYSTEM\CurrentControlSet\Services, לאתחל את ה-system pagefile - C:\pagefile.sys ולטעון לזיכרון את hal.dll שמהווה כמעין שכבת אבסטרקציה בין ntoskrnl לחומרה. אחד הדברים האחרונים שעושה ntoskrnl הוא להעלות את ה-session manager.



Session manager - smss

session manager subsystem - smss.exe, הינו תהליך user mode רגיל, עם כמה הבדלים מרכזיים. ראשית הוא נחשב חלק ממערכת ההפעלה שניתן לסמוך עליו, שנית, הוא native application, כלומר הוא לא משתמש ב-WinAPI בכדי להתממשק עם מערכת ההפעלה, אלא רק ב-native API. WinAPI היא הדרך הרגילה שמוגשת על ידי Microsoft להתממשקות עם מערכת ההפעלה, לעומת זאת Native API הוא API לא מתועד שמהווה כבסיס של WinAPI - כלומר פונקציות של WinAPI ישתמשו בפונקציות של Native API "מאחורי הקלעים".

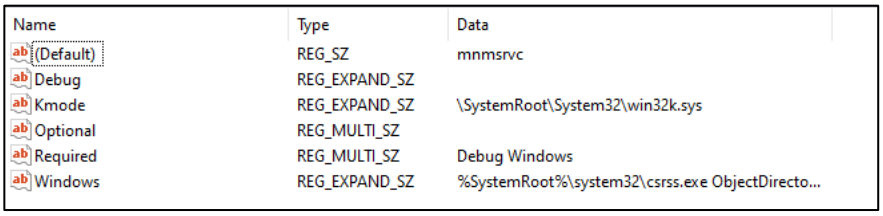
לכן smss יכול לבצע פעולות שלרוב התהליכים אין את היכולת לבצע, כמו יצירת security tokens - לכל משתמש יש token שמזוהה איתו, באמצעותו הוא מזדהה כאותו משתמש. smss לא משתמש ב-WinAPI בגלל שה-windows subsystem לא רץ כאשר smss רץ לראשונה, בעצם אחד התפקידים של smss הוא להריץ את ה-windows subsystem.

subsystem היא מין תת מערכת הפעלה שמאפרת התממשקות של קובץ הרצה עם אותה מערכת הפעלה – ה-windows subsystem שמכיל את WinAPI עדיין לא אותחל וזה תפקידו של smss לעשות זאת.

ל-windows subsystem יש שני חלקים: driver בשם Win32k.sys ורכיב user mode בשם csrss.exe. הוא מזהה אותם לפי ה-registry key:

```
HKLM\System\CurrentControlSet\Control\Session Manager\Subsystems
```

וה-values: Windows (user mode)/ו Kmode (kernel)



Name	Type	Data
(Default)	REG_SZ	mnmsrvc
Debug	REG_EXPAND_SZ	
Kmode	REG_EXPAND_SZ	\SystemRoot\System32\win32k.sys
Optional	REG_MULTI_SZ	
Required	REG_MULTI_SZ	Debug Windows
Windows	REG_EXPAND_SZ	%SystemRoot%\system32\csrss.exe ObjectDirecto...

ראשית smss טוען את Win32k.sys, ולאחר מכן הוא טוען את known DLLs לפי ערך ה-registry: .\HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs.

כעת smss מבצע את פעולתו העיקרית שהיא לאתחל את ה-session-ים, הוא יוצר שני instances של עצמו - session 0 & 1. Session 0 מייצג את תהליך ה-init (wininit.exe) ו-1 session מייצג את תהליך ה-logon, כלומר את החלק שקשור למשתמש (winlogon.exe).

לכל אחד מה-instances חייב שיהיה את חלק ה-user mode של ה-subsystem, לכן הם מריצים לפי ערך ב-registry את csrss.exe וכעת ניתן להריץ אפליקציות user mode שמשתמשות ב-WinAPI.

Wininit.exe שרץ ב-session 0 מתחיל שלושה תתי תהליכים:

- lsass.exe - The local security authority subsystem - מספק שירותי אימות למשאבים של מערכת ההפעלה, מנהל את פוליסת האבטחה המקומית.
- services.exe - the service control manager - טוען שירותי autostart, משתמש ב-lsass בכדי לאמת שירותים שלא רצים תחת system.
- lsm.exe - the local session manager - מנהל את כל ה-sessions שקיימים במערכת.

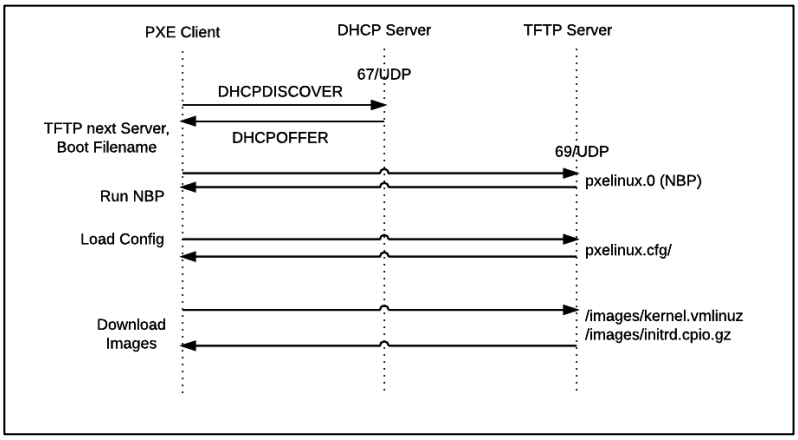
Winlogon.exe שרץ ב-session 1, אחראי לטפל ב-user logons. הוא מריץ את ה-logon interface - logonui.exe שמציג את מסך ההתחברות למשתמש. קובץ הרצה זה מעביר את ה-credentials שהמשתמש הכניס ל-lsass, אם תהליך האימות היה מוצלח, יורצו קבצי ההרצה הרלוונטיים תחת ה-values: UserInit & Shell שנמצאים תחת ה-key:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\
```

באופן דיפולטי UserInit מופנה ל-userinit.exe וערך ה-shell הוא explorer.exe. userinit.exe מריץ את נתיבי ה-startup וגם ערכי registry שמיועדים לרוץ בעת התחברות של משתמש.

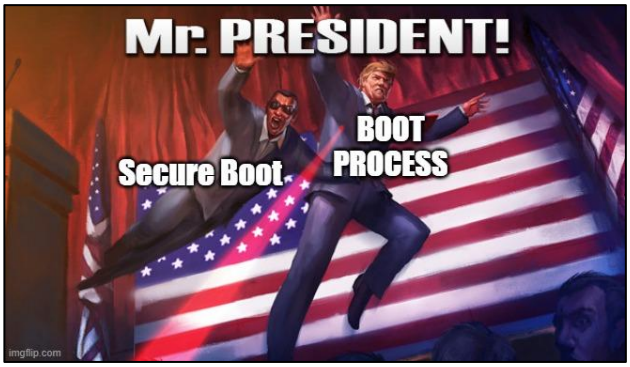
Network Boot (PXE)

זהו תהליך שבו המחשב שלנו יבצע boot דרך ה-network card ונקרא גם PXE. כלומר הוא פונה לשרתים רלוונטיים בכדי להשיג את קבצי ה-boot הרלוונטיים. כאשר המחשב שלנו יידלק נודיע ל-BIOS שלנו שאנחנו מעוניינים ב-network boot בעזרת הקשה על המקש המתאים (לרוב F12), אז נעזר בפרוטוקול DHCP ונשלח פקטת discover ונזהה את עצמנו בתור PXEClient ל-LAN שלנו. אז שרת ה-DHCP עונה לנו עם כתובת IP של שרת TFTP ושם של קובץ NBP (boot file) שנצטרך להוריד משרת ה-TFTP. לאחר מכן נפנה לאותו שרת בעזרת פרוטוקול TFTP ונוריד את הקובץ הרלוונטי ונריץ אותו, קובץ ה-NBP מוריד ומריץ קונפיגורציות, סקריפטים או images רלוונטיים בכדי להעלות את מערכת ההפעלה. קיימת גרסה משופרת יותר של PXE שנקראת iPXE, שמשתמשת בסקריפטים במקום בקבצי קונפיגורציה וכן יכולה להשתמש בפרוטוקול HTTP/HTTPS.



Securing The Boot Process

לאחר שהבנו כיצד באמת מתבצע תהליך ה-boot, הגיע הזמן להבין כיצד חשבו להגן על התהליך. ישנם הרבה פתרונות קיימים שמטרתם היא לאבטח את תהליך ה-boot במלואו, צירפתי את המרכזיים שלדעתי הכי נפוצים ומספקים את ההגנה העיקרית על התהליך.



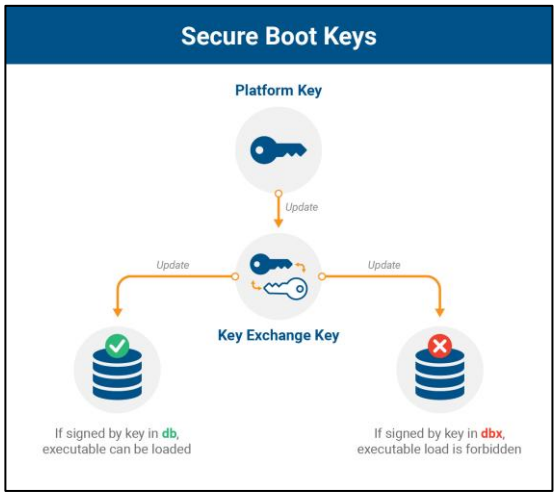
Secure Boot

זהו פיצ'ר אבטחתי שקיים רק על firmware-ים מסוג UEFI. בעזרתו ניתן לקבוע כי כל תוכנה שתרוץ בתהליך ה-boot תהיה חתומה על ידי ה-Original Equipment Manufacturer (OEM), כלומר מהשנייה שהמחשב נדלק, ה-firmware שלנו יבדוק את החתימה הדיגיטלית של ה-UEFI firmware drivers, EFI applications וקבצים של מערכת ההפעלה.

Secure boot בנוי על PKI בכדי לאמת את הרכיבים שנטענים.

Secure boot משתמש בכמה databases ששמורים על המחשב ב-NVRAM ונחרטים שם בעת יצירתו:

- Signature database (db) - מאגר של certificates של UEFI applications, operating system loaders and UEFI drivers. כל תוכנה שנמצאת במאגר זה מורשת להיטען בתהליך ה-boot.
- Revoked signatures database (dbx) - מאגר דומה ל-db רק שכל תוכנה שמופיעה במאגר זה אסורה לטעינה במערכת.
- Key enrollment key database (kek) - מאגר שמכיל מפתחות של כל מי שיכול לעדכן את המאגרים db ו-dbx.
- Platform key (pk) - זהו מפתח יחיד שמהווה כroot of trust לעדכון ה-kek.



[נלקח מתוך Cyber Security News]

תחום האחריות של UEFI secure boot נגמר לאחר בדיקת והרצת ה-bootloader, עד לשלב זה תיבדק החתימה של כל אלמנט שבתהליך שנטען. במהלך התהליך ישנה פנייה אקטיבית ל-db ו-dbx, אם ירוץ רכיב שהחתימה שלו מופיעה ב-db או שהחתימה שלו מופיעה ב-dbx אז הוא יחסם וייעצר תהליך ה-boot.

Intel Boot Guard

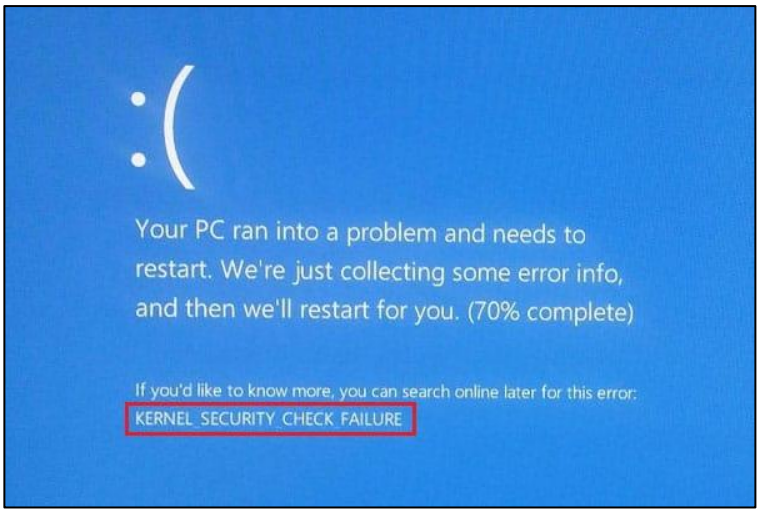
זה הוא פתרון הגנה מבית intel שנועד בכדי לוודא את תקינות ה-firmware שעולה. בהרבה מקרים ה- root of trust נקבע על ה-firmware מבלי לבדוק את החתימה שלו מראש, בגלל מחשבה שאולי זה הגורם הכי בטוח במערכת שלנו, אבל עם הזמן תוקפים התקדמו והצליחו גם להשפיע על החלקים הנמוכים ביותר של המערכת שלנו.

כאשר המערכת עולה ה-CPU מתחיל את ה – ME -Intel Management Engine, הקוד שלו שמור ברכיב ROM שמוודא את החתימה של ה-bootguard ACM module. לאחר מכן module זה מוודא את החתימה של ה-

UEFI firmware. בכדי לוודא את ה-firmware, ה-CPU משתמש במפתח הצפנה שמוטמע בתוך החומרה עצמה.

Windows Trusted Boot

זהו המשך של תהליך ה-secure boot של UEFI, trusted boot הוא הפתרון של windows לאימות שאר תהליך ה-boot שיותר רלוונטי ספציפית ל-windows. אם secure boot פועל עד רמת ה-bootloader אז Trusted boot ממשיך את פעולתו ומאמת את ה-image של ה-kernel וכל הרכיבים שבאים לאחר מכן עד לטעינת ה-third party drivers. שלב זה נעזר ב-TPM בכדי לוודא את החתימות של כל הרכיבים.



במקרה ו-trusted boot ייתקל ברכיב שאינו מאומת הוא אוטומטית יציג את מסך ה-BSOD.

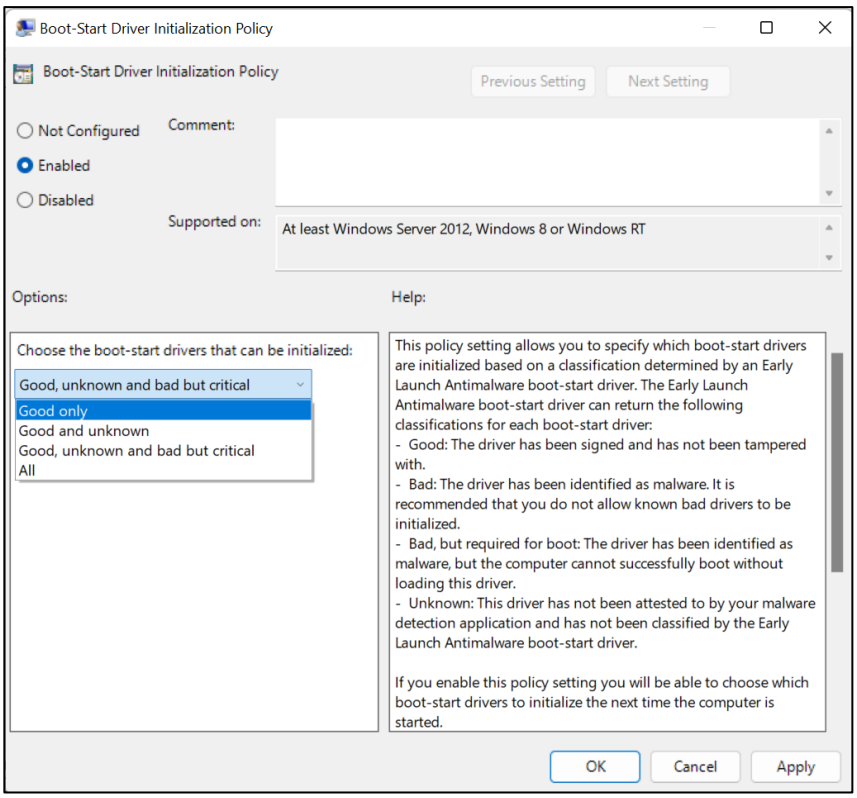
ELAM

בכדי להילחם ב-boot start drivers זדוניים, מייקרוסופט יצרה driver (WdBoot.sys) שמתחיל לפני כל שאר ה-drivers שעולים בעת הרצת ntoskrnl - ELAM driver מאמת כל driver אחר ביחד עם ה-dependencies שהוא מייבא. ELAM driver יכול להחזיר את התוצאות הבאות: ה-driver אינו מוכר, ה-driver מאושר, ה-driver נקבע כזדוני, ה-driver נקבע כזדוני אבל הוא קריטי לתהליך ה-boot והמחשב לא יעלה בלעדיו. windows bootloader טוען בעזרת ה-registry hive הרלוונטי - HKLM\ELAM, את החתימות הרלוונטיות לרשימות ה-allow/block. מתבצע unload ל-hive ברגע שה-ELAM driver סיים את פעולתו. ניתן לקנפג את ה-ELAM בעזרת חוקות GPO, כלומר ניתן לקבוע מה רמת האבטחה שנרצה - האם לטעון רק drivers שעברו את האישור או גם לטעון כאלה שלא ידועה לנו חתימה עליהם.

הקונפיגורציה של ה-ELAM נשמרת תחת ה-registry key הבא:

```
HKLM\SYSTEM\CurrentControlSet\Policies\EarlyLaunch\DriverLoadPolicy.
```

ELAM מאפשר גם לתוכנות AV לטעון ELAM drivers משלהם, לדוגמא mfeelamk.sys של mcafee.



צירפתי צילום מסך של הגדרת ה-GPO שלפיה ELAM קובע איזה driver לחסום. ההגדרה מופיעה תחת הנתיב Computer Configuration\Administrative Templates\System\Early Launch Antimalware. ובהגדרה הזו אפשר לקבוע אם רק driverים מאושרים יטענו או שבכלל אפשר לטעון כל driver אפשרי.

TPM

Trusted platform module, הוא צ'יפ שנועד בכדי לספק פעולות קריפטוגרפיות בסיסיות. TPM מותקן ישירות על ה-motherboard של המחשב ומתקשר ברמת החומרה. מחשבים שמשלבים גם TPM יכולים ליצור מפתחות הצפנה ואז להצפין אותם ככה שרק ה-TPM יכול לפענח אותם, תהליך זה נקרא "wrapping" or "binding" a key. לכל TPM יש מפתח הצפנה ייעודי שנקרא storage root key ששמור בתוך ה-TPM עצמו, החלק הפרטי של המפתח נקרא ה-endorsement key והוא בשום מצב לא יהיה חשוף לשום רכיב אחר במערכת. הוא לא ישמר באותו זיכרון שנשלט על ידי מערכת ההפעלה. ה-TPM מהווה חלק מרכזי באבטחת מערכת ההפעלה, וגם בתהליך ה-boot.

Measured Boot

Measured boot נועד בכדי לבצע "measure" לכל אובייקט שרץ בעת תהליך ה-boot. כלומר הוא מחשב את חתימות hash ושומר את התוצאה ב-TPM בצורה מאובטחת, ככה שבסוף התהליך יהיה ניתן לעבור אחר כל החתימות לוודא את תקינותן. ב-Measured boot לא נעשית בדיקה האם החתימה טובה או רעה אלא רק נעשה תיעוד בכדי לאמת את כל החתימות לאחר מכן. החתימות נשמרות ב- Platform Configuration Registers (PCRs) בתוך ה-TPM. PCR הוא מעין אוגר שנמצא בתוך רכיב ה-TPM, שכל אחד כזה מכיל hash אחר (sha1 או sha256). במהלך תהליך ה-boot כל חלק יתועד לתוך PCR רלוונטי, כאשר אנחנו כותבים לתוך PCR אנחנו אף פעם לא מחליפים את הערך אלא עושים לו extend. ניקח את הערך הישן ואת החדש, נחבר אותם ונבצע hash. אוגרי ה-PCR של ה-TPM מאותחלים ל-0 כאשר המערכת עולה, והם מתמלאים בחתימות של רכיבים שעולים בתהליך ה-boot.

PCRs מ-0-15 מיועדים לשימוש סטטי, הם נועדו בכדי לתת למערכת ההפעלה שקיפות לגבי המצב של עליית המחשב. PCRs 17-22 הם יותר דינמיים והם נועדו בכדי לבצע attestation לבדיקת האמינות של רכיבי מערכת ההפעלה (גם מצורפת טבלה עם פירוט). Attestation זו פעולה שמטרתה לוודא את האמינות של רכיב מסוים. בדרך כלל ננסה להשיג דגימה של פלט PCR לגיטימי ונשווה את התוצאה שיצאה לנו לתוצאה שאמורה להיות לגיטימית. במקרים מסוימים אפילו נשלח את תוצאת ה-PCRs שלנו לשרת שלישי שהוא יפנה ל-DB שמכיל תוצאות PCR לגיטימיות, ויחזיר לנו תשובה בהתאם - ככה לדוגמא ניתן ליישם attestation בארגון רחב.

הלוג שיוצר measured boot יכול לעזור לנו לוודא שאכן תהליך ה-boot עלה כמו שצריך, ב-windows הקובץ נמצא תחת הנתיב:

```
C:\Windows\Logs\MeasuredBoot
```

וב-linux תחת הנתיב:

```
/sys/kernel/security/tpm0/binary_bios_measurements
```

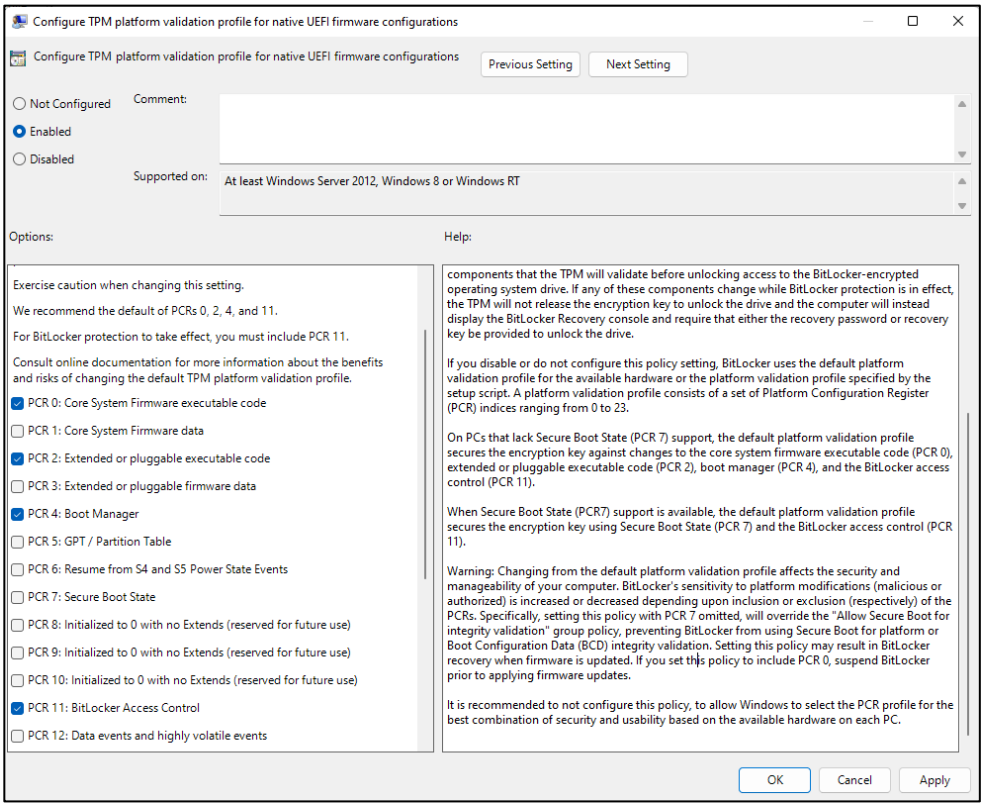
PCR Index	PCR Usage
0	SRTM, BIOS, Host Platform Extensions, Embedded Option ROMs and PI Drivers
1	Host Platform Configuration
2	UEFI driver and application Code
3	UEFI driver and application Configuration and Data
4	UEFI Boot Manager Code (usually the MBR) and Boot Attempts
5	Boot Manager Code Configuration and Data (for use by the Boot Manager Code) and GPT/Partition Table
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8-15	Defined for use by the Static OS
16	Debug
23	Application Support

[נלקח מההרצאה An Unified TPM Event Log for Linux]

BitLocker

דוגמא קלאסית לפיצ'ר אבטחתי שבנוי על רכיב ה-TPM. המטרה שלו היא להצפין את ה-hard drive, ככה שלא יקרה מצב שתהיה גישה לא מורשית ל-HD. הוא מצפין את כל ה-HD חוץ מהחלק שאחראי על עליית המערכת - ESP לדוגמא. מפתח ההצפנה ישמר ב-TPM בכדי לדאוג שה-HD לא יהיה ניתן לפענוח כאשר אינו מחובר למחשב. בעת עליית המחשב לאחר שה-TPM יאמת את כל ה-PCRs הרלוונטיים אליו הוא ישלח ל-CPU את מפתח ההצפנה.

מנגנון אבטחה שניתן להוסיף ל-bitlocker הוא pin / password / smart card. כלומר בכדי שמפתח ההצפנה ישתחרר מה-TPM המשתמש יהיה חייב להכניס את אלמנט האימות המתאים. כלומר לא רק אם ה-PCRs ב-state המתאים גם המשתמש צריך להזדהות.

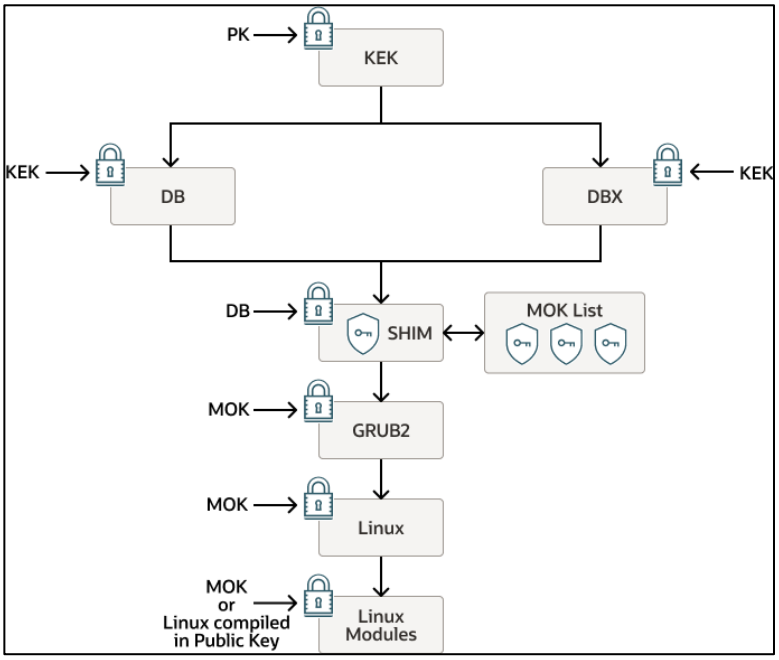


צירפתי גם תמונה שמראה כיצד באמת אפשר לקנפג את bitlocker, כלומר לפי אילו תנאים הוא משחרר את מפתח ההצפנה.

A Touch of Linux - shim

חלק זה רלוונטי בעיקר למערכות הפעלה מסוג Linux. בגלל שיש כל הרבה הפצות של Linux אז קשה מאוד לחתום את כולן קריפטוגרפית, לכן הוצע שיהיה רכיב שיהיה חתום על ידי Microsoft שיטען ראשון ואז ימשיך את תהליך ה-secure boot על שאר הרכיבים.

Shim היא תוכנית pre-bootloader שחתומה על ידי Microsoft, שמטרתה להמשיך את תהליך ה-secure boot ב-Linux. כלומר לוודא את החתימה שלהם ולטעון אותם רק אם הם מאושרים. בכדי לעשות זאת הוא משתמש ב-machine owner key (MOK), זו רשימה של מפתחות קריפטוגרפים שלפיה shim קובע אם אפשר לטעון רכיב מסוים או שלא. משתמשים יכולים להוסיף מפתחות ל-MOK בעצמם וככה עדיין להישאר ב-secure boot עם איזה bootloader או kernel שירצו.



[נלקח מתיעוד של Oracle]

Boot Attack Surface

הגענו לחלק המעניין של המאמר שבו באמת נראה כיצד ניתן לנצל לרעה את התהליך. בפרק הזה אציג מתווי תקיפה שלדעתי היו הכי מעניינים וקריטיים, נתחיל בלהכיר קצת יותר על Bootkit-ים, אז נעבור למתקפות שמשלבות גישה פיזית למחשב ולבסוף אציג שני CVEs מאוד מעניינים לדעתי שמנצלים בצורה דיי יצירתית את התהליך.

Bootkits

קבצי malware התקדמו מאוד עם השנים, והתחילו גם לנצל לרעה את תהליך ה-boot: Bootkits. לפני שהיו קיימים כל הפיצ'רים ההגנתיים שדיברנו עליהם מקודם, הרצה של malware ברמה כזו נמוכה של המערכת שלנו הייתה משימה לא כזו קשה, כי כל AV רגיל היה עולה בשלב מאוחר יותר בתהליך ועד אז תוקף יכל לבצע דיי כל מה שבראשו. עם השנים נהיה יותר ויותר קשה להשפיע על תהליך ה-boot בצורה זדונית בעיקר בגלל פיצ'ר ה-secure boot.

אבל עדיין תהליך ה-boot מהווה משטח תקיפה רחב מאוד ומלא ב-CVEs שרק מחכים שישתמשו בהם. הרבה אנשים שוכחים מהחלק הזה במערכת שלהם, בעיקר בגלל שלא נוגעים בו ביום יום. אז מאוד יכול להיות שהם לא עדכנו את ה-firmware שלהם תקופה ארוכה מאוד, וגם יכול להיות שהם בכלל משתמש ב-legacy BIOS במקום ב-UEFI. הפואנטה היא שגם עם כל מערכות ההגנה ב-bootkit-ים הם מתווה תקיפה מאוד רלוונטי שצריך לקחת אותו באקסטרה רצינות.

Bootkits בדרך כלל אוהבים לנצל לרעה את ה-ESP או אפילו ישירות את ה-NVRAM. על ה-ESP יימצאו קבצי boot מאוד רגישים שמהווים חלק קריטי מהעלייה של המערכת שלנו, וה-NVRAM מהווה גורם רגיש אפילו יותר. תוקף לדוגמא יכול להכניס DXE Driver משלו ובכך להריץ קוד זדוני בשלבי הריצה של ה-firmware עצמו.

Real Life Bootkits

יש כבר כמות יפה של bootkit-ים שכבר התגלו ונחקרו לעומק, אני לא עומד לסקור אותם במאמר הזה כי אני מרגיש כבר יצאו עליהם כל כך הרבה מחקרים שמציגים אותם הרבה יותר משאני יכול. אם תרצו לקרוא על כמה מוכרים אז בקישור פה תוכלו למצוא רשימה של כמה bootkit-ים מעניינים ולקרוא עליהם קצת. לדוגמא ישנו bootkit בשם BlackLotus שהוא ה-bootkit הראשון שהתגלה שהצליח לעקוף את secure boot, הוא הצליח באמצעות השמשה של חולשה. מפחיד לחשוב עוד כמה מסתובבים לנו בעולם ורק מחכים להתגלות.

Physical Access = Game Over

מתווה תקיפה שנשמע אפילו טיפה מצחיק הוא גישה פיזית לעמדה עצמה, בגלל שתהליך ה-boot הוא כל כך "ראשוני" במערכת ההפעלה שלנו אז הוא מסתמך המון על החומרה וגם על קנפוג שלו באמצעות ממשק ה-BIOS וכדומה.

סוג כזה של מתקפות נקרא Evil Maid Attack, על שם עוזרת מרשעת שמתעסקת לך עם המחשב בזמן שאתה לא מסתכל (סטייל mission impossible).

האפשרויות של תוקף הן אינסופיות עם כזו גישה, פחות אכנס לעומק בחלק הזה של מתקפות, אבל היה לי חושב להבהיר שניתן לעקוף כל פיצ'ר הגנתי שקיים על העמדה במקרה בו יש גישה פיזית לעמדה - הכל תלוי

בכמה אתה נחוש וכמה אתה מכיר את המטרה שלך. במשפחה זו של מתקפות נכללות מתקפות כמו TPM Sniffing, שהמטרה שלה היא להסניף את מפתח ההצפנה של bitlocker שנשלח ב-plain text מה-TPM ל-CPU. Microsoft טענו שמתקפה זו מאוד מסובכת לממש וכי היא לא הכי ריאלית, אז מישהו הצליח להוכיח להם והצליח רק תוך 43 שניות להדגים POC של המתקפה. נוסף על כך ב-firmware-ים שמשתמשים בבטריית ה-CMOS וגם יש להם סיסמא לממשק ה-BIOS, ניתן פשוט לנתק ולחבר את הסוללה בשביל לאפס את הסיסמא. אפשר להמשיך עוד ועוד על מתווי תקיפה כמו boot ל-recovery או ל-USB/CD, אבל יש מתקפות יותר מעניינות בהמשך.

[נלקח מShutter Stock]

Critical CVEs

CVEs בהקשר של firmware, הן לדעתי פי כמה וכמה יותר קריטיות, בעיקר בגלל שכמות האנשים שמעדכנים את ה-firmware שלהם מאז שקנו את המחשב שלהם היא אפסית. ה-firmware הוא רכיב דיי מוכחש במערכת שלנו בעיקר בגלל שהפעילות שלו דיי שקופה לנו.

כעת אציג לכם שתי חולשות שמאוד עניינו אותי אישית אז יצא לי לחקור עליהן לעומק. בחרתי אותן בעיקר בגלל שהן רלוונטיות לכל כך הרבה רכיבים ומספקות לתוקף הרבה השפעה על המערכת וסוג persistence שממש קשה לעלות עליו.

LogoFAIL

- כל מה שאני הולך להציג פה נלקח ממחקרים שנעשו ופורסמו על ידי חברת Binarly.

אני מניח שכולכם מכירים שבעת העלייה של המחשב שלנו יופיע לוגו של היצרן של motherboard שלנו:

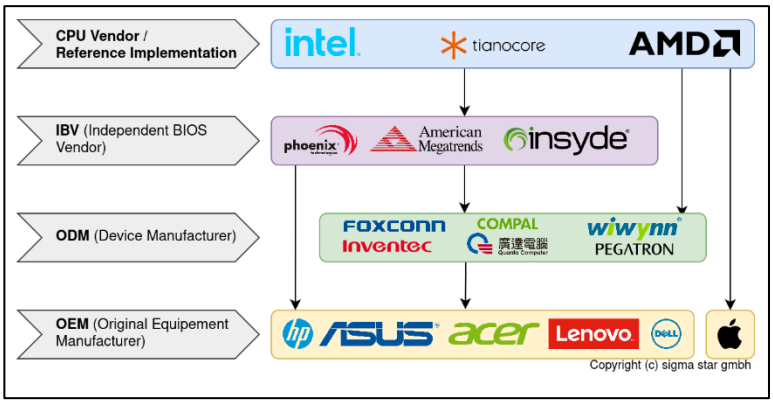
אבל אי פעם חשבתם איך בכלל מופיעה התמונה הזו על המסך שלנו? כי אנחנו עוד הרבה לפני מערכת ההפעלה. אם ניזכר בשלבי העלייה של UEFI וספציפית בשלב ה-DXE, נראה כי נטענים הרבה driver-ים שמאפשרים לנו את כל היכולות שיש ל-UEFI. אחד מהם הוא parser לתמונות bmp/jpg המאפשר להציגן על המסך.

עד לכאן הכל נשמע סבבה, אבל המצב מתחיל להסתבך ברגע שחברות התחילו לאפשר קסטומיזציה של התמונה שתופיע בתהליך ה-boot. כאן נוצר משטח תקיפה מעניין, האם זה אפשרי להכניס קבצים זדוניים ל-parser של התמונות וכך להשפיע על תהליך ה-boot?

זה בדיוק מה שחברת Binarly רצתה לחקור. היא התחילה בלבצע fuzzing (הכנסה של input לא צפוי למערכת מסוימת בכדי לבדוק את בדיקת הקלט שלה) על firmware של Lenovo ומיד הם ראו שעבור הרבה מאוד קלטים המערכת קרסה או ביצעה פעולות מוזרות. אז הם נכנסו יותר לעומק וגילו שעבור קלטים מסוימים הם הצליחו להריץ קוד בהרשאות של שלב ה-DXE.

אבל זה לא נגמר כאן, הבעיה הרבה יותר חמורה, בגלל שחברות כמו Lenovo לא כותבות את ה-firmware שלהן בעצמן, אלא נעזרות בקוד שהן מייבאות מחברות כמו phoenix או insyde.

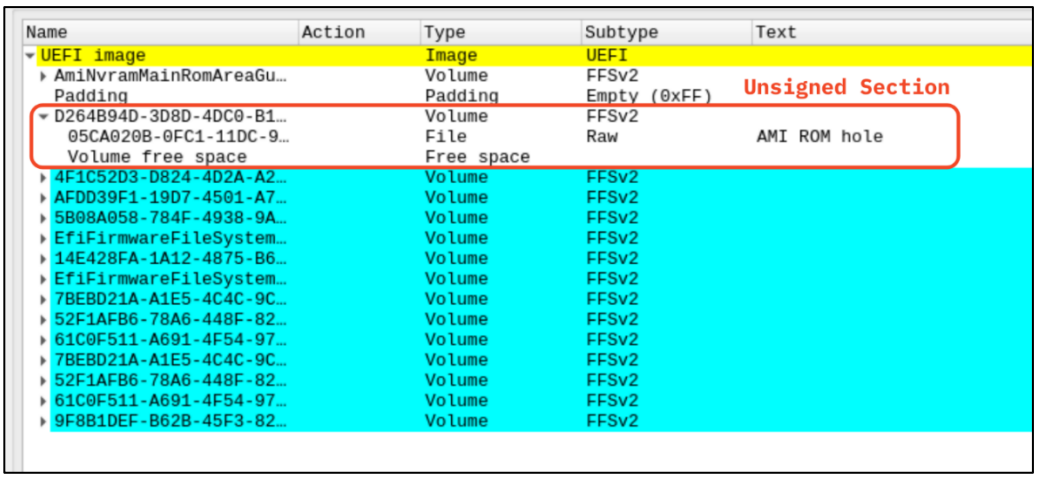
לכן ההשפעה היא הרבה יותר רחבה, כי גם אם נמצאה החולשה במחשב Lenovo אחד, היא יכולה להיות באותה המידה גם במחשבים של acer. זו הבעיה העיקרית בחולשות ב-firmware עצמן, גם היצרניות הגדולות בסופו של דבר משתמשות בקוד של חברה אחרת שמספקת שירותים לעוד עשרות חברות! בסופו של דבר binarly מצאו סה"כ 30 חולשות שונות על כמה firmware-ים שונים, המאפשרות לנו להריץ קוד זדוני בכל פעם שהתמונה נטענת למסך.



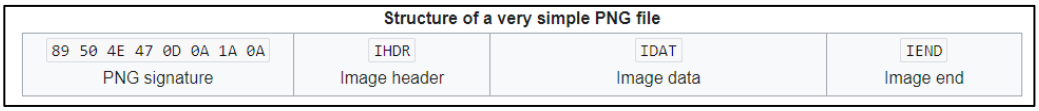
[דיאגרמה שמסבירה על חלוקת האחריות בפיתוח המחשב, נלקח מ-"Securely booting x86 using Heads - sigma star"]

כעת ניכנס לניתוח יותר עמוק של אוסף החולשות. כפי שציינתי מקודם החולשה מאפשרת לתוקף להריץ קוד בהרשאות של שלב ה-DXE, אבל משהו שלא ציינתי הוא שהמתקפה עוקפת פיצ'רים הגנתיים כמו: Secure boot & Intel boot guard. זה בגלל שהתמונה שוכנת במקום שלא חתום דיגיטלית, משום שהיצרנים מאפשרים למשתמשי הקצה להכניס תמונה משלהם.

צירפתי תמונת מסך מהכלי UEFITool שמאפר לראות איזה sections חתומים:



Binarly השתמשו באחת מהחולשות שמצאו, והשמישו אותה בכדי להדגים POC של המתקפה. השם הרשמי של החולשה הוא CVE-2023-39538. החולשה בנויה על integer overflow שבסופו של דבר מוביל ל-heap overflow שאיתו נצליח להריץ את ה-payload. בשביל להבין יותר טוב את החולשה ניכנס טיפה למבנה של קובץ PNG ואיך מפרסרים אותו. המבנה של קובץ PNG הוא כזה:



[נלקח מויקיפדיה]

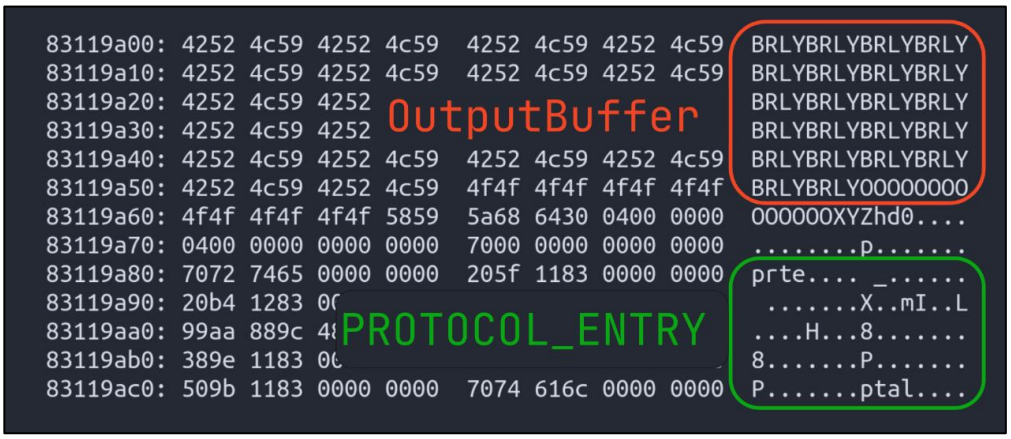
שני החלקים שהכי מעניינים אותנו הם IHDR שמכיל מידע על התמונה כגון Height and Width, ו-IDAT שמכיל את המידע עצמו של התמונה.

Binarly יצרו pseudo code של ה-DXE Driver שאחראי לפרסר את תמונת ה-PNG, אצרף את תמונה של השורות שרלוונטיות לחולשה:

```
// BRLY-LOGOFAIL-2023-016: Integer overflow
// on the argument of EfiLibAllocateZeroPool
OutputBuffer = EfiLibAllocateZeroPool(2 * PngWidth);
v7 = &OutputBuffer[PngWidth];
GlobalInfo.OutputBuffer = OutputBuffer;
```

כפי שניתן לראות בקוד נעשה allocate ל-buffer בשם OutputBuffer (שבהמשך אליו נכניס את ה-IDAT chunks), הגודל שלו נקבע לפי PngWidth שנמצא בתוך ה-IHDR. ובגלל שאנחנו שולטים בתמונה נוכל להגדיר בצורה לא לגיטימית width, נוכל להגדיל אותו ככה שהתוצאה של החישוב PngWidth * 2 תגלוש (integer overflow), כך שלבסוף OutBuffer יהיה קטן מדי מכדי להכיל את כל ה-IDAT chunks של התמונה. אז יתבצע heap overflow עם מידע שבשליטת התוקף (נוכל להחדיר לתמונה chunks לא לגיטימיים).

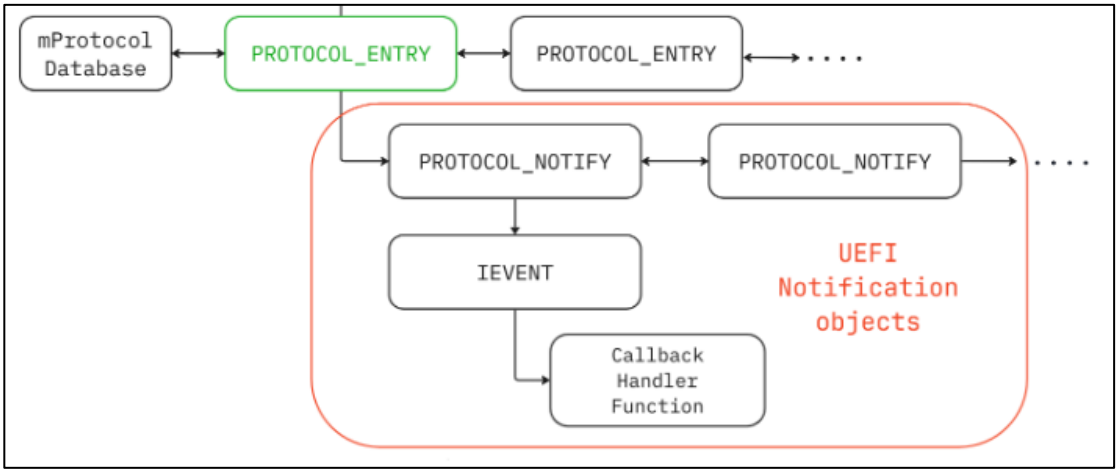
כעת עולה השאלה איזה מידע אנחנו דורסים באמצעות ה-heap overflow? לחוקרים ב-Binarly לא היו שום יכולות debugging על המכשיר, אז בכדי לראות את ה-state של ה-heap, ניסו לשמר את ה-state שלו לאחרי שמערכת ההפעלה עלתה. כלומר, גם לאחר ש-UEFI סיים את כל הפעולות שלו ומערכת ההפעלה עלתה לגמרי יהיה ניתן לחפש בזיכרון מידע שהם שמו שם. הם הצליחו לשמר את הזיכרון בעזרת שינוי של ה-header של ה-chunk ככה שהפונקציה FreePool תקרוס ולא תצליח לשחרר את הזיכרון. אז זה בדיוק מה שהם עשו, הם דרסו בעזרת heap overflow הרבה chunks שאחרי OutputBuffer והבינו מה היה באותו buffer בעזרת הרבה ניסוי וטעיה. לבסוף הגיעו למסקנה שנעשה allocate ל-OutputBuffer מיד לפני אובייקט בשם PROTOCOL_ENTRY:



Protocol הוא אלמנט קריטי כשמדובר ב-UEFI, הוא מהווה הנגשה שנוצרה בשלב ה-DXE למשאב מסוים. ניתן לחשוב על protocols ממש כמו API למשאב מסוים על המערכת שלנו.

אז האובייקט מייצג protocol מסוים וכל המידע שרלוונטי אליו, שני אובייקטים מעניינים שנמצאים בתוך PROTOCOL_ENTRY הם: PROTOCOL_INTERFACE ו-PROTOCOL_NOTIFY. שניהם מכילים pointers לפונקציות, כלומר שתוקף יכול להתעסק עם כל אחד מהם בכדי להריץ קוד משלו. ב-POC נבחר לשנות את ה-pointer של PROTOCOL_NOTIFY, שרץ בעת התקנה של אותו ה-protocol.

ה-POC ייגש למבנה ה-PROTOCOL_ENTRY וישנה בתוכו את ה-PROTOCOL_NOTIF , בו יש מבנה בשם IEVENT שמכיל את הקוד שירוץ - הקוד הזדוני שלנו:



עוד דבר שחשוב לוודא, הוא שה-protocol שאנחנו דורסים באמת יותקן לאחר הצגת התמונה אחרת כל מה שעשינו לא שווה כלום.

עכשיו נוכל להפנות את הריצה בעת ההתקנה של ה-protocol לקוד זדוני משלנו, ב-POC הקוד נשמר בתוך NVRAM variable שבגלל misconfiguration נשמר עם הרשאות ריצה במקומות הללו בזיכרון, וגם תמיד באותה הכתובת. בתוך ה-NVRAM variable יישמר shellcode שיפנה את הריצה ל-second stage payload שיימצא על הדיסק.

ה-second stage יבטל את secure boot בכך שישנה את הערך של משתנה הסביבה gSecurity2 ל-NULL ובכך יתבטל secure boot:

```
                                Set gSecurity2 to NULL ->
if (gSecurity2 != NULL) {       Secure Boot is Disabled!
  //
  // Verify File Authentication through the Security2 Architectural Protocol
  //
  SecurityStatus = gSecurity2->FileAuthentication (
                                  gSecurity2,
                                  OriginalFilePath,
                                  FHand.Source,
                                  FHand.SourceSize,
                                  BootPolicy
                                  );
```

בשביל באמת להדגים את היכולות של החולשה ניצור קובץ על הדיסק, אבל כרגע אין לנו DXE Driver שיודע לכתוב למערכת קבצים מסוג NTFS אלא רק לקרוא ממנה. לכן נחליף את ה-driver הנוכחי ב-driver משלנו וניצור קובץ על ה-desktop.

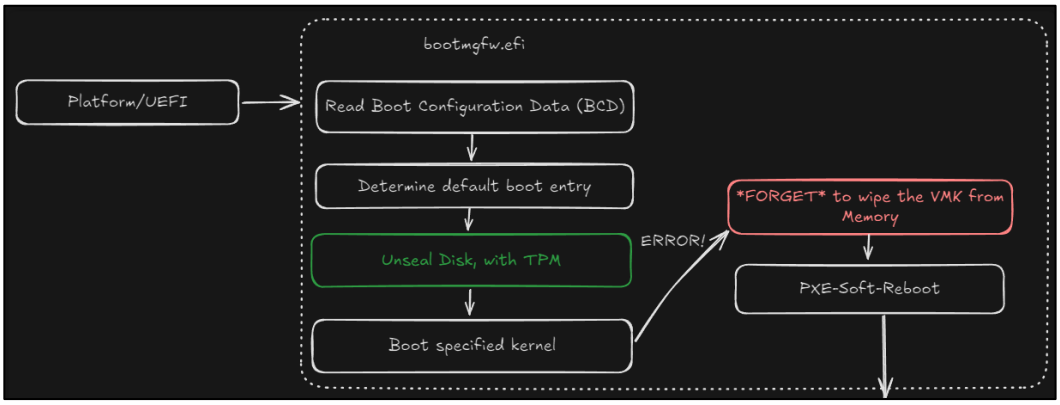
זהו! הצלחנו באמצעות תמונת PNG אחת להשתלט לגמרי על המערכת להשיג persistence שקשה מאוד לגלות. היה לי חשוב להציג את המתקפה הזו בכדי להראות כמה חזק תהליך ה-boot וגם כיצד ניתן להשתמש בדברים הכי טריביאליים בצורה זדונית. לקישור לסרטון שמראה את מהלך התקיפה לחצו פה.

BitPixie

הבעיה העיקרית שהייתה לי עם ה-POC של LogoFAIL הוא שבמקרה ש-bitlocker אכוף על העמדה כל המתווה שהצגתי עכשיו לא רלוונטי, משום שלא נוכל להשפיע על קבצים של מערכת ההפעלה בשלב ה-DXE כי הכונן עדיין מוצפן. כמובן שאפשר לשנות את ה-POC ככה שכן נצליח לעקוף את BitLocker אבל הרגיש לי שיש פתרון הרבה יותר פשוט. אז נתקלתי בחולשה בשם BitPixie, חולשה מ-2023 שמנצלת את PXE בשביל לעקוף את BitLocker. החולשה נמצאת ברכיב bootmgrfw.efi בגרסאות מסוימות של windows. אתם בטח שואלים את עצמכם למה אני בכלל מציג את החולשה הזו? בטח תיקנו אותה בעדכון תוכנה של לפני כמה שנים טובות, ואתם צודקים. במחשבים חדשים לא ניתן לנצל את החולשה, אבל מה אם ניתן להוריד את הגרסה של המחשב? אם ניזכר ב-PXE ניתן ממש להגדיר את רכיבי ה-boot של העמדה - אז למה שלא אגדיר גרסאות פגיעות?

זה פותח אותנו לעולם שלם של exploitation, ניתן לנצל כל חולשה ברכיב boot גם אם החולשה תוקנה לפני כמה שנים. נבצע downgrade בעזרת PXE, פשוט לא? ראשית נבין טיפה איך עובדת החולשה BitPixie, החולשה בנויה על איך bootmgrfw.efi מטפל בשגיאות בזמן תהליך ה-boot. בתהליך עלייה רגיל לאחר שמפתח ההצפנה של bitlocker שוחרר (כלומר כל ה-PCRs אומתו), ה-boot manager מסתכל על הגדרות ה-BCD ולפיהן קובע איזה boot loader להעלות. אבל מה יקרה אם ההגדרה לאיזה boot loader תהיה שגויה,

כלומר עם נתיב שאינו קיים - נגיע ל-recovery image שהוא יכול להיות כל דבר. אפילו עוד סביבת windows שתפענח את הדיסק באמצעות מפתח ההצפנה של bitlocker שטעון לזיכרון. לכן במקרה של error ופנייה למצב recovery יש למחוק את מפתח ההצפנה מהזיכרון. אבל בגרסאות שפגיעות- ל-BitPixie מפתח ההצפנה לא נמחק מהזיכרון, וניתן למצוא אותו במצב ה-recovery. אופציית ה-recovery שנבחר להגדיר היא PXE Soft Reboot, כלומר נטען מהרשת recovery image מבלי לבצע restart מלא על המערכת ואולי למחוק את מפתח ההצפנה.

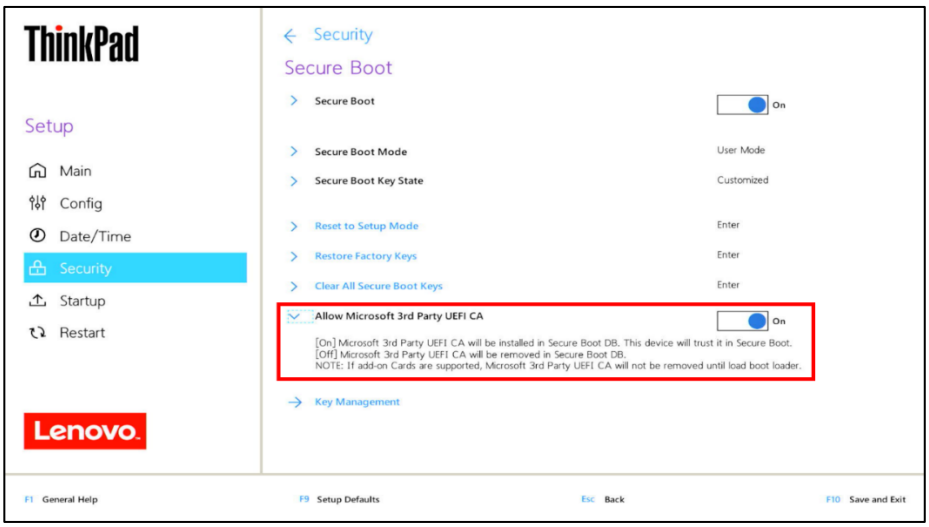


[נלקח מ- " Windows BitLocker -- Screwed without a Screwdriver "]

מכאן אנחנו יכולים לקחת את ה-exploitation לשני כיוונים, יש שני recovery images שאנחנו יכולים לבחור:

1. להעלות מערכת הפעלה מבוססת Linux.
2. להעלות מערכת הפעלה מבוססת Windows.

עולה השאלה מה נבחר? בעיקרון אין הבדל מהותי בין שתי האופציות אבל כן יש vendor-ים שחוסמים טעינה של רכיבים שלא חתומים ישירות על ידי Microsoft:



אז נבחר ב-Windows.

נעלה גרסה מאוד lightweight של WinPE- windows , ונשתמש ברכיבים לגיטימיים של windows. נשתמש בכלי WinPmem שמאחורי הקלעים משתמש ב-driver חתום - winpmem_x64.sys בשביל למצוא את המפתח הצפנה (נקרא VMK). לאחר שנמצא את מפתח ההצפנה נוכל לפענח את כונן C.

אז נסכם את מהלך ה-exploitation (מתחת לכל שלב אצרף תמונות שמציגות באמת את מה שנעשה באותו השלב):

1. ביצוע PXE ראשוני, ניתן לבצע באמצעות כניסה ל-Windows Recovery Environment (Shift+Restart).

```
#!/bin/bash

[ $# -eq 0 ] && echo "usage: $0 <interface>" && exit 1

scriptpath="$( cd -- "$(dirname "$0")" >/dev/null 2>&1 ; pwd -P )"
interface=$1

sudo ip a add 10.13.37.100/24 dev $interface

sudo dnsmasq --no-daemon \
        --interface=$interface \
        --dhcp-range=10.13.37.100,10.13.37.200,255.255.255.0,1h \
        --dhcp-boot=bootmgfw.efi \
        --enable-tftp \
        --tftp-root="$scriptpath/tftp" \
        --log-dhcp
```

2. ביצוע Downgrade בעזרת ה-PXE ל-bootmgrfw.efi, ניתן להשיג גרסה פגיעה בעזרת האתר - Winbindex.
3. להגדיר הגדרת BCD שגויה שתגרום ל-Error, נגדיר גם שבמקרה כזה ה-fallback יהיה PXE Soft Reboot.

```
bcdedit /store BCD_modded /set {default} path "\\"
```

```
bcdedit /store BCD_modded /set {%REBOOT_GUID%} pxesoftreboot yes
```

4. לבצע Boot לאותו boot manager פגיע דרך PXE Soft.
5. עלייה של WinPE, וטעינה רכיבים לגיטימיים של Microsoft: winload.efi, ntoskrnl.exe ... ניתן לקנפג עלייה של WinPE בעזרת הדוקומנטציה של Microsoft.
6. סריקה של הזיכרון במטרה לחפש את ה-VMK בעזרת הכלי WinPmem, שמשתמש ב-driver - winpmem_x64.sys.
7. שימוש ב-VMK בכדי לפענח את ה-recovery password של bitlocker.
8. שימוש בסיסמה בשביל לקרוא את כונן C.

ביססתי את מה שכתבתי פה על bitpixie על מאמר שהציג את החולשה, במאמר מצורף סרטון שמציג את כל ה-flow.

סיכום

אני מקווה שהצלחתם לקבל תפיסה טובה לגבי תהליך ה-boot של המחשב שלנו, והבנתם כמה הוא נפיץ. במידה ותוקף מצליח לנצל את התהליך לרעה אז לדעתי זה דיי game over. בעיקר בגלל העובדה שקשה לזהות כזה סוג של נוזקה, כפי שציינתי, הוא יכול לרוץ עוד הרבה לפני ה-AV/EDR.

לבסוף סיקרנו ניצול של אלמנטים שלא בהכרח ציפינו שניתן דרכם להשיג גישה למערכת - כמו החלפה של תמונת היצרן.

כמובן שיש עוד הרבה לאן לחקור בנושא אני עדיין מרגיש שאני עומד על קצה הקרחון, ושיש עוד הרבה דרכים לנצל את התהליך שרק מחכים להתגלות.

על המחבר

שמי רן, אני בן 20, Security Researcher במקצועי. זו הפעם הראשונה שיוצא להתנסות בכתיבת מאמר שכזה, מקווה שהצלחתי לחדש ולעניין :)

מקורות מידע

- Ultimate Guide: What Is GPT Disk, How to Use GPT in Windows - https://www.easeus.com/diskmanager/what-is-gpt.html
- How Computers Boot Up - https://manybutfinite.com/post/how-computers-boot-up/
- Secure Boot and Trusted Boot - Microsoft - https://learn.microsoft.com/en-us/windows/security/operating-system-security/system-security/trusted-boot
- Intel BootGuard final.pdf - https://github.com/flothrone/bootguard/blob/master/Intel%20BootGuard%20final.pdf
- Working With UEFI Secure Boot - Oracle - https://docs.oracle.com/en/operating-systems/oracle-linux/10/secure-boot/sboot-OverviewofSecureBoot.html#sb-overview
- Winbindex - https://winbindex.m417z.com/
- Bypassing BitLocker Encryption: Bitpixie PoC and WinPE Edition - https://blog.compass-security.com/2025/05/bypassing-bitlocker-encryption-bitpixie-poc-and-winpe-edition/
- Windows BitLocker -- Screwed without a Screwdriver - https://neodyme.io/en/blog/bitlocker_screwed_without_a_screwdriver/#teaser
- LogoFAIL - Binarly -
    - https://www.binarly.io/logofail
    - https://www.binarly.io/blog/the-far-reaching-consequences-of-logofail
    - https://www.binarly.io/blog/finding-logofail-the-dangers-of-image-parsing-during-system-boot
    - https://www.binarly.io/blog/inside-the-logofail-poc-from-integer-overflow-to-arbitrary-code-execution
    - https://pagabuc.me/slides/LogoFAIL_bheu23.slides.pdf
- bootkit-samples - github - https://github.com/hardenedvault/bootkit-samples