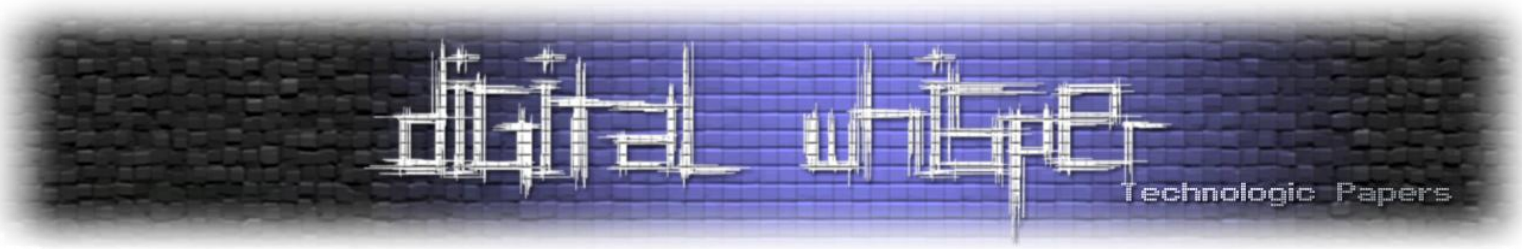


Digital Whisper

גליון 178, אוקטובר 2025

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל וספיר פדרובסקי
כתבים:	יניב קפיליאנוב, ירין הרמן, עידן שכטר, רן אברהם, יהב פסטינגר, סופיה שביקובסקי

פניות, תגובות, כתבות וכל הערה אחרת - נא לשלוח אל editor@digitalwhisper.co.il

דבר העורך

חודש סוער עבר עלינו, ובמיוחד לחובבי הז'אנר של עולמות ה-Supply Chain ואבטחת ענן, והפעם GitHub ו-Azure היו במרכז העניינים!

למקרה שלא התעדכנתם, יצא לנו לחוות החודש מספר מתקפות מעניינות ומסוכנות סביב פלטפורמת Github.

זה התחיל מבוקר אחד בו אלפי Repositories בשם s1ngularity צצו להם בן רגע ברחבי Github ובהם הרבה סודות, Token-ים, מפתחות וקטעי קוד שאמורים היו להיות פרטיים.

חוקרים מחברות אבטחה רבות החלו לשתף פרטים מניתוחים שעשו על המתקפה והרכיבו את הפאזל סביב האירוע. הסתבר שלפני מספר שבועות תוקף הצליח לנצל מיס-קונפיגורציה ב-Github Actions (מערכת האוטומציה של GitHub, שמריצה Workflows כמו בנייה ובדיקות קוד) שאפשרה לו להשיג גישה לסודות של Workflows שונים ובחלק מהמקרים אף לערוך את ה-Workflow-ים עצמם כך שיזליגו את ה-Token שבו השתמש אותו ה-Workflow. חלק מהסודות שדלפו היו מפתחות ל-AWS ו-npm-tokens לספריות שונות.

התוקפים זיהו מספר חבילות Nx (Framework פופולרי לניהול פרויקטי Monorepo ב-JavaScript/TypeScript), שהורדו על ידי אלפי משתמשים, והחלו להרעיל אותן בקוד שגונב סודות ומפתחות נוספים מהמפתחים שהורדיו והתקינו את אותן החבילות.

הספריות האלו הכילו קוד JS זדוני, שמינף את כלי ה-AI הלוקאלי (Copilot ודומיו), בכך שנתן לו prompt פשוט, המבקש לאסוף רשימה של סודות (משתני סביבה, GitHub Credentials וכו').

לאחר מכן, הקוד העלה את כל המידע שהשיג, מקודד פעמיים ב-base64, ל-Repository פומבי (תחת המשתמש שהתקין את החבילה) בשם s1ngularity. מה שאומר שמעבר לתוקף, כל העולם יכל לראות את הסודות האלו.

לפי מה שראיתי, חלק מה-repositories האלו הורדו אחרי יותר מידי זמן, ויש יותר מידי אנשים שעשו Scraping לכל ה-Repositories עם השם הזה על כל סודותיהם.

זו לא פעם ראשונה שיש חבילות npm זדוניות, אני זוכרת הרבה כאלו בעיקר למטרות crypto-mining, אך זו היתה ממש רועשת, בעיקר בגלל יצירת ה-Repositories הפומביים, המידע הרגיש שהכילו, ואיך אפשר שלא - השימוש ב-AI. בכל פעם שאחד מהסודות שהודלפו במסגרת המתקפה היה npm-token, עלתה גרסא חדשה פורסמה חבילה זדונית נוספת.

כשבועיים לאחר המתקפה הזו, ככל הנראה באמצעות אותן סודות שזלגו, התוקף הצליח להשיג npm-token בעל הרשאות לדחוף קוד לספריה בשם rxnt-authentication והרעיל גם אותה בקוד נוסף.

אחד ה-Payload-ים הזדוניים שנוספו לספריה הזו, היה שכאשר חבילה זו מותקנת, היא מריצה כלי בשם TruffleHog (כלי קוד פתוח לסריקת קוד ומאגרים לאיתור סודות), שסורק סביבות ענן AWS, GCP וכו' בחיפוש אחר Credentials. במידה והסקריפט מצא GitHub token, הוא יוצר Repository פומבי בשם Shai-Hulud, המכיל את כל המידע שנסרק.

ולמה דווקא השם הזה? אם קראתם את הספר "חולית" (או ראיתם את הסרטים), אני מאמינה שהשם "Shai-Hulud" לא זר לכם (אך לטובת השאר: מדובר בתולעת ענקית ומפלצתית מסדרת הספרים). כאשר הסקריפט נתקל ב-npm-token, הוא משתמש בו כדי "להתפשט" לחבילות נוספות - כמו תולעת. ובסיפור שלנו היא ה-Sequel למתקפת ה-s1ngularity.

התוקף, באמצעות התולעת הזאת, הצליח להשיג גישה ולהרעיל ספריות שכמות ההורדות השבועית של כולן יחד עומדת על יותר מ-2.5 מיליארד!

רשימת הספריות הנגועות פורסמו ונוקו, ולאט לאט נראה שאפשר להגיד שהאירוע מאחורינו. אך כמו הגל השני שהכיל לאחר המתקפה הראשונה - כנראה שאין לנו באמת מושג מה יוליד העתיד ומה אותו תוקף ינסה לעשות עם המידע שאותו השיג.

החודש במקרה היה גם הכנס הנפלא fwd:cloudsec, ולא יכלתי שלא לשים לב שהיו בו לא מעט הרצאות על GitHub! אז כנראה הנושא הזה לא כל-כך חדש, ואולי רק עכשיו מתרחשת מתקפה מספיק מטרידה כדי שתגיע גם אלינו: האנשים שפחות חיים מתקפות GitHub. לא משנה מה הסיבה, אין ספק שכעת נהיה הרבה יותר עירניים לעולם הזה, שיש בו סיכון עצום.

יש פה כמובן מספר דברים מעניינים, אני חושבת שאחד מהבולטים זה השימוש המשעשע ב-AI לצורך איסוף הסודות. כמובן שהיה ניתן לסרוק את כל הקבצים ולהביא את המידע הזה באמצעות קוד לא מאוד מורכב, אך השימוש במנוע ה-AI המקומי, הוא חשאי יותר ואולי אף יותר יעיל.

ואם כל זה לא הספיק, החודש פורסמה מה שכנראה נחשבת "החולשה הכי מפחידה שהיתה אי פעם ב-Azure".

החוקר Dirk-jan (אין מצב שחקרתם קצת על Azure והשם הזה זר לכם), גילה חולשה אדירה המאפשרת לכל בן אדם עם ידיעה של TenantId ומשתנה בשם netid, להשיג token לכל משתמש, בכל tenant, לא משנה איזה פוליסות מוחלות עליו.

במהלך המחקר שביצע Dirk-jan, הסתבר שיש ב-Azure אובייקט לא מתועד המכונה "Actor Token", מדובר ב-token פנימי שאיננו אמורים להיות מודעים אליו. הוא נוצר בעת ההתקנה של hybrid exchange. המטרה שלו היא לאפשר ל-exchange לתקשר עם שירותים בשם המשתמש (on behalf of) שמבקש אותם, אני מניחה שאתם כבר מבינים לאן זה הולך...

על מנת לייצר Token שכזה, התוקף נדרש לדעת מה הוא ערך ה-netid של המשתמש אליו הוא רוצה להתחזות (שזה אומנם מזהה פנימי, אך ניתן לגלות על ידי הרצת שאילתה די פשוטה מכל משתמש ב-Tenant, או כפי ש-Dirk-jan הציע, ניתן גם לנסות לעשות לו Brute force).

אגב כשאני אומרת כל משתמש, אני מתכוונת לכל משתמש, כן, בכל tenant. ויותר מכך, Dirk-jan גילה של-token הזה לא מתבצעת אכיפה של Conditional Access Policies או MFA.

בגדול, אפשר לאמר ש-Dirk-jan ניצח את עולם הענן עם החולשה הזו... מזל שהוא בצד של הטובים והחליט לדווח עליה!

זו באמת היתה שנה נפלאה עם המון גילויים חדשים ומעניינים, אני אנצל את הגליון הזה כדי לאחל לכולנו להנות מחופשת החגים, ושתהיה לנו שנה טובה, רגועה ושקטה!

וכמובן, לפני שניגש לתוכן הגליון, נרצה להגיד תודה לכל מי שישב והשקיע מזמנו וכתב לנו מאמר החודש. תודה רבה ל**יניב קפיליאנוב,** תודה רבה ל**ירין הרמן,** תודה רבה ל**עידן שכטר,** תודה רבה ל**רן אברהם,** תודה רבה ל**יהב פסטינגר,** תודה רבה ל**סופיה שביקובסקי**!

קריאה נעימה,

ספיר פדרובסקי ואפיק קסטיאל

תוכן עניינים

איך פורצים ומשתלטים על לוויין?

מאת יניב קפיליאנוב

הקדמה

דמיינו לכם קוביית מתכת, בגודל קופסת נעליים, חגה בגובה מאות קילומטרים מעל ראשכם. היא נראית תמימה, אבל בתוכה - מחשב לכל דבר. הוא מקבל פקודות, משדר נתונים, מפעיל חיישנים, מנהל משימות מדעיות, ולעיתים גם מתקשר עם מערכות קרקעיות קריטיות.

מי ירצה לפרוץ אליו? אולי מדינה זרה שתנסה לשבש ניסוי טכנולוגי מתקדם של יריבה, אולי החוקר שיחשוף מחדל בקוד פתוח ששוגר לחלל ואולי מתחרה מסחרי שיבקש לפגוע במשימת צילום או תקשורת. והאמצעים? לא טילים, אלא אנטנה, מחשב וקוד טוב. שליטה בפיקוד של לוויין משמעה לעיתים שליטה במשימה כולה: שינוי מסלול, הפסקת שידורים, גישה לנתונים גולמיים, שליחה של קושחה עוינת - או פשוט הפעלת המפרש שיסיים את המשימה.

איום תיאורטי בלבד? לא בהכרח, במקרים מסוימים, כל מה שנדרש הוא קוד באורך 32 ביט.

במסגרת המחקר חקרתי source code ו-firmware של הלוויין האקדמי PW-Sat2, לווין מחקר שנבנה באוניברסיטת ורשה לטכנולוגיה. מדובר בפרויקט אקדמי שהסתיים תפעולית בשנת 2021, לכן כל המאמר נכתב בפרספקטיבה תאורטית בלבד. PW-Sat2 שוגר והפעולה שלו הסתיימה, אך פרויקט PW-Sat ממשיך להתקיים, וגרסה עתידית, PW-Sat3, נמצאת בפיתוח. ייתכן שחלקים מקוד המקור של הגרסה השנייה (PW-Sat2) משמשים כבסיס גם בגרסה הבאה. לכן, פרסום הממצאים רלוונטי גם לעתיד הקרוב.

במחקר גיליתי כי מנגנון האימות (Authentication) לפקודות uplink מתבסס על קוד אבטחה יחיד בגודל 32 ביט, ללא הצפנה, חתימה דיגיטלית או מנגנון אימות נוסף. חישוב מתקפת Brute Force נאיבית העלה שהמהירות המוגבלת של החיבור (1200 bps) וחלון התקשורת המצומצם, הופכים את ההתקפה לבלתי אפשרית לאורך חיי המשימה של הלוויין. כמו כן, מבדיקה שטחית לא מצאתי חולשה בפרסור של פקודות uplink. עם זאת, ניתוח סטטי של קבצי הקושחה שפורסמו פומבית לפני השיגור הוביל אותי למציאת קוד האבטחה שנצרב על הלוויין, מה שמאפשר לתוקף לשלוח פקודות ללוויין ממש כמו תחנת הקרקע האמיתית. במאמר אדגים כיצד שליטה מלאה בפרוטוקול הפקודות מאפשרת ביצוע פעולות קריטיות, עד כדי העלאת קושחה זדונית למסלול (in-orbit firmware update).

במסגרת המאמר, אעסוק בעיקר בתוכנת ה-OBC (On Board Computer) של הלוויין, במבט על ה-source וה-firmware, במטרה למצוא דרכים לפרוץ ללוויין כתוקף שאין לו גישה לתחנת הקרקע, אלא עם צלחת שידור עצמאית.

גילוי נאות: מטרת המחקר היא העלאת מודעות וקידום סטנדרטים גבוהים של אבטחת מידע בעולם הלוויינות המודרני. לאחר המחקר, נעשו מספר ניסיונות ליצירת קשר דרך ערוצים שונים עם צוות PW-Sat, אך ניסיונות אלו העלו חרס.

מבוא: Curiosity in Orbit

כחוקר סייבר ומפתח Low Level, מעולם לא עסקתי ישירות בתחום הלוויינות, אך סקרנות לגבי "סייבר על לוויינים" הובילה אותי לנסות להבין כיצד נראית ארכיטקטורה של לוויין מודרני מהזווית התוכנתית ובעיקר אילו חולשות אבטחה עלולות להתקיים במערכות כאלה. במיוחד עניין אותי האם ניתן, תיאורטית, להשיג שליטה במערכת לוויינית **מבלי** להשתלט על תחנת הקרקע (מבצע סייבר קלאסי).

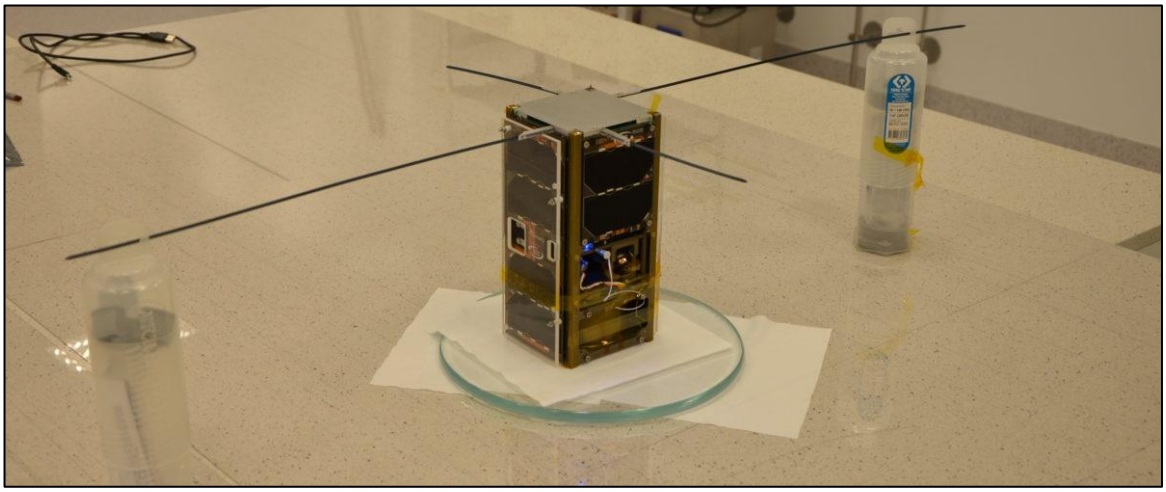
חיפשתי ב-github פרויקטי לוויינים Open source-ים, ומצאתי שישנם מעטים ולרוב הם יהיו לוויני מחקר קטנים הנקראים CubeSat הטסים במסלול LEO (Low Earth Orbit). הלוויינים החדשים, משתמשים בחומרות חדשות יחסית לתחום הלוויינות, כגון מעבדי ARM מודרניים. הדבר מאפשר לייצר מערכת תוכנתית מורכבת יותר ובעקבות זאת סביר שיהיו משטחים רבים יותר לחולשות.

בחיפושים נתקלתי בפרויקט קוד פתוח יוצא דופן - PW-Sat2. הלוויין הוא CubeSat (הסבר מפורט על סוג הלווין יצורף בהמשך) שנבנה והופעל על ידי סטודנטים מהפוליטכניקה של ורשה. מאגר הקוד לא כלל רק קוד מקור מפורט אלא גם גרסאות בינאריות מקומפלות של ה-firmware ששוחררו, הזדמנות נדירה לעיין בקוד לוויין אמיתי ומלא, זאת בניגוד לרוב המשימות המסחריות או הממשלתיות שאינן פומביות.

את המחקר כיוונתי לעבר מנגנוני התקשורת של PW-Sat2 והפרוטוקולים שמאפשרים uplink של פקודות (uplink הוא חיבור בכיוון אחד מהקרקע אל הלוויין, תחנת הקרקע משדרת והלוויין קולט) ובפרט במנגנון האימות של פקודות uplink.

רקע: CubeSats, PW-Sat2 ומודל תקשורת

לווייני CubeSat הם לוויינים זעירים בפורמט מודולרי המבוסס על קוביות בגודל 10×10×10 ס"מ הנקראות "יחידות". לוויינים אלו פותחו במקור למטרות חינוכיות ואקדמיות, אך הפכו לפלטפורמה פופולרית גם לפרויקטי מחקר, ניסויים טכנולוגיים ואפילו יוזמות מסחריות. בשל עלותם הנמוכה יחסית (מאות אלפי דולרים לפרויקט), הם זמינים גם לצוותים אקדמיים קטנים ומאפשרים ביצוע ניסויים בחלל במסלול לווייני נמוך (LEO). פרויקט PW-Sat2 תוקצב ב-180,000 יורו, כך שפגיעה בלוויין ופעילותו תהווה נזק כלכלי משמעותי לאוניברסיטה.



[תמונה מתוך האתר הרשמי של הפרויקט]

לווייני CubeSat הם לעיתים קרובות מערכות Embedded עם רכיב פלאש שעליו נצרב ה-firmware, מעבד מרכזי, זיכרונות, רכיבי תקשורת, מקורות מתח וחיישנים. המגבלות המובנות בפרויקטים מסוג זה, הן מגבלות חומרתיות ותפעוליות כגון זיכרון מוגבל, כוח חישוב מוגבל, רוחב פס וחלון תקשורת קצר. מגבלות אלו גורמות למפתחי המערכות לבחור בפתרונות אבטחה מנוונים ביחס לפתרונות מודרניים כגון אימות קריפטוגרפי לפקודות, secure boot והפרדת תחומי הרשאות. כמו כן, מפתחי לוויינים לרוב יהיו עם פחות מודעות אבטחתית מאשר שחקני סייבר שעלולים לאיים על המשימות, מה שתורם גם כן לפריצות בתחום.

בכל לוויין קיימים שני ערוצי תקשורת עיקריים: Uplink - פקודות מהקרקע ללוויין ו-downlink - טלמטריה ונתונים מהלוויין לקרקע. לרוב, ה-uplink איטי יותר לצורך אמינות וצריכת חשמל נמוכה, בעוד ה-downlink מהיר יותר. ערוץ ה-uplink נחשב לנקודת שליטה קריטית, ולכן הוא יעד מועדף לתקיפות סייבר. PW-Sat2 פעל במסלול LEO בין ה-3 בדצמבר 2018 ל-23 בפברואר 2021, סה"כ 813 ימים במסלול. ברמת התקשורת, הלוויין השתמש ב-uplink של 1200 bps (VHF), ו-downlink בקצבים גבוהים יותר.

איך נראית ארכיטקטורה של לוויין?

ארכיטקטורת מערכות לוויין נחלקת באופן מסורתי לשלושה רכיבים עיקריים: מקטע החלל (Space Segment), מקטע הקרקע (Ground Segment) ומקטע המשתמש (User Segment). מקטע החלל כולל את הלוויין עצמו, ובמקרים מסוימים גם קישורים בין לווייניים (ISL - Inter-Satellite Links) המאפשרים תקשורת ישירה בין לווינים. מקטע הקרקע אחראי לניהול ובקרה של הלוויין באמצעות טלפקודות (TC או TeleCommand) וטלמטריה (TM או Telemetry) המועברות בפרוטוקולי תקשורת ייעודיים (Space Protocols), ומבצע גם קליטה ועיבוד של הנתונים המתקבלים. מקטע המשתמש מתייחס למערכות קצה כגון, מחשבים, טלפונים חכמים או מערכות ניווט - הצורכות את השירותים שהלוויין מספק (כגון תקשורת, מיקום או תצפית).

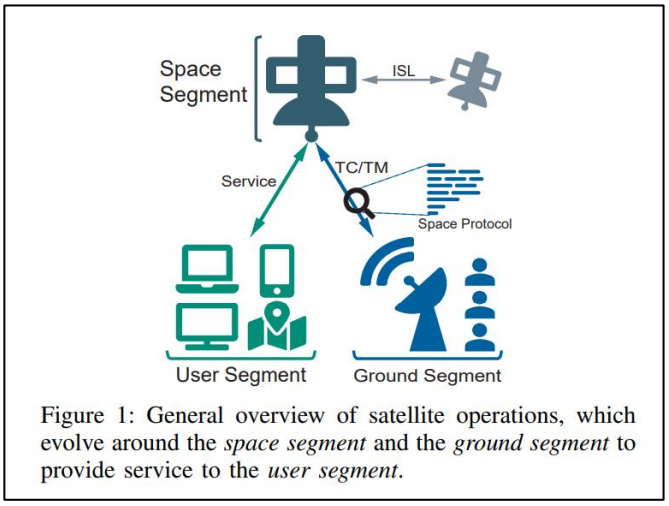


Figure 1: General overview of satellite operations, which evolve around the *space segment* and the *ground segment* to provide service to the *user segment*.

[תרשים מתוך המאמר Space Odyssey: An Experimental Software Security Analysis of Satellites]

לוויין מודרני מורכב משני חלקים עיקריים: ה-Bus (המוצג באפור כהה) וה-Payload (באפור בהיר), כפי שמוצג בתמונה מטה. ה-Bus כולל את תתי המערכות החיוניות לתפעול הלוויין: מערכת בקרת הכיוון והיציבות (ADCS - Attitude Determination and Control System) האחראית לשמירה על זווית וכיוון מדויקים בחלל; מערכת אספקת הכוח (EPS - Electrical Power System) המייצרת ומנהלת אנרגיה חשמלית באמצעות פאנלים סולאריים וסוללות; **מערכת בקרת הנתונים והפיקוד (CDHS - Command and Data Handling System) המהווה את מרכז הבקרה והמחשוב של הלוויין; ומערכת התקשורת (COM - Communications) המאפשרת העברת פקודות ונתונים בין הלוויין למקטע הקרקע**. ה-Payload מהווה את הרכיב הייעודי של הלוויין בהתאם למשימתו. לדוגמה, מצלמות תצפית, חיישנים מדעיים או משדרי תקשורת. השילוב בין ה-Bus ל-payload יוצר מערכת שלמה שמסוגלת לבצע את משימתה תוך שמירה על תפקוד רציף ואמין לאורך זמן בחלל.

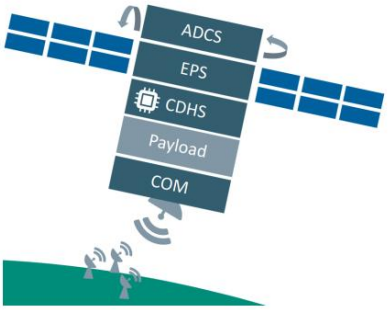


Figure 2: Overview of the different components typically found in a common satellite that features default components for a satellite payload (light grey) and bus (dark grey).

[תרשים מתוך המאמר Space Odyssey: An Experimental Software Security Analysis of Satellites]

בהסתמך על הארכיטקטורה שהוצגה לעיל, מוקד העניין במחקר זה הוא תת המערכת CDHS (Command and Data Handling System) ובפרט הרכיב במערכת שאחראי על עיבוד מסגרות הנתונים (Frames) המתקבלות ממקלט התקשורת (ה-Receiver). המקלט עצמו ממוקם בתת המערכת COM, האחראית על קליטת האותות מהקרקע, המרתם לנתונים דיגיטליים והעברתם ל-CDHS. בשלב זה, ה-CDHS מבצע Parsing לפריימים, בודק את תקינותם, מפענח את הפקודות המוכלות בהם, ומתרגם אותן לפעולות ממשיות המבוצעות על ידי שאר תתי-המערכות בלוויין.

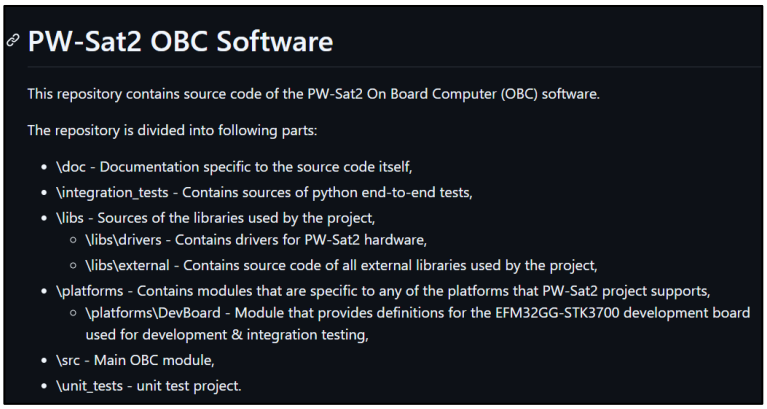
רכיב זה מהווה את "המוח" של הלוויין, ובשל שליטתו הישירה על כלל המערכות, הוא מהווה נקודת מפתח עבור המחקר.

תחילת עבודה

תחילה, הורדתי את ה-Source המלא של PW-Sat2 מ-GitHub, הכתוב ב-C++ והתחלתי לעיין בו.

מתוך כלל הקבצים, בחרתי להתמקד בחלקים שמטפלים בפרוטוקול התקשורת ובמנגנון הטלפקודות (telecommands) במטרה להבין כיצד מתבצע האימות מול הפקודות הנשלחות מהקרקע.

מהי היררכיית ה-repository?



החלקים המעניינים הם:

- ה-src, שם יושב המודול המרכזי של OBC (On Board Computer) - לוגיקת אתחול המערכת.
- Libs\drivers - מכיל את ה"דרייברים" בשביל החומרות השונות, כלומר הקוד שיודע לקבל פקודה מהחומרה הרלוונטית נמצא פה.

ארכיטקטורת קליטת המסגרת (Frame Reception)

מצאתי את המחלקה שאחראית על התקשורת שיושבת בנתיב libs\drivers\comm.

```
class CommObject final : public ITransmitter,       //
                         public IBeaconController, //
                         public ICommTelemetryProvider,
                         public ICommHardwareObserver
```

הלוגיקה לקליטת פריימים נמצאת בפונקציה CommObject::ReceiveFrameInternal, שמבצעת תהליך דו-שלבי: תחילה קריאת אורך (2 בתים) שמייצגים את גודל ה-frame, ולאחר מכן משיכת הפריים המלא. לאחר מכן האובייקט Frame מעובד וכולל מטא-דאטה כמו RSSI ודופלר (מושגים מתחום תקשורת RF), המטא-דאטה נוצר ב-communication hardware על גבי הלוויין ומהווה header לפני הפריים עצמו.

```
bool CommObject::ReceiveFrameInternal(gsl::span<std::uint8_t> buffer, Frame& frame, AggregatedErrorCounter& resultAggregator)
{
    if (buffer.size() < 2)
    {
        return false;
    }

    bool status = this->SendCommandWithResponse(Address::Receiver, num(ReceiverCommand::GetFrame), buffer.subspan(0, 2), resultAggrega
    if (!status)
    {
        return status;
    }

    Reader reader(buffer.subspan(0, 2));
    auto size = reader.ReadWordLE();
    buffer = ReceiveSpan(size, buffer);
    status = this->SendCommandWithResponse(Address::Receiver, num(ReceiverCommand::GetFrame), buffer, resultAggregator);
    if (!status)
    {
        return status;
    }
```

מנקודת מבט של ארכיטקטורת הלוויין, החיבור בין 2 המערכות, ה-CDHS וה-COM טמונה בקטע הקוד הזה, הקורא לפונקציה SendCommandWithResponse עם הפרמטר Address::Receiver שמאחורי הקלעים שולחת את הפקודה ReceiverCommand::GetFrame לכתובת חומרה של ה-Receiver דרך ממשק I^2C, ממתינה, ואז קוראת את תגובת החומרה ל-Buffer.

פענוח ואימות (UplinkProtocol::Decode)

לאחר המטא-דאטה שציינתי, המהווה את החלק הראשון של ה-Frame, יושב הפריים הפנימי, זה blob דאטה כפי שנשלח מהקרקע:

```
DecodeTelecommandResult UplinkProtocol::Decode(span<const uint8_t> frame)
{
    Reader r(frame);

    auto code = r.ReadDoubleWordBE();
    auto command = r.ReadByte();
    auto parameters = r.ReadToEnd();
```

המבנה פשוט, ונראה כך:

- [4 בתים] - קוד אבטחה (32 ביטים)
- [1 בית] - מזהה פקודה (command id)
- [N בתים] - פרמטרים

האימות על הפקודה שהתקבלה מתבצע בהמשך הפונקציה UplinkProtocol::Decode, בהשוואה פשוטה מול שדה באובייקט Hardcoded שנקרא securityCode_:

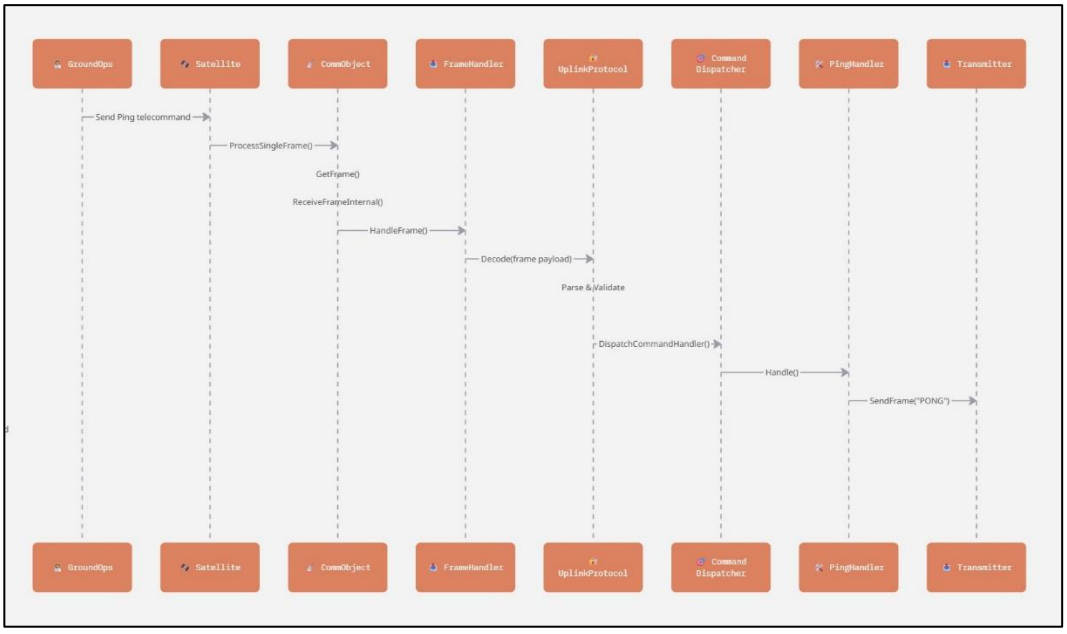
```
if (code != this->_securityCode)
{
    return DecodeTelecommandResult::Failure(DecodeTelecommandFailureReason::InvalidSecurityCode);
}
```

זה מעניין! האימות היחיד שיש על פקודה שנשלחת מהקרקע הוא השוואה פשוטה מול קוד בן 4 בתים.

Flow מלא מקבלת ה-frame בלוויין ועד הרצת הפקודה

1. משיכת ה-frame מה-communication h/w.
2. אימות הקוד שיש ב-frame מול ה-security code הקבוע מראש.
3. הרצת פונקציית ה-Handle המתאימה לפקודה לפי ה-command id ב-frame.

דיאגרמת flow מלאה של פרסור Frame וביצוע פקודה:



מודל האיומים: מי התוקף ומה הן יכולותיו?

לפני שנצלול למתקפות אפשריות, חשוב להגדיר מודל איומים שמתאים ללוויין:

- **יריב "חובבן" עם ציוד רדיו**:
  - תחנת קרקע חובבנית, אנטנות מתאימות, מודולציות ידועות, גישה לזמני מעבר (TLEs).
  - היריב אינו יודע מה קורה ברמת פרוטוקול uplink, הוא יכול להאזין ל-downlink ולקלוט שידורים לא מוצפנים.
- **חוקר אבטחה/אקדמיה**:
  - גישה מלאה לקוד המקור/בינארים שפורסמו, ידע ברוורסינג (IDA/Ghidra/radare2).
  - היריב עם הידע המתאים יכול לשדר frame-ים מתאימים ב-uplink ולהשמיש חולשות.
- **מדינה/ארגון עם משאבי RF**:
  - יכול לייצר רצף תקשורת ארוך יותר מהתחנה הרשמית בעזרת הצבת תחנות רבות גלובלית.
  - היריב בעל כלל יכולות הRF, ידע פנים על פרוטוקולים והשמשת חולשות ברמה גבוהה.

אבטחה באמצעות קוד "סודי" באורך 32 ביטים

מי קובע את הקוד?

חיפשתי בקוד איפה מוגדר הקבוע של ה-securityCode ומצאתי את הדבר הבא:

```
namespace settings
{
    constexpr std::uint32_t CommSecurityCode = ${COMM_SECURITY_CODE};
}
```

בקובץ ה-cmake של הפרויקט יש ברירת מחדל לקוד שנכנס כפרמטר (Variable Substitution), אך ניתן להעביר כפרמטר קוד אחר, ככל הנראה מה שקורה כאשר מורידים גרסה:

```
platforms > mcu > FlightModel > settings.cmake
1   set(COMM_SECURITY_CODE "0xBBCCDDEE" CACHE STRING "32-bit COMM security code written in hex with leading 0x")
```

גילינו בעיה קריטית, ההשוואה הפשוטה הזו של הקוד שנשלח ב-frame אל מול הקוד שנקבע מראש פעם אחת לפני שיגור, היא שומר הסף היחיד שמונע מתוקף לכוון צלחת לוויין על הקרקע לכיוון הלוויין ולשלוח אליו פקודות. המסקנה המתבקשת היא לנסות לעשות Brute Force על הקוד עד שנגיע לאחד הנכון ונקבל חזרה טלמטריה שמאשרת את הפקודה.

קיימת בעיה אחת - אין לי צלחת כזאת. מה שאני יכול לעשות, זה לחשב ולהבין האם המתקפה ברת יישום. החישוב נדרש מפני שבמקרה הנ"ל, בשל הקצבים הנמוכים, לא מובן מאליו שהמתקפה אפשרית, זאת בשונה משני מחשבים מחוברים בכבל, או ברשת האינטרנט על הקרקע.

Brute Force: חישוב ישימות תחת מגבלות קצב שידור

פרמטרים:

- קצב Uplink: 1200 bps
- גודל פריים (פינג מינימלי): 5 בתים (32 bit code + 1 byte cmd)
- סה"כ קשר יומי (לוויין מעל תחנת קרקע אחת): 7 דקות = 420 שניות

קצב ניסיונות משוער:

$$\text{frames/sec} = \frac{1200 \text{ bps}}{5 \text{ bytes} \times 8 \text{ bits}} = 30 \text{ שנייה/ניסיונות}$$

- **ליום**: 420 × 30 = **12,600** נסיונות

זמן לפגיעה בערך יעד רנדומלי (לדוגמה 1.18×10^9 ≈ 0x46707057)

$$\frac{1,180,000,000}{12,600} \approx 93,650 \text{ ימים} \approx 256 \text{ שנים}$$

פרספקטיבה של חיי המשימה

משך המשימה הכולל: **813 ימים** מספר ניסיונות כולל פוטנציאלי: **10,243,800** = 813 × 12,600
וזה כ-**0.23%** בלבד מהמרחב (2^32)

Brute-Force "נאיבי" בקצב השידור הנתון (1200 bps) לא היה מגיע לערך היעד אפילו אם היינו מנסים לאורך כל חיי הלוויין.

אבל - אם אני תוקף מעצמתי

כל החישוב עד כה היה בהנחה שיש תחנת קרקע אחת ממנה תוקפים, לצורך העניין, "התחנה" יכולה להיות צלחת אחת בגג הבית בו אתם גרים. בואו נניח תרחיש אחר, התוקף הוא גוף עם משאבים כמו של מעצמה ויש לו 30 תחנות פרוסות במקומות שונים על כדור הארץ, במקום חלון שידור של 7 דקות, היה לו חלון שידור של כשעה וחצי (90 דקות),

במקום **12,600** ניסיונות ביום, היינו מקבלים **162,000** ניסיונות ביום, שמתרגמות ל:

$$\frac{1,180,000,000}{162,000} \approx 7,283 \text{ ימים} \approx 20 \text{ שנה}$$

התוצאות עדיין אינן מרשימות.

ברור לכל הדעות שלפי החישוב המתקפה איננה ברת יישום, ברצוני לבדוק האם ישנו תרחיש תאורטי בו המתקפה רלוונטית. נניח ומדובר בלוויין מעט יותר חזק מבחינת קצבי שידור, כזה שבאמת מעניין מעצמות, קצב השידור בו היה לצורך העניין 20 kbps. היינו מקבלים פי 17 יותר ניסיונות בשנייה ובתרחיש כזה נקבל תוצאה של 426 ימים ≈ 1 שנה ו- 2 חודשים.

בתרחיש זה אנו היינו מספיקים להגיע לקוד הנכון בקבועי הזמנים של המשימה, לקבל Pong חזרה מהלוויין ולהתחיל להשתלט עליו. זהו תרחיש תאורטי לחלוטין, כמו כן, שמתאים רק למעצמות ולווייניים שעובדים בקצבים גבוהים. לצערי, אין לי יכולות מעצמתיות ולכן אמשיך לפתרונות אחרים.

ניסיון למצוא חולשה ב-frame parsing

לאור ממצאי האימות החלש והעובדה שמתקפת Brute Force לא ברת יישום, ביקשתי לבדוק האם קיימת אפשרות עוקפת - למשל **חולשת buffer overflow** שתאפשר כתיבה מעבר לגבולות הזיכרון **לפני** שמתרחשת בדיקת הקוד, ובכך לעקוף את מנגנון ההשוואה.

החשד נבדק בנקודה בה מתקבלת מסגרת uplink מן החומרה, דרך הפונקציה ProcessSingleFrame. תחילה בחנתי כיצד מוגדר ה-buffer שאליו נטענת המסגרת. ההגדרה הרלוונטית:

```
std::uint8_t buffer[PrefferedBufferSize];
```

כאשר PrefferedBufferSize מוגדר כ-**255 בתים**. מטרתי הייתה לבדוק מה מתרחש במקרה שבו נשלחת מסגרת ארוכה יותר - כלומר, חריגה פוטנציאלית מגבולות המערך.

הנתיב שעובר ה-frame מתחיל בקריאה לפונקציה ReceiveFrameInternal, אשר בתוכה מתבצעת קריאה לגודל המסגרת באמצעות ReadWordLE, ולאחר מכן, טעינת המסגרת אל תוך ה-buffer. הקטע הרלוונטי בפונקציה ReceiveFrameInternal:

```
Reader reader(buffer.subspan(0, 2));
auto size = reader.ReadWordLE();
buffer = ReceiveSpan(size, buffer);
status = this->SendCommandWithResponse(Address::Receiver, num(ReceiverCommand::GetFrame), buffer, resultAggregator);
```

במילים פשוטות, הקריאה לפונקציה ReceiveSpan מוודאת כי אין ניסיון לקרוא מעבר לגודל ה-span שהוגדר:

```
static gsl::span<std::uint8_t> ReceiveSpan(std::uint16_t frameSize, gsl::span<std::uint8_t> buffer)
{
    if (buffer.size() <= frameSize + 6)
    {
        return buffer;
    }
    else
    {
        return buffer.subspan(0, frameSize + 6);
    }
}
```

כאן ניתן לראות שהתנאי שומר על כך שלא ייקרא מעבר לגודל המוגדר של buffer. למעשה, אם נשלח גודל מסגרת חריג, הפונקציה תחזיר פשוט את ה-buffer המקורי (עד 255 בתים), ולא תנסה לגשת מעבר לגבול. כלומר, אין Overflow ישיר ברמת קריאת המסגרת.

המסקנה - מנגנון קריאת המסגרת כולל הגנה בסיסית אך אפקטיבית מפני overflow - ואין ניסיון לקרוא מעבר לגבולות ה-span שהוגדר מראש .עם זאת, ראוי לציין כי שכבות נמוכות יותר המטפלות בחומרה, למשל תעבורת I2C, עשויות לכלול משטחי תקיפה נוספים. במחלקות כמו I2CLowLevelBus קיימת גישה ישירה יותר לרכיבי חומרה, שייתכן וכוללת נתיבים פחות מוגנים. בשלב זה בחרתי שלא להעמיק בניתוח שכבות אלו, אך זהו כיוון מעניין למחקר אחר.

הרגע שבו Brute Force וחולשות מפסיקים להיות רלוונטיים

כיווני המחקר עד כה לא הניבו פרי לכן חזרתי למקורות, לפרויקט ה-github באתר, וכך נתקלתי בעמוד ה-Releases. העמוד מכיל את הגרסאות של ה-firmware אשר נצרבו על הלוויין ורצו בחלל. לגבי אחת הגרסאות נכתב בפירוש:

The binary was uploaded as an in-orbit update to the PW-Sat2 On-board Computer on - March 17th, 2019, allowing for switching between Deep Sleep and full flight software.

הנ"ל לא משאיר מקום לספק, אם אני אמצא ב-Released Firmware את קוד האבטחה שעוצר אותי מלהריץ פקודות, זה יהיה האחד שנמצא בחלל ונוכל להשתמש בו.

הורדתי את ה-zip ומצאתי בתיקיית build\FlightModel את ה-firmware בשם pwsat. החלטתי לזרוק את הקובץ לתוך IDA ולהתחיל לרוורס. הדבר הראשון שרציתי למצוא זה את ה-securityCode_ שצרוב על הלוויין. למזלנו הfirmware קומפל עם סימבולים (DWARF debug information) מה שמאוד מקל על העבודה. כיוונתי את עצמי לאזורים שבהם מתבצע אתחול של אובייקטים וציפיתי למצוא ערך סטטי של securityCode_ שטוענים לתוך רגיסטר. לאחר חיפוש קצר מצאתי:

```
MOVW    R6, #(Main.Communication.UplinkProtocolDecoder._securityCode - 0x880182D8)
MOVW    R5, #(Main.Communication.SupportedTelecommands - 0x880182D8)
MOVW    R4, #(Main.Communication.SupportedTelecommands._telecommands.gap0+4 - 0x880182D8)
MOVW    R0, #(Main.Communication.SupportedTelecommands._telecommands.gap0+8 - 0x880182D8)
LDR     R1, =Main.adcs.coordinator
MOVW    R3, #(Main.Communication.SupportedTelecommands._telecommands.gap0+0x60 - 0x880182D8)
STR.W   R1, [R10,LR]
LDR.W   LR, =off_B3054
MOVW    R8, #(Main.Communication.SupportedTelecommands._telecommands.gap0+0xC - 0x880182D8)
STR.W   LR, [R10,R7]
LDR     R7, =0x46707057
MOVW    R2, #(Main.Communication.SupportedTelecommands._telecommands.gap0+0x44 - 0x880182D8)
STR.W   R7, [R10,R6]
```

אפשר לראות כאן בברור ש- securityCode_ מאותחל עם הערך 0x46707057 – מצאנו Secret Leak.

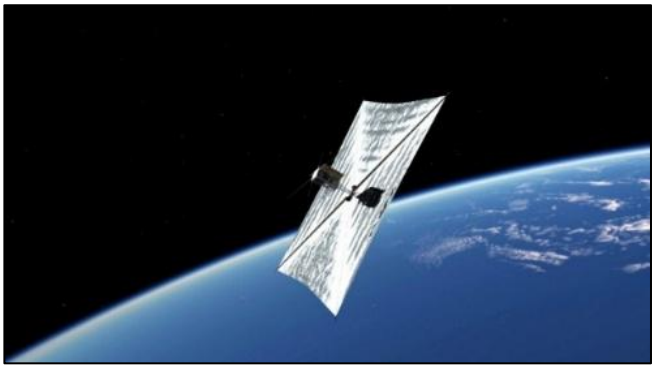
עכשיו תאורטית, אם אכוון צלחת מהקרקע לכיוון הלוויין ואבנה telecommand מתאים יחד עם קוד האבטחה הזה, הלוויין יפרסר את ה-frame ויריץ את הפקודה ללא שום מגבלות!

מה אפשר לעשות עם שליטה מלאה?

רשימה חלקית של Telecommands קריטיים שקיימים לפי קוד המקור:

- OpenSail - פריסת המפרש הפיזי (אירוע בלתי הפיך שמתחיל את סיום המשימה)
- EraseBootTableEntry / WriteProgramPart / FinalizeProgramEntry - כתיבה/מחיקה של קושחה
- SetTime / SetBitrate / SetBootSlots - שינוי פרמטרים מערכתיים
- PowerCycle / ResetTransmitter - שיבוש תקשורת
- RawI2CTelecommand- גישה ישירה להתקני I2C
- ReadMemoryTelecommand - קריאת זיכרון (כולל מפתחות/סודות נוספים)

ככה נראית פריסת מפרש, שמתחיל תהליך יציאה מ-Orbit - כן כן, סיום המשימה ואין דרך חזרה:



[תמונה מתוך האתר הרשמי של הפרויקט]

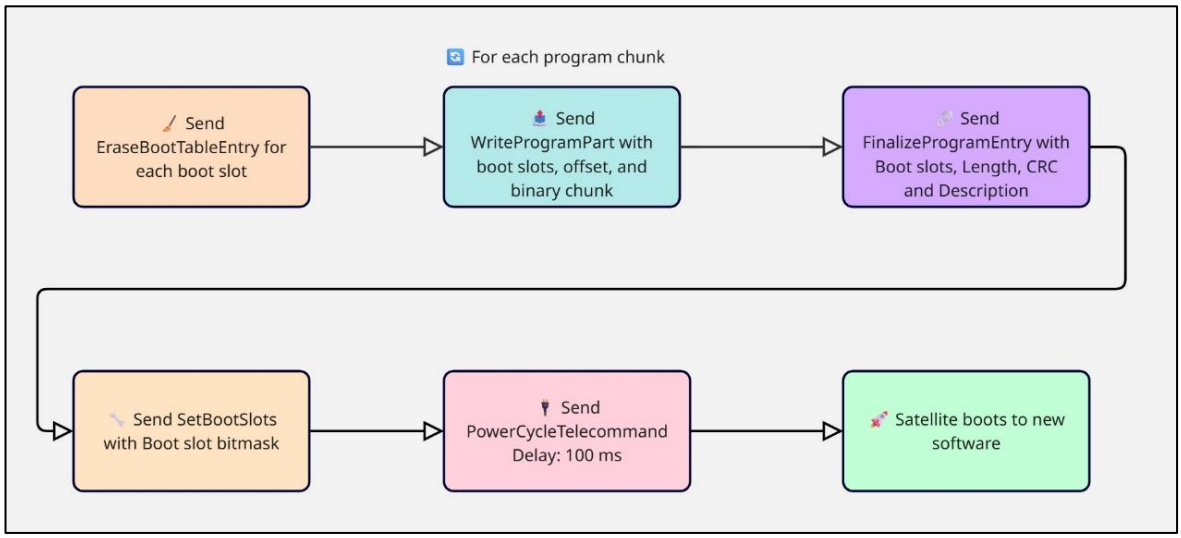
עדכון firmware בחלל: כשפיצ'ר לגיטימי הוא וקטור תקיפה

עכשיו כשיש לנו את הקוד הסודי, ניתן להריץ קוד משלנו, תאורטית כמובן.

ב **17-במרץ 2019** הועלה firmware חדש לחיסכון באנרגיה (OBC_DS_V1.0). בעדכון נוסף Telecommand שנקרא Deep Sleep שאפשרה חיסכון באנרגיה.

העדכון התבצע באמצעות רצף ה-TC הבאים:

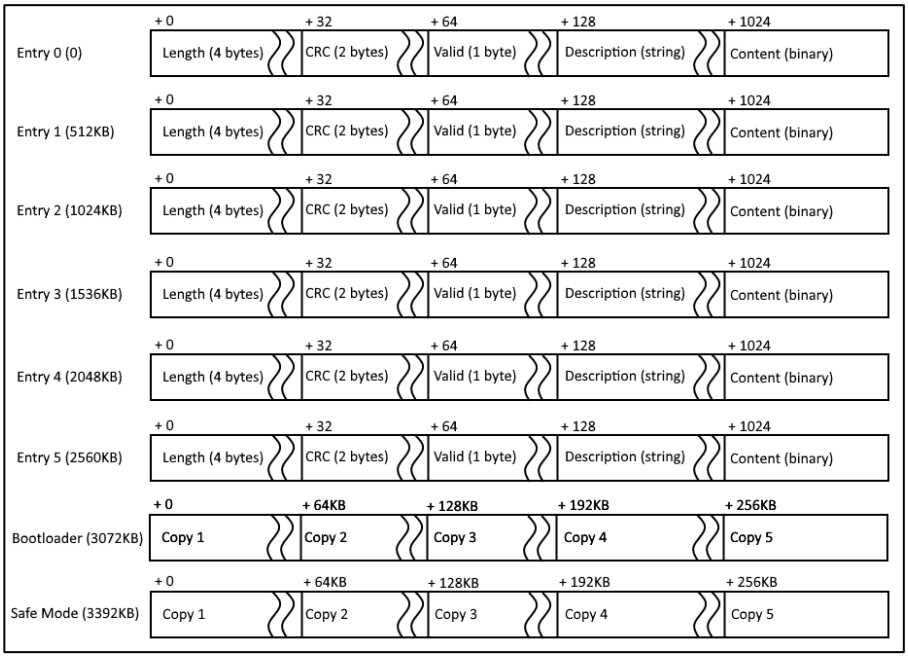
- WriteProgramPart - כתיבה של חלקי firmware
- FinalizeProgramEntry - איחוד לוגי של החלקים
- SetBootSlotsTelecommand - עדכון של ה-BootSlots בהתאם על מנת שלאחר restart יעלה ה-firmware החדש



היכולת להחליף בינארי מרחוק היא קריטית לתפעול כמו במקרה הזה, אך מייצרת פגיעות בהעדר מנגנוני אימות. תוקף יכול:

- להעלות קושחה הרסנית (Brick)
- לשתול backdoor שידליף מידע
- להנדס false telemetry ובכך לשבש מידע שחוזר למרכז השו"ב (שליטה ובקרה)

תיעוד מתוך הפרויקט לגבי מבנה ה-flash ואיך ה-firmware-ים השונים מאוחסנים, ניתן לראות כאן את ה-slot-ים שציינתי קודם:



[מתוך ה-github של הפרויקט]

לגבי בנייה של תוכנה מתאימה שנוכל לצרוב - אין שום בעיה, יש לנו את כל הקוד ואת ה-toolchain, יכולנו לעשות כמה שינויים קלים בפרויקט, לקמפל ולשלוח!

גם במקרה הזה, אין מנגנון אימות על ה-firmware, בטח שלא secure boot chain. הווידוא היחיד הוא CRC על ה-payload שאנחנו מחשבים מראש ושולחים, והוא בתורו מאומת מול חישוב CRC נוסף על גבי הלוויין עצמו. המנגנון לא נועד להגן מפני הרצת firmware לא חתום, אלא לוודא שלא היה bit flip בדרך.

כל שנשאר הוא לכוון צלחת לשמיים, לקמפל קוד פרי ידינו ולשלוח באמצעות רצף הפקודות הנכון - וכך נוכל להשתלט על הלוויין!

סיכום

סקרנות לגבי לוויינים מודרניים, הובילה אותי לחקור את קוד המקור של PW-Sat2 ולרוורס את הfirmware שפורסם. המחקר הוביל לתובנה שהלוויין אינו מאובטח באמצעים מודרניים, ומנגנון האימות היחיד שקיים על מנת לאמת פקודות שנשלחות מהקרקע הוא קוד בן 32 ביטים שנקבע מראש לפני השיגור. הבנתי שמתקפת Brute Force אינה פיזבילית בגלל קצבי השידור הנתונים, וגם הפרסור של ה-frame מוגן מ-Buffer overflow, אך בהמשך באמצעות קצת מחקר OSINT ורוורסינג גיליתי Secret Leak, צוות העבודה פרסם את ה-firmware שנשלח לחלל, **כולל הקוד הסודי**. הדבר הוביל, תאורטית, לפגיעה בלוויין או הרצת קוד - ניצחון!

כפי שציינתי בתחילת המאמר, פרויקט PW-Sat ממשיך להתקיים והגרסה הבאה כבר בפיתוח. סביר להניח, כי חלקים מקוד המקור של הגרסה שחקרתי משמשים את הגרסה העתידית. כתוצאה מכך, הלווין הבא עלול להיות פגיע באותם מקומות.

עולם ה-CubeSats או ננו-לוויינים מציע כיום "שטח ניסוי" כמעט חופשי - עם קוד פתוח, תיעוד עשיר, ומשימות שאינן תמיד מוגנות מספיק בפני איומי סייבר. תוקף בעל משאבים ויכולת טכנית יכול לנצל את השקיפות הזאת כדי ללמוד, לנצל או לפגוע במשימות חלל. במקרה הזה, השמיים הם לא הגבול ואפשר לדמיין איזה שחקנים שונים ומגוונים עלולים לנסות ולפגוע במשימות חלל.

התחום נמצא כיום בנקודת מפנה: הלוויינים נהיים חכמים, מחוברים, ועשירים ביכולות. בהתאמה, כל איומי הסייבר שאנחנו מכירים בעולם על פני כדור הארץ הופכים להיות רלוונטיים לחלל. זה הזמן שגם אבטחת הסייבר בלוויינים תעבור תפנית. מערכות קריטיות, נתונים רגישים, משימות מדעיות ועוד מונחים על הכף.

המלצות למשימות עתידיות

1. **אימות קריפטוגרפי לכל טלפקודה** - שימוש ב-MAC מבוסס מפתח סימטרי (כגון HMAC-SHA256) או חתימה דיגיטלית א-סימטרית כגון (Ed25519).
2. **רנדומיזציה ורוטציה של מפתחות פר משימה** - כל משימה תקבל מפתח ייחודי שיתחלף לאורך המשימה. אין להשתמש בערכי ברירת מחדל או hard-coded.
3. **הפרדת תחומי הרשאות (Privilege Levels)** - פקודות מסוכנות ידרשו אימות נוסף, חתימה מרובת משתתפים (multi-party auth), או מסלול פיקוד ייעודי.
4. **Rate Limiting וניהול ניסיונות כושלים** - האטה מדורגת של קצב העיבוד, מנגנוני backoff ונעילה (lockout) למניעת מתקפות brute-force או מילון.

5. **אימות קושחה בחתימה (Secure Boot in Space)** - שרשרת אמון מלאה (Root of Trust) מה-bootloader ועד ה-application.

6. **טלמטריה לזיהוי תקיפות** - ניטור ודיווח על התנהגויות חשודות כגון ניסיונות אימות כושלים, שימוש בקודים שגויים, או דפוסי RF חריגים.

7. **ניהול מפתחות ופקודת חירום (Key Rotation & Kill Switch)** - אפשרות לעדכון קוד/מפתח במהלך המשימה, כולל מנגנון לחסימת רכיבים במקרי חירום.

על המחבר

יניב קפיליאנוב, חוקר סייבר, מומחה במחקר ופיתוח Low Level ומערכות הפעלה.

מוזמנים לפנות אליי בכל נושא דרך הלינקדאין שלי.

מקורות מידע

- https://github.com/PW-Sat2/PWSat2OBC - PwSat2 Github Repository
- https://pw-sat.pl/ - האתר הרשמי של הפרויקט
- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10351029 - מאמר על אבטחת סייבר בלווייינים

מבנה הקרנל הלינוקסי וטכניקות ניצול שלו

מאת ירין הרמן

הקדמה

דמיינו שאתם נכנסים למסעדה מפוארת, שלושה כוכבי מישלן. השולחנות ערוכים, המלצרים לבושים בחליפות, והריחות מהמטבח מבטיחים חוויה בלתי נשכחת. אתם פותחים את התפריט ומזמינים מנה עילאית, אבל כשהמנה שלכם מוכנה, היא נזרקת אליכם בתוך קופסת קרטון פשוטה של פלאפל. הטעם אולי מצוין, אבל ברור לכם שהחוויה השתנתה לגמרי. האריזה, הסדר, והאופן שבו הדברים מוגשים – משפיעים לא פחות מהחומר עצמו.

[באיור - המחשה של מסעדת המישלן- (מקור)]

ככה זה גם במחשבים. המשתמשים רואים את הממשק, את התוכנות, את האייקונים היפים – אבל מאחורי הקלעים ישנו מנגנון עצום שמסדר את הכל: מחליט איזה תהליך יקבל זמן ריצה, מי ישמור על הזיכרון, ואיך המשאבים יחולקו. זהו הקרנל של מערכת ההפעלה, והארגון הפנימי שלו הוא לא פחות מה"מסעדה" שבה כל שאר המרכיבים מתקיימים.

אבל כמו שבמסעדה טובה יש גם מי שמנסה להתגנב למטבח האחורי ולגנוב את המתכון הסודי של קציצת הסרטן, גם הקרנל לא חסין. החוליות החלשות שלו, כשנחשפות, הופכות לנתיב כניסה והסלמת הרשאות מפתה מאוד בעבור תוקפים.

במאמר הזה אציג איך הקרנל הלינוקסי בנוי, מה קורה שם מתחת למכסה המנוע, ואיך בדיוק אותם מנגנונים מופלאים שנועדו לשמור על סדר – עלולים להפוך לכלי משחק בידי התוקפים.

תזמון תהליכים ו-Context Switching

ה-scheduler של ה-kernel פועל כמו פקח תנועה, שמחליט איזה תהליך (או thread) ירוץ על ה-CPU הבא ולכמה זמן. לינוקס משתמש ב-preemptive scheduler. כלומר, ה-kernel יכול להפסיק תהליך רץ כדי לעבור לאחר, וכך למנוע מתהליך יחיד להשתלט על ה-CPU. ה-scheduler ברירת המחדל הוא CFS, קיצור של Completely Fair Scheduler. תפקידו הוא לחלק את זמן ה-CPU באופן הוגן בין התהליכים.
ב-CFS, לכל תהליך מוקצה משקל (weight) המבוסס על priority או ערך ה-"nice" שלו, והוא נשמר במבנה של כל התהליכים ה-runnable, כאשר המפתח הוא ה-virtual runtime - ערך זמן שגדל מהר יותר עבור תהליכים עם עדיפות נמוכה. ה-scheduler תמיד יבחר את ה-task עם ה-virtual runtime הקטן ביותר (זה שקיבל הכי מעט זמן CPU ביחס לאחרים) כדי להריץ אותו הבא, מה שמדמה CPU multitasking "הוגן ואידאלי". בצורה זו, כל תהליך מקבל עם הזמן חלק שווה והוגן מה-CPU. ממש קומוניזם בקרנל.

בעוד ש-CFS מטפל בעומסי עבודה רגילים (מדיניות scheduling מסוג SCHED_NORMAL), לינוקס מספקת גם מחלקות scheduler נוספות לצרכים מיוחדים. לדוגמה, real-time scheduling מאפשרת למשימות בעלות עדיפות גבוהה לבצע preempt לאחרות ולהמשיך לרוץ עד לסיום או למשך זמן קבוע. קיימת גם מחלקת SCHED_DEADLINE (מבוססת Earliest Deadline First) עבור משימות רגישות לדדליינים, ומחלקת IDLE ששמה כן היא - עבור עבודות ברקע בעדיפות נמוכה מאוד. לכל מחלקת scheduling יש חוקים משלה, אך כולן משתלבות במסגרת ה-scheduler של ה-kernel.

אז הכל טוב ויפה? יש ויסותים לתהליכים כאילו מדובר בויסותים לחדר אוכל? בזה נגמר הסיפור? אז לא. בוא נקביל את התהליכים לחדר אוכל. יש ויסותים לקורס א' ב-12 ולקורס ב' ב-12:10. מה יקרה כאשר קורס א' מאחר? או שקורס א' מחכה למפקד שלו שישחרר אותו לחדר אוכל? בכל הזמן הזה קורס ב' לא יוכל להיכנס לאכול? מצב בעייתי בלשון המעטה. בשביל זה יש אדם שעומד בקדמת חדר האוכל שיכול לומר לקורס ב' להיכנס לפני קורס א'. אם נצא מההקבלה הזאת, מה קורה כאשר תהליך מסוים צריך לחכות למשהו? או שהוא רץ כבר למספיק זמן? (כלומר לקח יותר מדיי זמן לבחור את המנה הבשרית)? בדיוק בשביל זה יש לנו context switch.

Context Switch מתרחש כאשר ה-scheduler מחליף את ה-CPU מתהליך אחד לאחר. זה יכול לקרות כאשר ה-timeslice של התהליך מסתיים, כאשר התהליך נחסם (למשל בהמתנה ל-I/O), או כאשר תהליך בעל עדיפות גבוהה יותר מתעורר. תהליך ה-context switching כולל שמירה של מצב התהליך הנוכחי ושחזור מצב התהליך הבא. ה-kernel מחזיק עבור כל תהליך מבנה task control block (או בקיצור TCB) מסוג struct, עם כל המידע הדרוש לחידוש הריצה (רשומות CPU, מיפויי זיכרון ועוד). כאשר מתבצע context switch, פונקציית ()schedule של ה-scheduler בוחרת את התהליך הבא להרצה. שגרת ה-context switch ברמה הנמוכה שומרת את רשומות ה-CPU ואת מצב הביצוע של התהליך היוצא, וטוענת את המצב השמור של התהליך הנכנס.

Context Switch תוכנן להיות תהליך מהיר אבל בפועל הוא מוסיף גם הרבה עומס על המעבד. בכל context switch הקרנל צריך:

לשמור את כל המידע של התהליך הנוכחי (מצב רשומות ה-CPU, מצב הזיכרון, מיקום בקוד).
לטעון את כל המידע של התהליך הבא.
לעיתים גם לרוקן (flush) את ה-CPU cache וה-TLB, כי הנתונים שם שייכים לתהליך הקודם ואי אפשר להשתמש בהם בצורה בטוחה עבור התהליך החדש.
לומר את תפילת הדרך.

המשמעות היא שכל זמן שה-CPU משקיע במעבר הזה הוא זמן שבו הוא לא מריץ קוד אמיתי של אף תהליך. אם המעברים האלה קורים לעיתים קרובות מדי, ה-CPU מבזבז יותר זמן על שמירה ושחזור מצבים ופחות זמן על עבודה אמיתית - וזה פוגע בביצועים.

טכניקות ניצול

באגים ב-scheduler עלולים להוביל ל-race condition exploits.

Race Condition הוא מצב שבו התוצאה הסופית של פעולה תלויה בסדר ובעיתוי המדויק שבו שניים או יותר תהליכים/threads מבצעים פעולות - ואם הסדר הזה משתנה, התוצאה עלולה להיות שונה, ולעיתים לא צפויה או לא בטוחה.

ואם נדבר סייברית, ויותר ספציפית בהקשר של הסלמת הרשאות, חולשות Race condition מנצלות את הפער בזמן שבין בדיקת האבטחה לבין השימוש במשאב, כך שהתוקף יכול לשנות את המשאב בזמן הזה ולעקוף את הבדיקה.

ה-scheduler, על ידי ביצוע מהיר של context switching בין threads, עלול בלי כוונה לאפשר מרוצים כאלה אם קוד הקרנל לא מסונכרן. לדוגמה, הפגיעות המפורסמת (Dirty COW CVE-2016-5195) ניצלה Race condition בין שני kernel threads - אחד כתב באופן מתמשך ל- copy-on-write mapping של קובץ (copy on write - מנגנון בזיכרון שגורם לכך שכאשר שני תהליכים חולקים דף זיכרון לקריאה בלבד, ואם אחד מהם מנסה לכתוב אליו - ה-kernel עושה עותק פרטי עבורו, במקום לשנות את המקור), בעוד השני קרא שוב ושוב ל-Syscall שנועדה לאפס את המיפוי הזה. בעצם, במילים פשוטות, מה שקרה הוא:

Thread אחד שניסה ליצור עותק פרטי של קובץ כלשהו כדי שיוכל לכתוב אליו, ו-Thread שני שמנסה לשחרר את המיפוי של אותו קובץ.

התוצאה של זה הייתה הרסנית. משתמש ללא הרשאות הצליח לקבל גישת כתיבה לקבצים בבעלות root שהוגדרו לקריאה בלבד. Dirty COW הצליח כי ה-scheduler ביצע context switches בין ה-threads בדיוק ברגעים הנכונים כדי להטעות את מנגנון ה-copy-on-write.

הסבר יותר מפורט על פגיעות זו קיים בגיליון 77.

אם אתם רוצים לפתח קרנל (א' כל בבקשה אל), אתם חייבים להשתמש ב-locks כדי להגן על נתונים משותפים. אחרת, תוקף עלול לנצל את המרוץ הזה כדי להסלים הרשאות (למשל, על ידי שינוי קובץ לאחר בדיקת האבטחה אך לפני השימוש).

דוגמה קלאסית היא Time-of-check-to-time-of-use במערכות קבצים: תוכנית setuid עם הרשאות root עשויה לבצע stat() על קובץ (לבדוק הרשאות) ואז open() עליו. באותו הזמן, תוקף יכול לתזמן rename על הקובץ, כך שייפתח קובץ אחר לגמרי (אולי שלו, אולי זדוני, מי יודע). מרוצים כאלה אינם באג ישיר ב-scheduler, אך ה-scheduler מאפשר את ההתקפה על ידי context switching בזמן המדויק.

בעבר גם מתקפות (Denial-of-service DoS) היו נושא נוסף לדאגה. תוקף היה יכול לבצע fork למספר גדול של תהליכים או להעמיס על ה-CPU כדי לגרום לכך שמשימות קריטיות לא יקבלו משאבים. ה-scheduler כולל מנגנוני הגנה (כמו nice levels) כדי להתמודד עם זה, אך היסטורית נעשה שימוש ב-fork-bombs או לולאות כבדות ל-CPU כדי לנעול מערכות. כיום ניתן להגביל את כמות תהליכים למשתמש פשוט – באמצעות RLIMIT_NPROC (פקודת ulimit -u מציגה כמה תהליכים משתמש יכול ליצור), אשר מונעת ממשתמש רגיל ליצור מספר אינסופי של תהליכים.

תהליכים זדוניים יכולים גם לנצל עדיפויות scheduling - למשל, קביעת real-time priority ללא מגבלות מתאימות עלולה לאפשר לתהליך עוין להשתלט על ה-CPU.

ניהול זיכרון: Virtual Memory, Paging, and Allocation

בדומה לווינדוס, לינוקס משתמשת בזיכרון וירטואלי כדי לתת לכל תהליך מרחב כתובות מבודד. מרחב הכתובות הווירטואלי של כל תהליך מחולק ל-user-space (לקוד ונתוני האפליקציה) ו-kernel-space (שמורה למערכת ההפעלה). למשל, ב-64 bit, החלק התחתון של מרחב הכתובות הוא זיכרון המשתמש (שונה לכל תהליך), בעוד החלק העליון הוא מרחב הכתובות של ה-kernel (ממופה בכל process לשם יעילות, אך מוגן כך שקוד משתמש לא יכול לגשת אליו).

MMU (קיצור של Memory Management Unit) וטבלאות עמודים (page tables) מטפלות בתרגום בין כתובות וירטואליות לכתובות פיזיות ב-RAM. הקרנל שומר עבור כל תהליך טבלת עמודים מרובת-שלבים (למשל level-4 או level-5 ב-x86-64) שממפה עמודים וירטואליים ל-page frames פיזיים. כאשר תהליך מנסה לגשת לכתובת זיכרון, ה-MMU בודק בטבלת העמודים את המיקום הפיזי המתאים.

מנגנון זה לא רק יוצר את האשליה שלפיה לכל תהליך יש זיכרון רציף שמתחיל בכתובת 0, אלא גם אוכף בידוד (תהליך אחד לא יכול לראות את הזיכרון של אחר אם הוא לא ממופה בטבלת העמודים שלו) והגנה (למשל, עמודים יכולים להיות מוגדרים כ-read-only, או no-execute). אם תהליך ניגש לכתובת שאין לה מיפוי תקף, ה-CPU מייצר page fault, והקרנל או טוען את העמוד (אם הוא זיכרון חוקי שנמצא בדיסק - swapping) או הורג את התהליך במקרה של גישה לא חוקית.

דיברנו על pages בזיכרון, איך הקרנל יודע להקצות ולנהל אותם? הכל בזכות מערכת חברית במיוחד (סטגדיש) בשם Buddy system!

Buddy System מחלק את הזיכרון לבלוקים בגודל של N^2 דפים. הזיכרון הפנוי נשמר ברשימות של בלוקים (buddies) בגדלים שונים של חזקות של 2. כאשר מתקבלת בקשה להקצות זיכרון (למשל, 8 דפים), ה-buddy allocator יחפש את הבלוק הקטן ביותר שיכול לענות על הבקשה. אם נמצא רק בלוק גדול יותר (נניח 16 דפים), הוא יחלק אותו לשניים, ישמור אחד כהקצאה והשני יסמן כזמין. כאשר משחררים זיכרון, ה-buddy allocator ינסה למזג buddies בחזרה לבלוקים גדולים יותר אם אפשר. השיטה הזו יעילה להקצאות בגודל של דף ומעלה, אבל עלולה לסבול מפרגמנטציה בהקצאות קטנות.

להקצאות קטנות יותר (אובייקטים של buffers לדוגמה), הקרנל משתמש ב-slab allocator שמנהל caches של אובייקטים. ה-slab allocator שומר caches עבור סוגי אובייקטים בשימוש תדיר, כאשר כל cache מחלק מראש דפים ל-slots בגודל קבוע עבור אותו אובייקט. כאשר הקרנל צריך מבנה חדש (למשל task_struct או inode), הוא יכול פשוט לקחת אובייקט מאותחל מראש מה-slab cache המתאים. שיטת ה-slab caching משפרת ביצועים וממחזרת זיכרון ביעילות, תוך הפחתת פרגמנטציה בהשוואה להקצאה ישירה מ-buddy system.

באופן כללי בלינוקס הזיכרון מחולק לשני אזורים עיקריים - User Space ו-Kernel Space. User Space הוא האזור שבו רצות האפליקציות והתהליכים של המשתמש, בעוד Kernel Space הוא האזור שבו רץ הקרנל וכל הקוד הקריטי שמנהל תהליכים, זיכרון וכו'.

הקוד והנתונים של ה-kernel נמצאים ב-kernel-space מוגן, שנגיש רק כשה-CPU נמצא ב-Kernel Mode. תהליכי user-space לא יכולים לקרוא או לשנות ישירות זיכרון של ה-kernel או של תהליכים אחרים. גם כאשר תהליך מפעיל system call ונכנס ל-kernel mode, כל מצביע (pointer) שהוא שולח מה-user memory חייב להיבדק ולהיות נגיש רק באמצעות פונקציות בטוחות כמו copy_from_user/copy_to_user, שמוודאות שהכתובות שסיפק המשתמש תקינות ונגישות. כך נשמרת ההפרדה המלאה בין תחומי הזיכרון.

לכל תהליך במערכת יש מרחב כתובות וירטואלי נפרד, אך החלק של ה-Kernel זהה בין כל התהליכים. במצב User Mode לא ניתן לגשת לחלק הזה, וניסיון לעשות זאת יגרום לשגיאת גישה (segmentation fault). המבנה הזה מאפשר שכאשר תהליך מבצע System Call, המעבר ל-Kernel Mode יהיה מהיר, שכן ה-CPU יכול להשתמש באותו זיכרון Kernel ללא צורך בהעתקות נוספות.

בשנת 2018 התגלתה חולשת Meltdown, שאפשרה במעבדים מסוימים לקרוא זיכרון קרנלי מתוך User Space באמצעות מנגנון Speculative Execution, ובכך לעקוף את ההפרדה הזו. הפתרון שהוטמע בלינוקס נקרא (Kernel Page Table Isolation KPTI). בעזרת KPTI, כאשר ה-CPU נמצא במצב User, הזיכרון הקרנלי כלל אינו ממופה ב-Page Table של התהליך, למעט Stub מינימלי הדרוש למעבר למצב לקרנל. ברגע שהמעבד עובר ל-Kernel Mode, לדוגמה בזמן System Call או Interrupt, ה-Kernel מחליף ל-Page Table שמכיל גם את זיכרון הקרנל, וכאשר חוזרים ל-User Mode מוחזר ה-Page Table המצומצם.

אם תוכנית במצב user מנסה לקרוא כתובת של הקרנל, היא תקבל fault או (עם KPTI) הכתובת כלל לא תופיע ב-page table שלה. הפרדה זו מונעת מפוגענים ב-userland לשנות או להשפיע ישירות על מרחב הכתובות הקרנלי.

טכניקות ניצול

Buffer Overflows: טכניקה זו מתרחשת כאשר ה-kernel מעתיק יותר נתונים ל-buffer בזיכרון ממה שהוקצה לו, ובכך כותב על זיכרון סמוך. בקרנל מצב כזה מסוכן במיוחד, משום שהוא יכול לשכתב מבנים קריטיים כמו מצביעי פונקציות, פרטי הרשאות (credentials), או משתנים קריטיים למערכת.

לדוגמה, ב-2021 התגלתה חולשה במודול הרשת TIPC (CVE-2021-43267) שהייתה heap buffer overflow: הקוד הקצה buffer על סמך ערך גודל שנשלט ע"י התוקף, אך השתמש בערך אורך אחר שסופק גם הוא ע"י התוקף (בלי בדיקה מספקת) בעת קריאת memcpy. תוקף היה יכול לשלוח פקטה זדונית שגרמה לקרנל להקצות מקטע heap קטן ואז לכתוב מעבר לגבולותיו, מה שהוביל לשכתוב נתונים סמוכים ב-heap. ניצול overflow כזה אפשר השגת הרצת קוד מרוחק על הקרנל. לא נעים בלשון המעטה. לרוב, ניצולים מהסוג הזה ישכתבו control data (כמו כתובות החזרה ב-stack), או state data חשוב (כמו credentials או flags) כדי לבצע הסלמת הרשאות.

Use-After-Free ו-Heap Spraying: מתקפות UAF מתרחשות כאשר ה-kernel ממשיך להשתמש במצביע לאובייקט לאחר שהוא שוחרר והוחזר ל-allocator. תוקפים יכולים לנצל זאת ע"י גרימה לכך שה-allocator יקצה מחדש את אותו זיכרון לאובייקט שנשלט ע"י התוקף, ובכך לגרום ל-kernel לעבוד על נתונים זדוניים כאילו היו האובייקט המקורי.

לדוגמה, תוקף פותח את הקובץ /dev/ptmx המון פעמים (למשל 100). כל פתיחה כזו גורמת לקרנל להקצות אובייקט חדש מסוג tty_struct בתוך ה-heap.

tty_struct הוא מבנה נתונים שנמצא בקוד שמנהל את מערכת הטרמינלים, הקונסולות ודברים דומים לכך. בין היתר, כל tty_struct מחזיק מצביע לטבלה שנקראת ops table - טבלה של מצביעי פונקציה.

הקרנל משתמש בה כדי לדעת אילו פונקציות להריץ כאשר תהליך של המשתמש עושה פעולות על הטרמינל (כמו ioctl נגיד). אם יש חולשת overflow או UAF, התוקף יכול לשכתב אחד ה-tty_struct האלו.

באופן ספציפי - הוא יכול לשכתב את ה-ops pointer שלו. במקום שאותו מצביע יצביע לטבלה אמיתית של פונקציות, הוא יצביע לכתובת שהתוקף בחר.

כדי להפוך ניצול כזה לאמין, נהוג להשתמש ב-Heap Spraying - הקצאת מספר רב של אובייקטים מסוג מסוים כדי "לרסס" את ה-heap בתבנית צפויה. המטרה היא למלא אזורי זיכרון פוטנציאליים של הקורבן בנתונים נשלטים. Heap Spraying היא למעשה שיטה לעצב את פריסת ה-heap. ה-slab allocators של הקרנל מנסים לבצע רנדומיזציה של מיקומי הקצאות ויש בהם מנגנוני איתור באגים כמו KASAN (קיצור של Kernel Address Sanitizer), אך תוקפים מעצמתיים (ומשועממים מאוד) יכולים לעיתים עדיין "לפסל" את ה-heap באמצעות רצפים מתוכננים של הקצאות ושחרורים כדי למקם את ההקצאה הבאה בדיוק היכן שירצו.

Memory Corruption: בסופו של דבר, רוב טכניקות הניצול של זיכרון מכוונות להסלמת הרשאות. על-ידי פגיעה בזיכרון הקרנל, תוקף יכול למשל לדרוס את מבנה ה-struct cred של התהליך שלו כדי לקבל הרשאות root, או לבטל דגלי אבטחה. בקרנל, המבנה struct cred מכיל את מזהי המשתמש והקבוצה (UID/GID) ואת ה-capabilities של תהליך. ניצולים בקרנל לעיתים מחפשים בזיכרון את ה-cred struct של התהליך הנוכחי ודורסים אותו (או עושים שימוש חוזר באובייקטים של cred ששוחררו) כדי להגדיר UID=0.

קיימות גם טכניקות כמו "return-to-direct-mapped memory" - מאחר שהקרנל ממפה ישירות את הזיכרון הפיזי במרחב הכתובות שלו, overflow יכול למקם מבנה מזויף בזיכרון במקום שבו הקרנל עלול להשתמש בו.

Bypassing Memory Isolation: חלק מהניצולים מכוונים לעקוף את מנגנון ההפרדה הבסיסי של הזיכרון. לפני שפותחו מנגנוני הגנה, משתמש יכול היה למפות זיכרון בכתובת 0 ולנצל באגים של NULL pointer dereference בקרנל. אם הקרנל ניסה לגשת בטעות למצביע NULL, בפועל הוא היה ניגש לעמוד זיכרון שנשלט על-ידי המשתמש, ובכך אפשר להריץ קוד זדוני. בקרנלים מודרניים נמנע מיפוי כתובות נמוכות על-ידי משתמשים, ותכונות חומרה כמו SMAP למשל מונעות מהקרנל להריץ קוד ישירות מ-User Space או לגשת לזיכרון משתמש ללא בדיקה מתאימה. בנוסף, מתקפות Side-Channel כמו Meltdown CVE-2017-5754 הצליחו לפרוץ את מנגנון ההפרדה על-ידי קריאת זיכרון קרנל באמצעות speculative execution (מעין "ניחוש" של הקרנל איזה הוראה תתבצע הבאה). בתגובה לכך, פותח KPTI, שכתבתי עליו לעיל.

System Calls - ממשק לקרנל

תוכניות שרצות ב-User Space אינן יכולות לגשת ישירות לחומרה או לפונקציות של הקרנל; עליהן לבקש שירותים באמצעות קריאות מערכת (sycalls) - נקודות הכניסה המוגדרות ממצב user ל-Kernel Mode. קריאת מערכת דומה לקריאה לפונקציה בקרנל, אך מיושמת באמצעות פקודת CPU מיוחדת או interrupt

(ארחיב על כך בחלק הבא של המאמר) שגורמות ל-trap - מצב שבו המעבד עוצר את רצף הביצוע הרגיל ועובר באופן מתוכנן למצב קרנל. לדוגמה, בארכיטקטורת x86-64, קוד משתמש מפעיל syscall על-ידי טעינת מספר הקריאה ל-rax, ואז ביצוע הפקודה syscall.

כאשר trap מתרחש, ה-CPU מבצע מספר פעולות אוטומטיות:

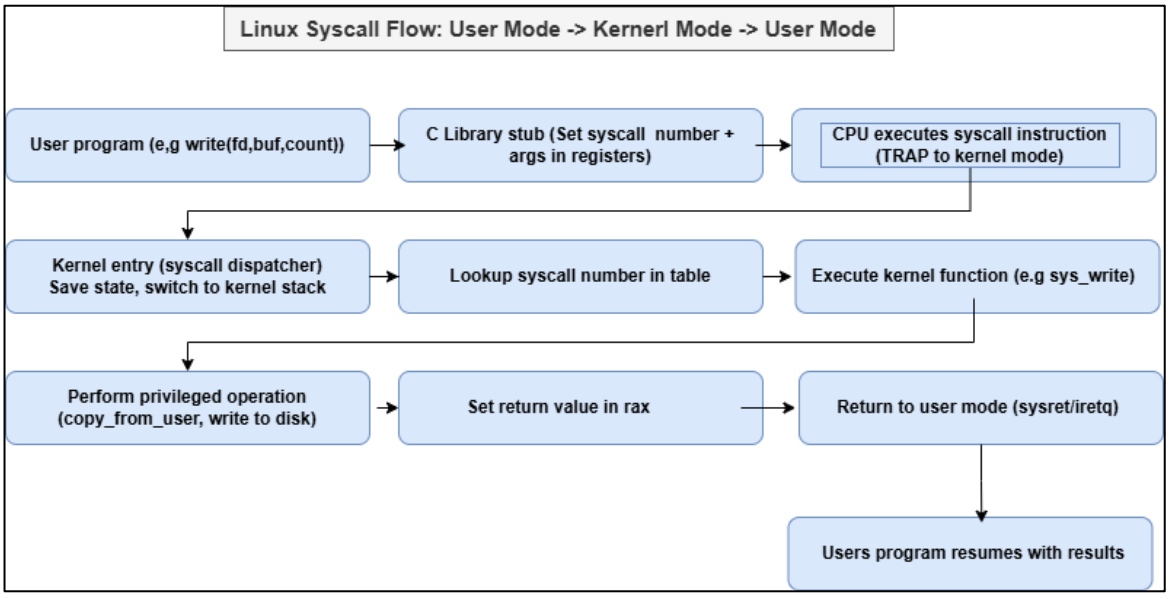
הוא עובר למצב 0 ring

עובר ל-stack קרנל ייעודי

קופץ לנקודת כניסה מוגדרת מראש בקרנל - ה-syscall dispatcher

בשלב זה התהליך נמצא "בתוך הקרנל" ומריץ קוד. ה-dispatcher של הקרנל שומר את הרישומים של המשתמש, ולאחר מכן משתמש במספר הקריאה כדי לאתר בטבלת ה-syscalls (מערך של מצביעי פונקציות) את המימוש של הקריאה. לדוגמה, אם הערך ב-rax הוא 60 (המספר של sys_exit ב-x86-64), ה-dispatcher יקרא לפונקציה של exit.

עכשיו, איך המעבר בין פונקציה פשוטה ב-User Mode להרצת קוד בקרנל מתרחש? נבין באמצעות דוגמה. תוכנית משתמש קוראת לפונקציה ()write. ספריית ה-C מכינה את הארגומנטים ומבצעת את קריאת המערכת write. ה-CPU עובר לKernel Mode, קוד הכניסה של הקרנל שומר את מצב ה-CPU ומזהה את מספר ה-syscall. לאחר מכן הוא קורא לפונקציה sys_write דרך טבלת ה-syscalls. המימוש של sys_write יעתיק את הנתונים מה-buffer של המשתמש לזיכרון הקרנל ויבצע את פעולת הכתיבה. בסיום, הוא קובע ערך חזרה (למשל מספר הבתים שנכתבו) ב-rax ומחזיר. בסופו של דבר, ה-dispatcher של ה-syscall מכין את החזרה ל-user mode: הוא משחזר את מצב ה-CPU והרישומים השמורים של התהליך ומחזיר את המעבד ל- user mode. מנקודת מבט של תוכנית המשתמש, היא פשוט קראה ל- ()write וקיבלה תוצאה בחזרה - מבלי לראות את כל מנגנון המעבר המורכב שמתרחש מאחורי הקלעים.



במהלך syscall, הקרנל רץ בהרשאות מלאות ולכן הוא חייב לאמת את כל הקלטים שמגיעים מ-User Space לפני שהוא משתמש בהם. הדבר כולל מצביעים - חייבים לוודא שהם מצביעים על זיכרון ששייך באמת לתהליך הקורא ונמצא ב-user space, ולא על זיכרון הקרנל - וגם גדלים, כדי למנוע buffer overflow בקרנל.

לדוגמה, הקרנל לא יבטח בצורה עיוורת במספר שמגיע מ-user space אם הוא מתכוון להשתמש בו כאינדקס במערך או כאורך - הוא יוודא שהוא נמצא בטווח הצפוי. אם syscall מצפה למצביע למבנה, הקרנל עשוי קודם להעתיק את המבנה הזה ל-buffer פנימי שלו, או לכל הפחות לוודא שכל שדה בו בטוח לשימוש.

בדיקה של מצביעים חשובה במיוחד: הקרנל לעולם לא ידרוס או ייגש ישירות לכתובת שסופקה מה-user אם היא מצביעה למרחב הקרנלי. אילו היה עושה זאת, תוקף יכול היה להעביר ל-syscall כתובת מתוך זיכרון הקרנל, ולגרום לכך שהקרנל יקרא או יכתוב לזיכרון שלו עצמו - מה שהיה מפר את מנגנון ההפרדה בין הרשאות. נאמר כספוילר להמשך - יש שהיו אומרים שמה שכתבתי עכשיו זה "הוראות שנכתבו בדם".

ניתן דוגמה היפותטית: אם הקרנל לא היה בודק את המצביע בקריאת write(fd, buf, count),

תוכנה זדונית הייתה יכולה להעביר מצביע למבנה פנימי של הקרנל. הקרנל היה מעתיק את התוכן הזה ואולי כותב אותו לקובץ, מה שהיה מוביל לדליפת מידע מהקרנל למשתמש. להיפך - אם בקריאת read() לא הייתה מבצעת בדיקת מצביע, ייתכן שהקרנל היה כותב לתוך כתובת בזיכרון הקרנלי, ובכך משחית את הנתונים שלו עצמו.

בקרנלים מודרניים נמנעים מכך באמצעות פונקציות כמו copy_from_user(), שנכשלות אם הכתובת שסיפק המשתמש אינה כתובת תקפה ב-user space. בנוסף, נעשה שימוש במנגנוני חומרה כמו SMAP (Supervisor Mode Access Prevention), שמונעים מהקרנל לגשת ישירות לזיכרון של משתמש אלא אם נאמר אחרת.

טכניקות ניצול

קריאות מערכת הן משטח התקיפה העיקרי עבור תוקפים, משום שדרכו ה-userland מתקשר עם ה-kernel. רבות מהחולשות ההיסטוריות בלינוקס נבעו מפגיעויות במימושי syscalls. אפשר להתייחס לתקיפות הפוטנציאליות דרך syscalls ממש כמו השיר דיינו בהגדת פסח.

אם ה-syscall handler לא מאמת כראוי את הארגומנטים, תוקפים יכולים לנצל זאת, דיינו. כפי שצוין, היעדר ולידציה של מצביעים (pointers) עשוי לאפשר גישה לזיכרון ה-kernel. לדוגמה, נניח syscall היפותטי sys_mycopy(dst, src, len) שממומש בלי לבדוק ש-dst היא כתובת ב-user-space. תוכנה זדונית יכולה לקרוא ל-syscall עם dst שמצביע למבנה רגיש של ה-kernel ו-src לנתונים בשליטתה – ה-kernel עלול אז להעתיק את הנתונים למבנה שלו עצמו ולדרוס אותו. זאת הסלמת הרשאות פשוטה במלוא מובן המילה. רק חבל שהעולם לא כל כך פשוט (או בעצם לא חבל, תלוי את מי שואלים). כיום, מפתחי הקרנל דואגים מאוד לוולידציה על כל הפרמטרים וקריאות המערכות השונות.

דוגמה היסטורית נהדרת לניצול חולשה מסוג זה היא CVE-2014-3153. באג זה היה ב-futex syscall, שמנהל Userspace Mutex. הפגיעה התמקדה בפונקציה futex_requeue() שאיפשרה להעביר תהליכים מ-wait queue אחת לאחרת. הבעיה נבעה מבדיקות חסרות על פרמטרים שהגיעו מ-user space, במיוחד במנגנוני Priority Inheritance. תוקף יכל לנצל את החוסר בסנכרון ולגרום ל-race condition, שבסופו הקרנל עבד עם מצביעים לא תקינים או שכבר שוחררו. ניצול מעשי (כמו ה-Towelroot exploit באנדרואיד) שינה את מבנה ה-cred של התהליך הנוכחי כדי להפוך את ה-UID ל-0.

חלק מה-syscalls מקבלים נתונים מ-user space – לדוגמה, read(), או קריאות ioctl() שמעבירות מבנים. אם קוד הקרנל שמטפל בנתונים האלה לא מבצע בדיקות אורך בצורה נכונה, עלול להתרחש Buffer Overflow ולגרום לכתיבה מעבר לגבולות ה-kernel buffer, דיינו.

דוגמה חדשה יחסית היא החולשה בקריאת sendmsg() (CVE-2019-20386). קריאת מערכת זו מאפשרת שליחת הודעות עם control messages, כולל העברת descriptors בין תהליכים. כאשר תהליך שלח control message מסוג scm_rights עם נתונים מסוימים, הקרנל לא בדק את אורך הקלט מול גבולות buffer, ומשם הדרך להסלמת הרשאות או להקרסת המערכת קצרה מאוד.

ישנן syscalls שבבסיסן לגיטימיות אך עלולות להיות מסוכנות אם מופעלות ללא בקרות מתאימות, דיינו. תוקפים מחפשים דרכים לקרוא להן גם כשאין להם הרשאות לכך. למשל, ה-kexec_load syscall מאפשר למשתמש root לטעון קרנל חדש לזיכרון – ברור שמשתמש לא-מורשה לא אמור לקרוא לזה. אם מתרחשת תקלה זמנית בהרשאות (אפילו הרשאת ה-sudo הכי קטנה), תוקף יכול להשתמש ב-syscall כזה כדי להסלים את ההרשאות שלו לצמיתות, למשל על-ידי טעינת קרנל משלו.

אני אצטט עכשיו משפט שמפתחים מאוד אוהבים לומר - זה לא באג זה פיצ'ר. אלה הן קריאות מערכת לגיטימיות שקיימות בעבור פיתוח לגיטימי, אך אם לא מגבילים אותם בצורה מספקת - נחשפים למרחב תקיפה שלם.

אף על פי שלא מדובר בדרך כלל בהסלמת הרשאות ראשונית, לאחר שתוקף כבר השיג גישת כתיבה לקרנל הוא יכול לבצע hooking לטבלת ה-syscalls כחלק מ-rootkit. הרעיון הוא להחליף מצביע בטבלת ה-syscalls כך שיצביע לפונקציה זדונית במקום למימוש המקורי.

באופן זה התוקף יכול ליירט קריאות מערכת – למשל, לחבר מחדש את open כדי להסתיר קבצים מסוימים, את getdents כדי להסתיר תיקיות, או את execve כדי להעניק הרשאות גבוהות אוטומטית לכל תוכנית שמורצת.

במערכות מודרניות קיימות הגנות כמו קרנל עם זיכרון לקריאה בלבד, וכן מנגנוני integrity checkers שמסוגלים לעיתים לזהות טבלאות syscalls שהשתנו. צעדים אלה נועדו להפחית את הסיכוי ש-rootkits יצליחו להשתמש ב-Syscall Table Hooking מבלי להתגלות.

Interrupts הם אותות מחומרה או מתוכנה שעוצרים זמנית את ריצת ה-CPU ומעבירים אותו ל-handler routine. כאשר התקן (כמו מקלדת לדוגמה) דורש תשומת לב, הוא שולח Interrupt Request (IRQ) ל-CPU. ה-CPU, עם קבלת ה-interrupt, שומר את ההקשר הנוכחי שלו (בדומה ל-context switch) וקופץ ל-Interrupt Service Routine (ISR) של אותו interrupt. כל IRQ חומרתי ממופה ל-ISR, וה-kernel מגדיר זאת ב-Interrupt Descriptor Table (IDT). חשוב לציין שכאשר ISR רץ, הוא נמצא ב-context מיוחד: אין לו תהליך משתמש משויך. המשמעות היא ש-ISR לא יכול "לישון" או לבצע עיבוד ממושך - עליו לפעול במהירות ולאשר/לאפס את ה-interrupt בהתקן. דומה מאוד לאיזושהי מערכת הפעלה לא? שכחתי איך קוראים לה. הלכתי להסתכל מעבר לאדן החלון. אולי אזכר.

במערכות מרובות-מעבדים (שזה רוב המערכות כיום), ניתן לנתב Interrupts אל ליבות ספציפיות. כברירת מחדל, ייתכן שכל ה-hardware interrupts ינותבו ל-CPU0, אך זה אינו אופטימלי משום שזה עלול להעמיס על מעבד יחיד. IRQ balancing הוא הפרקטיקה של הפצת עומס הטיפול ב-interrupts בין ליבות שונות. הקרנל מאפשר להגדיר עבור כל interrupt ערך בשם IRQ affinity - כלומר bitmask של ליבות ה-CPU המורשות לטפל באותו interrupt.

במערכות רבות פועל תהליך משתמש בשם irqbalance, כלומר daemon שמבצע התאמות אוטומטיות בפרקי זמן קבועים: הוא משנה את ה-affinities כדי לאזן את העומס, ומעביר interrupts מליבה עמוסה לליבה פנויה.

טכניקות ניצול

בצורה כללית אפשר להגיע למצב של Race Condition דרך Interrupts. כלומר, ה-Interrupt עצמו הוא לא הניצול, אך דרכו אפשר להגיע להרצת קוד בקרנל. לדוגמה, אם הקרנל מסמן buffer כמשוחרר ואז מפעיל פעולה חומרתית - interrupt שמגיע בזמן הזה יכול להקצות מחדש את ה-buffer לפני שהפעולה החומרתית משתמשת בו, מה שיוצר UAF. תוקפים שיש להם שליטה מסוימת על מקורות interrupts יכולים לתזמן מצבים כאלה. רבים מהדרייברים של ההתקנים כוללים interrupt handlers שהם חלק מהקרנל. באגים בתוך interrupt handler מסוכנים במיוחד משום שהם רצים ב-IRQ context, לרוב בהרשאות גבוהות ועם מעט מאוד הגנות (למשל, לעיתים לא כל בדיקות האבטחה של system calls מתבצעות שם). אם תוקף יכול לשלוח נתונים או אותות שמנצלים חולשה ב-ISR, הוא יכול להריץ קוד ב-Kernel Mode או להפיל את המערכת.

דוגמה קונקרטית היא ניצול בדרייבר USB - CVE-2016-2384. כאשר התקן כזה חובר, תת-המערכת של ה-USB בקרנל (שרצה בחלקה בזמן interrupt בעת חיבור התקנים) טיפלה באופן שגוי ב-descriptor לא צפוי ושחררה אובייקט פעמיים. כשהקרנל משחרר אובייקט פעמיים, ה-allocator של ה-heap עלול "לחשוב" שהזיכרון פנוי, ובפועל לשמור עליו ברשימת Free. תוקף מעצמתי יכול לעשות Heap Spraying (כתבתי על כך לעיל) ובסופו של דבר לקבל הרצת קוד ב-kernel.

סיכום

בסופו של דבר, הקרנל של לינוקס הוא לא רק "צינור" שמחבר בין המשתמש לחומרה – הוא המוח שמנהל את כל המערכת. מהניהול של זמן המעבד בעזרת ה-CFS, דרך חלוקת הזיכרון במנגנוני Paging ו- Buddy Allocation, ועד לניהול קריאות המערכת והגנה על המרחב הקריטי של הקרנל – הכל עובד בהרמוניה עד שמישהו מנסה לשבור את הכללים. לאורך השנים הקרנל השתדרג כל הזמן, ועבר שיפורים שצמצו משטחי תקיפה קיימים אך גם יצרו משטחי תקיפה חדשים. הקרנל הלינוקסי הוא עולם אינסופי, שמכיל בתוכו כל הזמן הפתעות.

על המחבר

שמי ירין הרמן, וזהו המאמר השלישי שלי. לכל שאלה, תגובה או טענה, מוזמנים לכתוב לי במייל:

yarinherman05@gmail.com

מקורות

- https://documentation.ubuntu.com/real-time/latest/explanation/schedulers/
- https://linux-kernel-labs.github.io/refs/heads/master/so2/lec3-processes.html
- https://hackmd.io/@bachtam2001/BkZkudoLq
- https://www.cs.toronto.edu/~arnold/427/18s/427_18S/indepth/dirty-cow/index.html
- https://www.form3.tech/blog/engineering/linux-fundamentals-user-kernel-space
- https://www.geeksforgeeks.org/operating-systems/operating-system-allocating-kernel-memory-buddy-system-slab-system/
- https://www.sentinelone.com/labs/tipc-remote-linux-kernel-heap-overflow-allows-arbitrary-code-execution/
- https://www.kernel.org/doc/gorman/html/understand/understand014.html
- https://santaclz.github.io/2024/01/20/Linux-Kernel-Exploitation-Heap-techniques.html
- https://thinkty.net/general/2024/04/29/bottom_half.html
- https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yoochan
- https://enterprise-support.nvidia.com/s/article/what-is-irq-affinity-x
- https://xairy.io/articles/cve-2016-2384
- https://linux-kernel-labs.github.io/refs/pull/282/merge/lectures/syscalls.html

הצפנה מקצה לקצה - המשך

מאת עידן שכטר

הקדמה

במאמר הקודם הבנו את החשיבות של הצפנה מקצה לקצה בימינו, ואף למדנו כיצד לממש מנגנון בסיסי בעצמנו, כזה שעל פניו נראה נטול בעיות ופגמים.

במאמר זה נבצע עליית מדרגה ונבין את הקושי במימוש מנגנון הצפנה מקצה לקצה כאשר איום הייחוס גבוהה, לדוגמא, במצב בו המפתח הפרטי של אחד הצדדים נגנב.

מצב זה אמנם נשמע חריג (והוא אכן כך, מאחר והאדם הפשוט לא נמצא תחת איום ייחוס שכזה), אך מימוש פרוטוקולי הצפנה מורכבים ומתקדמים הפך להיות סטנדרט - אפליקציות פופולריות רבות עובדות קשה כדי לאפשר גם ל"משתמש הפשוט" להנות ממנגנוני אבטחה המשפרים את הפרטיות שלו פלאים.

התרחיש שנרצה למנוע

כפי שלמדנו במאמר הקודם, הסוד המשותף המיוצר על ידי שני משתמשים המעוניינים לתקשר בצורה מוצפנת הוא "נגזרת" של מפתחות ההצפנה שלהם, או בצורה קצת יותר מדויקת - של המפתחות הפרטיים שלהם.
כמובן שגם למפתחות הפומביים יש חשיבות, והם נחוצים לטובת חישוב הסוד המשותף, אבל אותם ניתן להשיג בקלות לעומת המפתחות הפרטיים, עליהם נשאף לשמור ככל שביכולתנו.

התרחיש עליו נדבר במאמר זה מתחיל כאשר המפתח הפרטי של אחד הצדדים מגיע לגורם שלישי ללא ידיעתו, מצב המאפשר לאותו גורם לייצר את הסוד המשותף ולפענח את ההודעות המוצפנות הנשלחות בין שני המשתמשים ו"לשבור" את מנגנון ההצפנה מקצה לקצה.

מקור הבעיה

הפתרון שנציע לבעיה מתבסס על העובדה שגורם שלישי בהכרח הצליח לשים ידו על המפתח הפרטי שלנו. בהנחה והמפתח הפרטי שלנו נמצא בידיו של גורם שלישי שכעת מסוגל לפענח הודעות מוצפנות, למה שלא פשוט נחליף את המפתח הפרטי שלנו אחת לכמה זמן? אם המפתח אכן נגנב, בשלב מסוים הוא יוחלף במפתח חדש ויהפוך לא רלוונטי!

בנוסף, בהנחה ונדאג להיפטר ממפתחות ישנים, פתרון זה מונע מגורם שלישי שהשיג את המפתח הפרטי הנוכחי לפענח הודעות שנשלחו על סמך מפתחות ישנים יותר (מאפיין המכונה "סודיות מושלמת קדימה", בו נגע בהמשך המאמר).

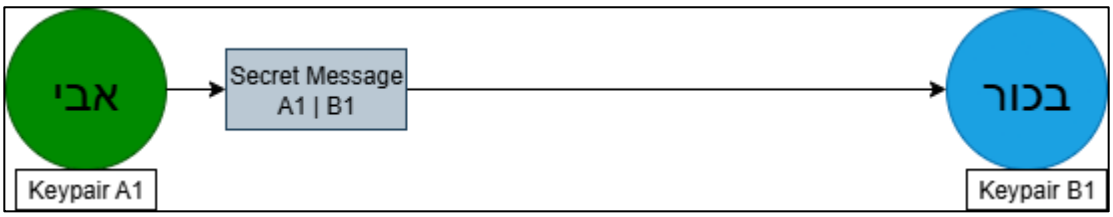
אבל רגע, אם הגורם השלישי כבר הצליח להשיג את המפתח הפרטי שלנו, למה שלא יצליח לעשות זאת שוב ולהשיג גם את המפתח החדש?

גם אם נתעלם מעובדה זו, ניסיון מימוש מנגנון שכזה ילמד אותנו שניהול שיחה מוצפנת מתמשכת תוך כדי חידוש מפתחות היא אינה פעולה של מה בכך.

ריענון מפתחות

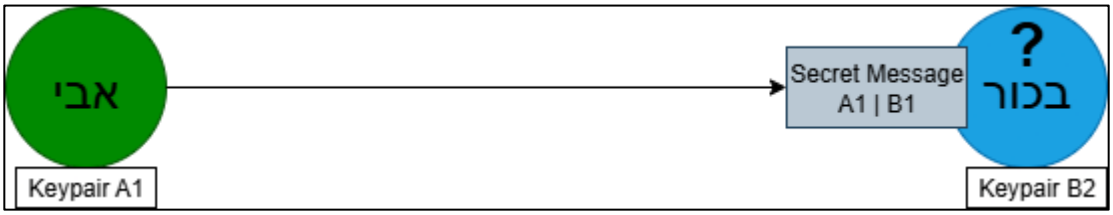
הבעיה המרכזית במימוש מנגנון שמבצע "ריענון" מחזורי של מפתחות הצפנה היא הקושי בסנכרון בין שני הצדדים לאורך זמן.

כאשר אבי שולח לבכור הודעה, ההודעה תוצפן באמצעות סוד משותף המבוסס על המפתחות האסימטריים שיש לכל אחד מהם באותו רגע נתון:

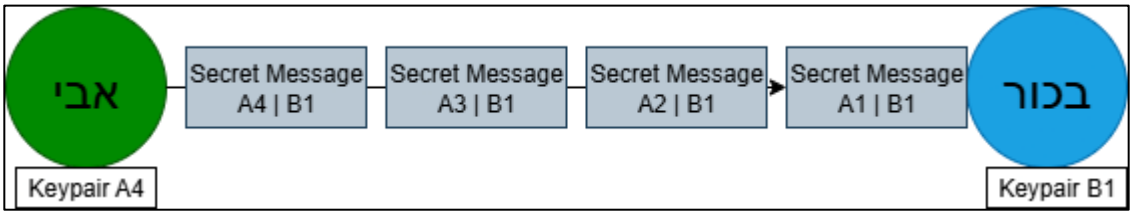


דוגמא פשוטה לבעיית סנכרון היא מצב בו אבי ישלח לבכור הודעה על סמך צמד המפתחות B1 (כלומר, ישתמש במפתח הפומבי B1 על מנת לחשב את הסוד המשותף) אך בדיוק ברגע שיסיים להצפין את ההודעה וישלח אותה לשירות, בכור יבצע ריענון מפתחות ויעבור להשתמש בצמד המפתחות B2.

מאחר ובכור לא שומר את המפתחות הישנים שלו, לא יוכל לפענח את ההודעה שקיבל מאבי. (נוכל לפתור את בעיה זו בכך שנוסיף לפרוטוקול מנגנון ש"נועל" מפתחות בין שני הצדדים, ומוודא שהמפתח בו נעשה שימוש ברגע הנתון אכן עדיין רלוונטי, אך פתרון זה לא פותר בעיות משמעויות אחרות).



דוגמא נוספת לבעיית סנכרון היא מצב בו אחד הצדדים לא מתחבר לשירות הרלוונטי במשך זמן, ובכך לא מעדכן את המפתחות שלו, בעוד שהצד השני ממשיך לשלוח לו הודעות ולהחליף מפתחות תוך כדי.



במידה והמפתח של בכור הוא זה שנגנב, פענוח ההודעות עדיין יהיה אפשרי על ידי גורם שלישי, וזאת כי רק הצד השני החליף את המפתחות שלו (חישוב הסוד המשותף יתבסס על המפתח הפרטי של בכור, זה שנגנב, ועל המפתח הפומבי הנוכחי של אבי, שניתן להשיג בקלות).

בנוסף, בכור לא יוכל לפענח הודעות שהוצפנו על בסיס המפתחות A1, A2, A3 של אבי, וזאת משום שאבי רענן את המפתחות שלו מספר פעמים בזמן שבכור לא היה מחובר לשירות.

One Keypair to Rule Them All

בעיה משמעותית נוספת שלאו דווקא נובעת משימוש במפתחות סטטיים (חידוש מפתחות לא ימנע את הבעיה) היא העובדה שזוג מפתחות בודד ישמש את המשתמש כדי להצפין הודעות בכלל השיחות של השירות, לכן גורם שלישי השם ידו על המפתח הפרטי יוכל לחשב את הסוד המשותף עבור כל שיחה רלוונטית, ולראות את כלל ההודעות של המשתמש.

אנו למדים ששימוש במנגנון הצפנה שתיארנו, כזה הנחשב לפרימיטיבי ועושה שימוש במפתחות קבועים באופן רחב, אינו פתרון יעיל ומספק כאשר מדובר באיום הייחוס הרלוונטי.

לשם כך, נדרש פתרון חדשני יותר. אפליקציית סיגנל, הנחשבת לאחת מאפליקציות המסרים המיידיים המאובטחות ביותר כיום, מציעה פתרון יפיפה לבעיה זו בפרוטוקול קוד פתוח המכונה גם הוא סיגנל, פרוטוקול בו עושות כיום שימוש אפליקציות כמו וואטסאפ, פייסבוק וסקייפ.

אפליקציית Signal

סיגנל התחילה את דרכה ב-2010 כאפליקציה בשם TextSecure, אשר אפשרה למשתמשיה לתקשר בצורה מוצפנת מקצה-לקצה בקלות, כחלק ממיזם בשם Whisper Systems שהוקם על ידי החוקר והיזם מוקסי מרלינספייק.

לאחר ש-Whisper Systems נרכשה על ידי Twitter (כיום X), אפליקציית TextSecure הופצה חינם כקוד פתוח.

זמן לא רב לאחר מכן, מוקסי עזב את Twitter והקים את Open Whisper Systems במטרה להמשיך ולפתח את TextSecure ואפליקציה נוספת, RedPhone, אשר אפשרה למשתמשיה לנהל שיחות קוליות מוצפנות מקצה לקצה.

ב-2015 Open Whisper System מיזגו את האפליקציות TextSecure ו-RedPhone אל אפליקציה יחידה בשם Signal, שהפכה עם השנים ל-Signal שאנו מכירים היום.

בימים אלה, אפליקציית Signal מפותחת על ידי Signal Messenger LLC, אשר ממומנת על ידי Signal Technology Foundation - ארגון ללא מטרות רווח שהוקם על ידי מוקסי מרלינספייק ובריאן אקטון (ממקימי WhatsApp).

השפעה

TextSecure הייתה מהאפליקציות הראשונות שהציעו הצפנה מקצה לקצה והובילה תהליך משמעותי שהפך את ההצפנה החזקה למיינסטרים - שינוי תפיסה עצום לעומת העבר, שבו פרטיות נחשבה כמשהו עבור "אלו שיש להם מה להסתיר".

> ***"Arguing that you don't care about the right to privacy because you have nothing to hide is no different than saying you don't care about free speech because you have nothing to say"***
>
> ***- Edward Snowden***

מעבר לכך שאפליקציית סיגנל היא חינמית וקוד פתוח, גם השרת של סיגנל מופץ כקוד פתוח, דבר המאפשר לאזרחים במדינות בהן הגישה לשירותי סיגנל חסומה להרים תשתית מקומית משלהם, לתקשר בצורה בטוחה ולהבטיח את החירות שלהם לפרטיות.

פרטיות הינה אידיאולוגיה עבור סיגנל, ומעבר לתחזוקה של פרוטוקול הצפנה מדהים כקוד פתוח, סיגנל נמנעת באופן מוחלט משמירה של מידע על המשתמשים שלה, ולכן גם אם תשתף פעולה עם רשויות, תוכל לספק להם מידע מינימלי. עובדה זו הופכת את השימוש באפליקציה לאטרקטיבי עוד יותר עבור אותם אינדיבידואלים המעוניינים להבטיח את הפרטיות שלהם. (כמו כן, אי אפשר להתעלם מהעובדה שמדובר בחרב פיפיות.

בדיוק כמו קריפטו ו-TOR, גם היתרונות של סיגנל מנוצלים כדי לברוח מהחוק ולבצע פשעים).

פרוטוקול Signal

מאחר ומדובר בפרוטוקול מסובך ורחב, לא נצלול אל עומק כל פרטיו הטכניים.

המרכיבים העיקריים עליהם נדבר הם פרוטוקול החלפת המפתחות X3DH ואלגוריתם Double Ratchet, עליהם ניתן להסתכל כמנגנון דו שלבי המאפשר לשני משתמשים לתקשר בצורה מוצפנת תוך כדי חידוש מפתחות מתמשך, גם כאשר אחד הצדדים אינו מחובר!

X3DH

פרוטוקול X3DH (Extended Triple Diffie-Hellman) מאפשר לשני משתמשים לייצר סשן באמצעותו יוכלו לתקשר באופן מוצפן ובטוח.

על סשן ניתן להסתכל כמצב קריפטוגרפי משותף בין שני משתמשים, המכיל את המידע הדרוש להם כדי לתקשר על פי הפרוטוקול הרלוונטי.

לדוגמא, ע"פ הפתרון הפרימיטיבי עליו דיברנו עד כה, יצירת הסשן שקולה ליצירת סוד משותף בעזרת DH. במקרה של סיגנל, המשתמשים יעשו שימוש ב-X3DH, גרסה "משודרגת" של DH, על מנת לייצר סוד משותף שבעזרתו יוכלו להתחיל לתקשר האחד עם השני.

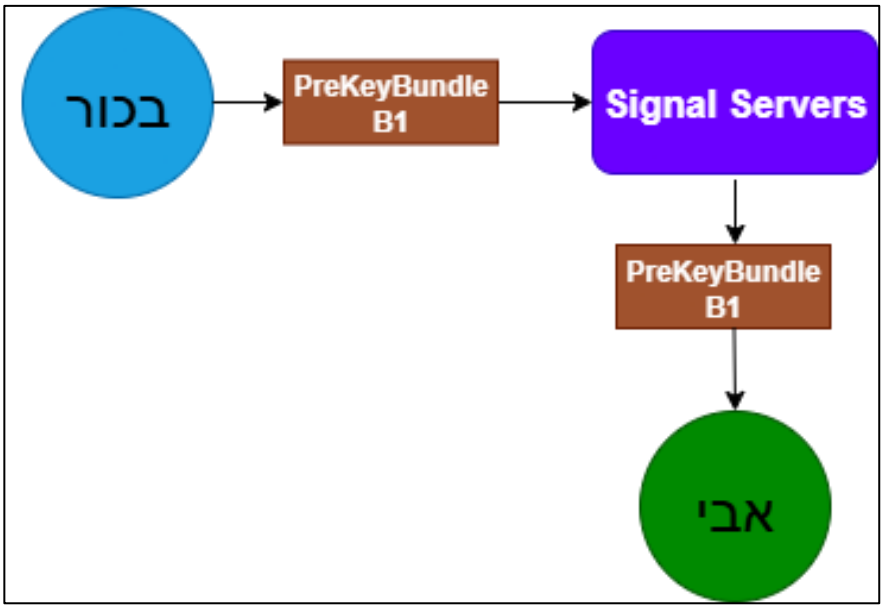
על פי פרוטוקול סיגנל, כל משתמש מחזיק זוג מפתחות אסימטריים הנקראים **Identity Keys**, אשר משמשים כזהות הקריפטוגרפית שלו. בשונה מהמקרה הקלאסי עליו דיברנו עד כה, זוג המפתחות משמש לאותנטיקציה ולא להצפנה.

מפתחות ההצפנה בסיגנל מבוססים עקומים אליפטיים (EC) ומשמשים לביצוע Diffie-Hellman וגם לחתימה דיגיטלית, לדוגמא 25519Ed.

מאחר ולא נצלול לחתימות דיגיטליות במאמר זה, עיקרון שחשוב לזכור הוא שחתימה דיגיטלית אשר בוצעה על ידי מפתח פרטי, תוכל להיות מאומתת על ידי המפתח הפומבי המתאים. דבר המאפשר למשתמש להוכיח שהוא מחזיק במפתח פרטי מסוים ובכך להוכיח את הזהות שלו.

Key Publishing

כל משתמש בסיגנל מעלה לשרתים של סיגנל אחת לכמה זמן "חבילת" מפתחות הנקראת PreKeyBundle המאפשרת למשתמשים אחרים לייצר מולו סשן קריפטוגרפי, גם כאשר הוא אינו מחובר לשירות! כאשר אבי ירצה לייצר סשן מול בכור כדי לשלוח לו הודעה, הוא יפנה לשרתי סיגנל ויבקש לקבל PreKeyBundle רלוונטי.



PreKeyBundle מכיר מספר פרמטרים שונים, אך אלה הנחוצים לביצוע X3DH הם:

Public Identity Key - מפתח הזהות הפומבי של בכור, מפתח קבוע המשמש כתעודת הזהות הקריפטוגרית של בכור בעולמות סיגנל, ובין היתר משמש כדי להוכיח את הזהות שלו.

Signed PreKey - מפתח זמני (מתחדש אחת לכמה זמן) אשר נחתם על ידי מפתח הזהות הפרטי של בכור (Private Identity Key). תפקידו העיקרי של מפתח זה הוא להוכיח את האותנטיות של ה-PreKeyBundle הרלוונטי ולמנוע מצב בו ישתמש ב-PreKeyBundle שלא הועלה לשירות על ידי בכור. (ניתן לוודא כי החתימה אכן בוצעה על ידי ה-Private Identity Key של הישות איתה נרצה לתקשר).

One-Time PreKey - מפתח חד פעמי המשמש ליצירת סשן אחד בלבד! (בפועל, בכור יעלה מספר גדול של One-Time PreKeys לשירות (כ-100 במספר) כדי לאפשר למשתמשים שונים לייצר איתו סשן. האפליקציה תוודא אחת לכמה זמן שכמות ה-One-Time PreKeys שבשרת לא נמוכה מדי ולא תמנע יצירה של סשנים עתידיים, אך גם במקרה זה יש פתרון, עליו לא נדבר במאמר זה).

לאחר שאבי יבקש את ה-**One-Time PreKey** הרלוונטי, הוא ימחק מהשרת.

X3DH In Practice

לאחר שאבי קיבל את ה-PreKeyBundle של בכור ווידא את האותנטיות שלו (בדק את חתימת ה- Signed PreKey), עליו לייצר זוג מפתחות זמני (Ephemeral KeyPair) וכעת הוא מוכן לבצע X3DH. בפועל, X3DH הוא שרשור של מספר חישובי DH (DH1 מייצג את ערך ה-DH הראשון שנחשב, DH2 את השני וכן האלה):

$$DH1 = DH(IK_A, SPK_B)$$

מורכב ממפתח הזהות הפרטי של אבי (יוצר הסשן) וה-Signed PreKey של בכור (הצד המקבל).

$$DH2 = DH(EK_A, IK_B)$$

מורכב מהמפתח הפרטי הזמני של אבי (אשר נוצר עבור סשן זה בלבד) ומה-Public Identity Key של בכור.

$$DH3 = DH(EK_A, SPK_B)$$

מורכב מהמפתח הפרטי הזמני של אבי, ומה-Signed PreKey של בכור.

$$DH4 = DH(EK_A, OPK_B)$$

מורכב מהמפתח הפרטי הזמני של אבי, ומה-One-Time-PreKey של בכור.

- במצבים מסוימים, בהם נגמרה אספקת ה-One-Time PreKeys על השרת (לדוגמא במצב בו הצד המקבל לא התחבר לאפליקציה תקופה ולא העלה מפתחות חדשים לשרת) החישוב של DH4 לא יבוצע, וה-X3DH יבוצע על סמך 3 ערכים, ולא 4.

נשרשר את ערכי ה-DH שקיבלנו, ולבסוף נעביר אותם דרך פונקציית KDF:

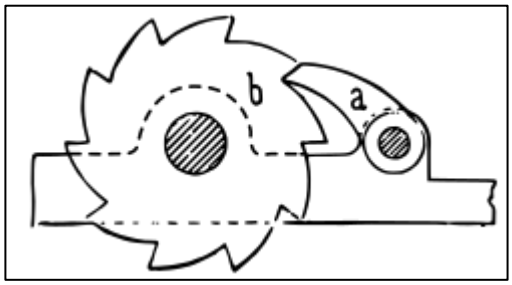
$$SK4 = KDF(DH1 \,||\, DH2 \,||\, DH3 \,||\, DH)$$

התוצאה שנקבל מכונה SK (Session Key) והיא משמשת כקלט הראשוני עבור אלגוריתם Double Ratchet. כפי שציינו, X3DH הוא הדרך של שני משתמשים לייצר ביניהם סשן ראשוני (ומכאן השם של הפלט שלו, Session Key) והשימוש בו נעשה בפעם הראשונה שהם מתחילים לתקשר. לאחר שסיימו לבצע X3DH בהצלחה, האחריות עוברת ל-Double Ratchet.

Double Ratchet

האלגוריתם מתבסס על שני מנגנונים דמויי גלגל שיניים במהותם (או ליתר דיוק מחגר משנן- מנגנון המאפשר תנועה סיבובית או קווית של חלק מכונה בכיוון אחד בלבד ומונע את תנועתו בכיוון ההפוך), המשמשים לייצור והחלפה של מפתחות.

אך לפני שנבין את מנגנונים אלה, נבין כיצד האלגוריתם עובד ב-High Level.



כאשר שני משתמשים מתקשרים מעל סיגנל ומבצעים X3DH כדי להתחיל סשן, הם ישתמשו ב-Session Key שיצרו, כדי בסופו של דבר לגזור מפתחות סימטריים לטובת הצפנה ופענוח של הודעות.

אך כפי שכבר הבנו, השימוש במפתחות אסימטריים קבועים לא בטוח ולכן בסיגנל המשתמשים יחדשו את המפתחות האסימטריים שלהם אחת לכמה הודעות.

בפועל, חידוש המפתחות מתבצע בכל פעם שצד מסוים שולח הודעה ראשונה אחרי שקיבל כבר הודעה מהצד השני:

הצד שמחדש את המפתחות שלו יצרף את המפתח הפומבי החדש להודעה ששלח, כך שהצד השני ישתמש בו כדי לבצע חישוב DH, שבסופו של דבר יאפשר לו לפענח את תוכן ההודעה. כאשר המשתמש שזה עתה קיבל את ההודעה ישיב למשתמש ששלח לו אותה, גם הוא יחדש את המפתחות באותה צורה שזה עתה תיארנו.

KDF chains

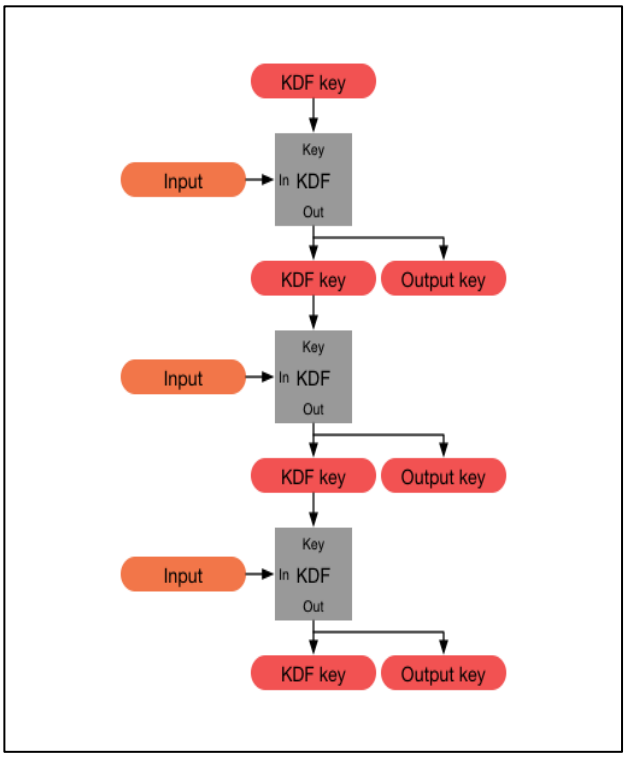
כדי להבין את האלגוריתם בצורה מיטבית, עלינו תחילה להבין קונספט ליבתי עליו מתבסס האלגוריתם המכונה KDF Chain, או בעברית שרשרת KDF.

בגדול, שרשרת KDF היא מנגנון קריפטוגרפי שבו פונקציית KDF בעלת מספר פלטים משורשת לעצמה בצורה הבאה:

הפלט הראשון הוא KDF key / Chain key, מפתח סודי המשתנה בכל איטרציה של השרשרת, חיוני ליצירת השלב הבא בשרשרת (למעשה ישמש כקלט בשימוש הבא בפונקציית ה-KDF).

הפלט השני הוא Output key, שהוא המפתח המשמש להצפנה כללית.

שרשרת שכזו מאפשרת לנו לייצר מפתחות הצפנה (Output keys) אותם ניתן לייצר אך ורק בהינתן ה- KDF key, אשר נשאר סודי לאורך התהליך.



התכונות המרכזיות של שרשרת KDF הן:

- גמישות - למי שאין ידע על מפתח ה-KDF, לא יוכל להבדיל בין הפלטים השונים של פונקציית ה-KDF, ולמעשה לא יכול להבדיל בין ה-KDF Key, ל-Output key.
- סודיות קדימה - גורם שהצליח להשיג את ה-KDF key הנוכחי, לא יוכל לחשב מפתחות קודמים "התאוששות" **(Break-in recovery)** - המנגנון יודע לרפא את עצמו במידה וה-KDF Key נחשף.

על הנייר, נראה שאם גורם שלישי משיג את ה-KDF Key, שום דבר לא ימנע ממנו לחשב את המשך השרשרת, אך אלגוריתם Double Ratchet פותר בעיה זו בעזרת מנגנון עליו נלמד כעת.

(בעבר, פרוטוקול סיגנל היה מכונה "**Axolotol**" על שם סלמנדרה מקסיקנית שיכולה "לחדש" חלקים מגופה - https://signal.org/blog/signal-inside-and-out/)

Diffie-Hellman ratchet

מטרת מנגנון זה היא חידוש והחלפה מתמשכים של מפתחות אסימטריים עליהם מתבססים המפתחות הסימטריים המשמשים להצפנה ופענוח של הודעות.

הכל מתחיל אחרי X3DH, שבפועל עושה קצת יותר מלייצר Session Key.

בפועל, הפלט של X3DH משמש כ-Seed עבור Double Ratchet, ובאופן ספציפי כקלט הראשון בשרשרת ה-KDF (ה-KDF key הראשון) עליה המנגנון Diffie-Hellman ratchet מתבסס.

בקונטקסט של מנגנון זה, ה-**KDF key** בשרשרת ה-**KDF** מכונה **Root Key**.

כאשר מתרחשת החלפת מפתחות מכיוון אחד המשתמשים, מתבצעת יצירה של חוליה חדשה בשרשרת וזאת על ידי שימוש בפונקצית KDF המקבלת 2 ערכים ומחזירה 2 ערכים:

(RK_new, CK_sending) = **KDF_RK**(RK_old, DH_output)

- **RK_old** - ה-Root Key הנוכחי (Session Key הוא למעשה Root Key לכל דבר ועניין, ואף אפשר להתייחס אליו כ-RK_0, שמו המיוחד נובע מהיותו הראשון בסשן).
- **DH_output** - תוצאת ה-DH שהתקבלה על בסיס המפתחות החדשים.
- **RK_new** - ה-Root Key החדש.
- **CK_sending** - ה-**Chain Key** הנוכחי, שאת משמעותו נבין מיד.

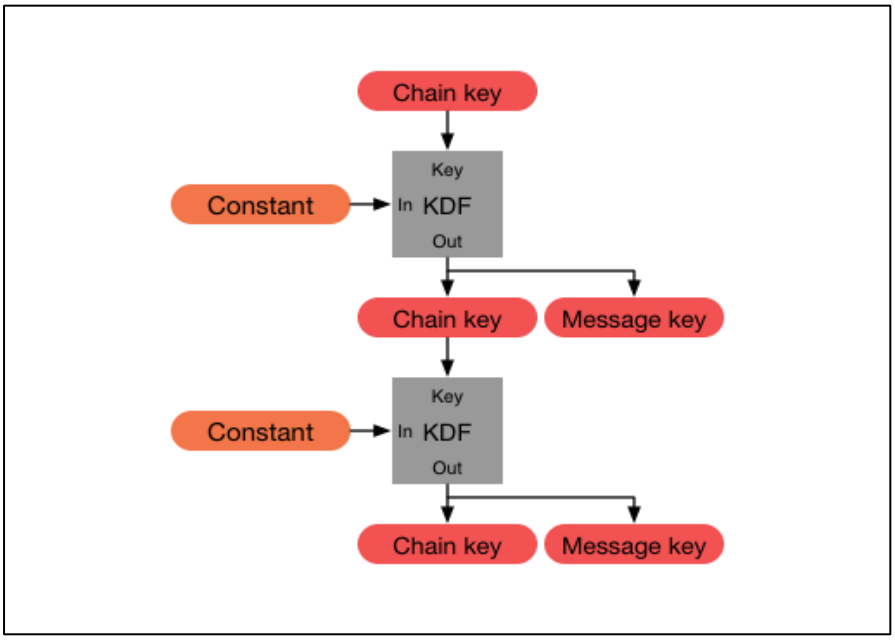
כאשר נקבל את ערך ה-**Root Key** החדש, נמחק באופן מידי את ערך ה-**Root Key** הישן, וזאת על מנת למנוע מגורם שלישי לחדש חוליות ישנות יותר בשרשרת.

Symmetric-key ratchet

המנגנון השני באלגוריתם, המכונה Symmetric-key ratchet, הוא המנגנון האחראי לייצר מפתחות סימטריים לצורך הצפנה ופענוח של הודעות.

בשונה ממנגנון ההצפנה עליו דיברנו במאמר הקודם, בו משתמש מגריל מפתח AES כדי להצפין הודעה, מפתחות ההצפנה הסימטריים בסיגנל המשמשים להצפנה ופענוח של הודעות מיוצרים בעזרת שרשרת KDF, בה ה-**KDF Key** הוא למעשה ה-**Chain Key** שמיוצר על ידי ה-**Diffie Hellman ratchet**. שרשרת ה-KDF עליה מתבסס המנגנון פשוטה מאד:

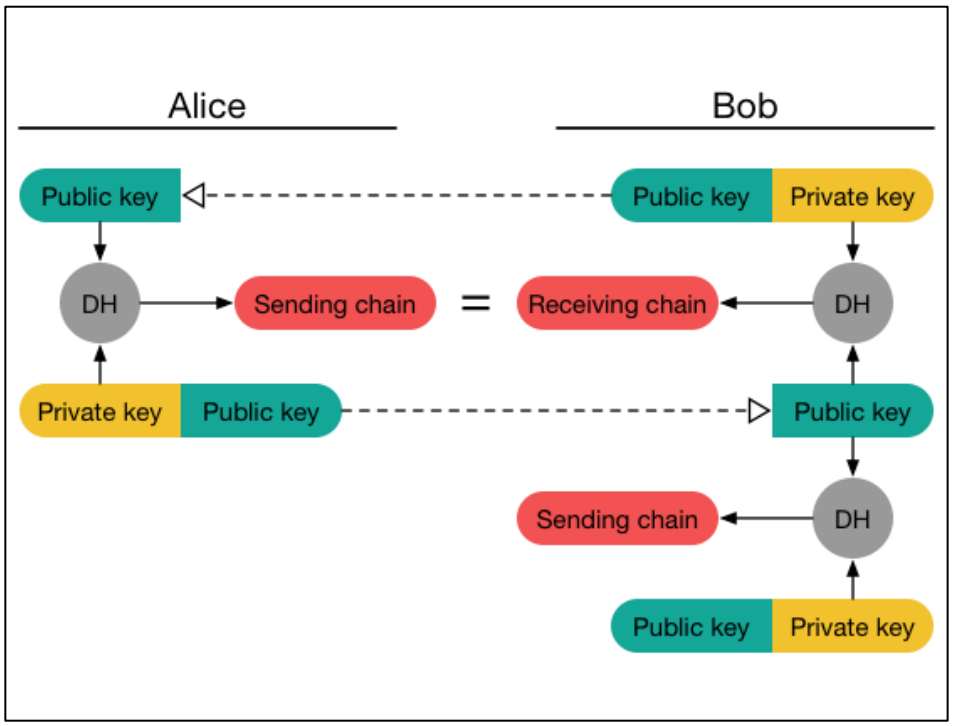
הקלט הראשוני הוא ה-Chain Key שמיוצר באיטרציה של Diffie Hellman ratchet, ובכל איטרציה של השרשרת, נוצר **Chain Key** חדש, ו-**Message Key** המשמש להצפנה / פענוח של הודעה בודדת:



.

מאחר ושני הצדדים בסשן משתמשים באותו Chain Key התחלתי כדי לייצר את שרשרת ה-KDF, הם גם מייצרים Message Keys זהים בכל איטרציה, דבר המאפשר להם לחלוק מפתחות לצורך הצפנה ופענוח, מבלי לשלוח את המפתחות הללו אחד לשני. בפועל, כאשר צד A מחדש את המפתחות שלו ומקדם את מנגנון ה-Diffie Hellman ratchet בשלב, הוא מייצר שרשרת KDF המכונה Receiving chain המשמשת אותו לפענוח של הודעות. לעומת זאת, כאשר צד B יגלה שבוצעה התקדמות בשרשרת, הוא יצור שרשרת KDF זהה לזו שיצר משתמש A, המכונה Sending chain, רק שזו תשמש אותו להצפנה של הודעות.

(בפועל, מאחר והמפתחות המיוצרים על ידי השרשראות זהים, המשתמשים יכולים להשתמש ב- Sending & Receiving chains גם להצפנה וגם לפענוח של הודעות!)



שרשראות אלה ישמשו את הצדדים כדי להצפין ולפענח הודעות, עד אשר אחד הצדדים יחדש את המפתחות שלו, מה שיגרום להתקדמות שרשרת ה-Diffie Hellman וכתוצאה מכך ליצירה של Sending & Reciving chain חדשים.

בשונה ממנגנון ההצפנה שהכרנו במאמר הקודם, בו משתמש מגריל מפתח AES אקראי ומשתמש בו כדי להצפין הודעה, בסיגנל המשתמשים מייצרים מפתחות על סמך סוד משותף מתחדש ומעדכנים אחד את השני איזה מפתח הצפין איזה הודעה על סמך האינדקס של המפתח בשרשרת ה-KDF הנוכחית.

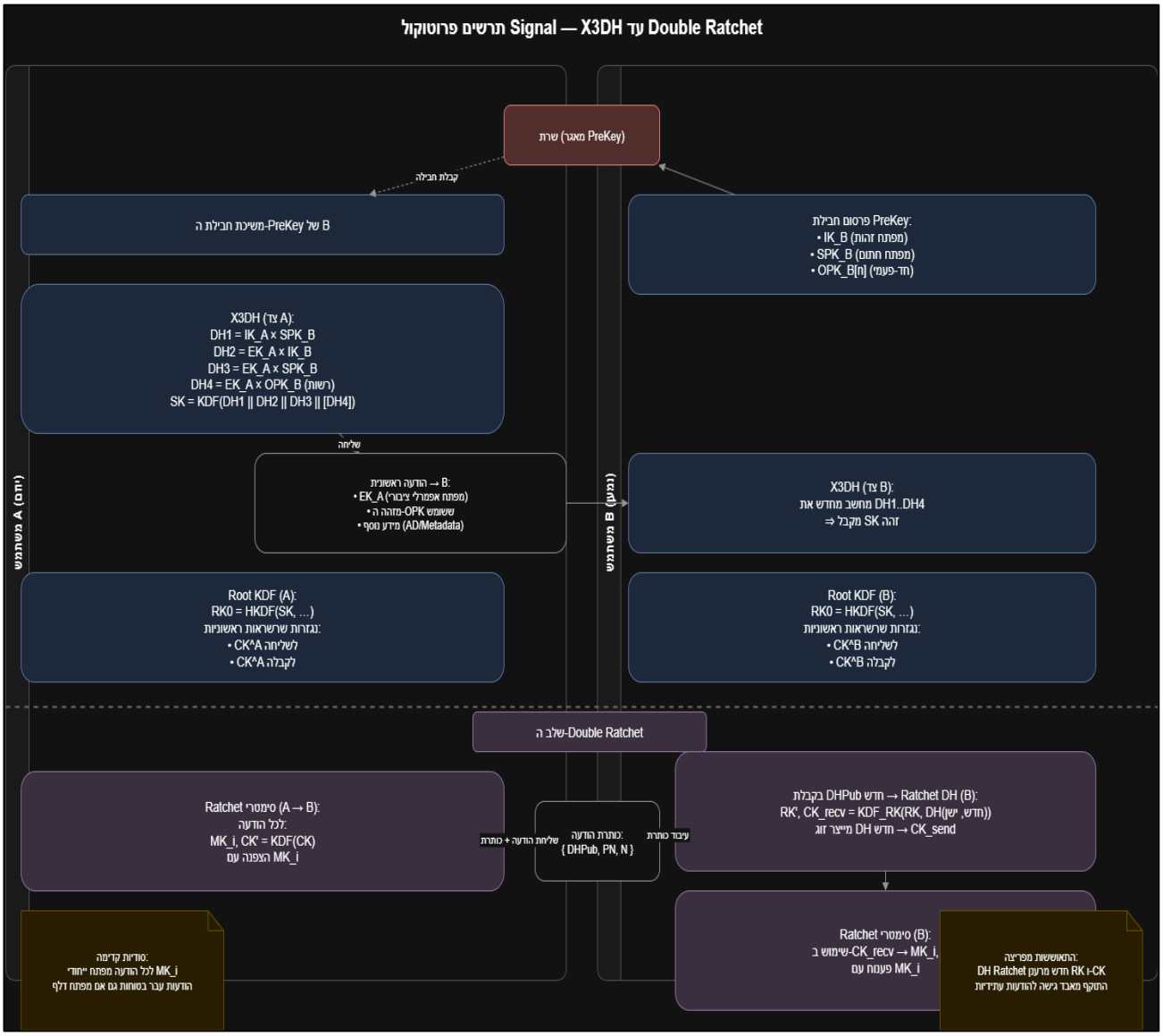
מנגנון זה מוודא שמפתח בודד משמש להצפנה ופענוח של הודעה בודדות, ואף מוודא שלא ניתן להשיג גישה למפתחות ישנים יותר (גם כי הם נמחקים, וגם כי הם מחושבים על בסיס פונקציית KDF אותה לא ניתן להפוך). החיסרון היחיד במנגנון זה הוא שבהינתן מפתח X ו-Root Key, נוכל לחשב את המפתח X+1 בשרשרת. אך בעיה זו נפתרת על ידי Diffie-Hellman ratchet, אשר מוודא שה-Root Key הנוכחי מתחדש לעיתים תכופות, כך שגם אם הגיע לידיים הלא הנכונות, לאחר מספר הודעות הוא כבר לא יהיה רלוונטי.

לאחר שהכרנו את המנגנונים העיקריים המרכיבים את הפרוטוקול, בואו נחזור עליהם:

X3DH- פרוטוקול החלפת מפתחות המאפשר לשני משתמשים לייצר סשן אחד עם השני על סמך מפתחות הצפנה המאוחסנים בשרתים של סיגנל, דבר המאפשר לשני המשתמשים להתחיל סשן גם אם אחד מהם אינו מחובר. הפרוטוקול משתמש במספר מפתחות הצפנה שונים ואף מבצע בדיקת זהות (Identity Keys, Signed PreKey) כדי לייצר מפתח הנקרא Session Key אשר משמש כקלט הראשוני של Double Ratchet.

Double Ratchet - אלגוריתם המאפשר לשני משתמשים לייצר מפתחות הצפנה סימטריים להצפנה ופענוח של הודעות, תוך חידוש מתמשך של המפתחות האסימטריים עליהם הם מתבססים. האלגוריתם מורכב מ-2 שרשראות KDF, האחת (Diffie-Helman ratchet) אחראית על חידוש מפתחות אסימטריים, והשנייה (Symmetric key ratchet) אחראית על יצירת מפתחות סימטריים על בסיס מפתח סודי המיוצר על ידי השרשרת הראשונה.

ניתן להסתכל על השילוב בין השתיים כלולאת for מקוננת, כך שעל כל איטרציה של Diffie-Helman ratchet, תתבצע אחת או יותר איטרציות של Symmetric key ratchet.



כעת, לאחר שהכרנו את הפרוטוקול קצת יותר, נחזור לבעיות שהצפנו בתחילת המאמר ונבין איך סיגנל פותרת אותם:

- מפתחות אסימטריים סטטיים - בעזרת Double Ratchet, סיגנל מאפשרת ל-2 משתמשים לחדש מפתחות אסימטריים לעיתים תכופות, גם אם אחד המשתמשים אינו מחובר לשירות באותם רגעים.
- שימוש במפתחות אסימטריים עבור יותר מסשן אחד - סיגנל פותרת בעיה זו בכך שהיא מאפשרת למשתמש להעלות מפתחות הצפנה חד פעמיים לשירות, כאלה המאפשרים למשתמשים לייצר ביניהם סשן בהתבסס על פרמטרים קיפטוגרפיים יחודיים עבור אותו סשן, לדוגמא, על ידי One-Time PreKey - מפתח הצפנה בו נעשה שימוש ליצירת סשן אחד בלבד.

מקורות מידע וקריאה נוספת

- תיעוד מעמיק של פרוטוקול סיגנל - https://signal.org/docs/
- הספרייה הרשמית המממשת את הפרוטוקול - https://github.com/signalapp/libsignal

סיכום

במאמר זה אתגרנו את מנגנון ההצפנה שעליו דנו במאמר הקודם במטרה להבין את המורכבות הניכרת ביצירת פרוטוקול הצפנה אשר עומד בסטנדרטים מודרניים.

הכרנו את ההיסטוריה של אפליקצית סיגנל ואת ליבת פרוטוקול ההצפנה שלה (X3DH, Double Ratchet), אשר פותר בצורה אלגנטית את הבעיות שהוצגו בתחילת המאמר, עובדה אשר עודדה מספר רב של אפליקציות לאמץ את הפרוטוקול לחיקיהן.

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.

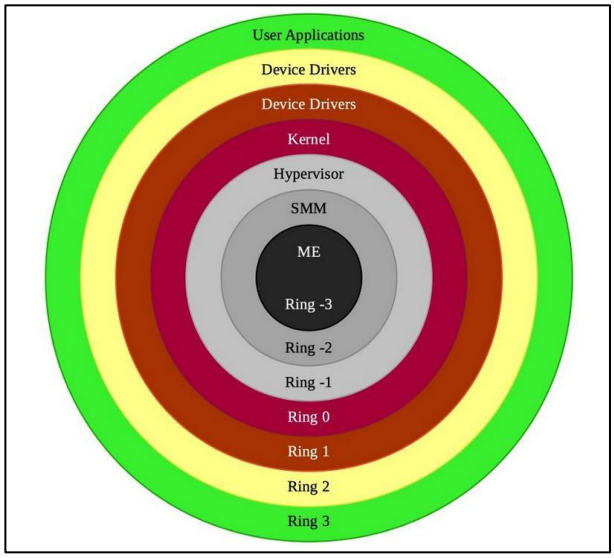
המצבים שאליהם אני מתייחס הם real mode ו-protected/long mode:

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.

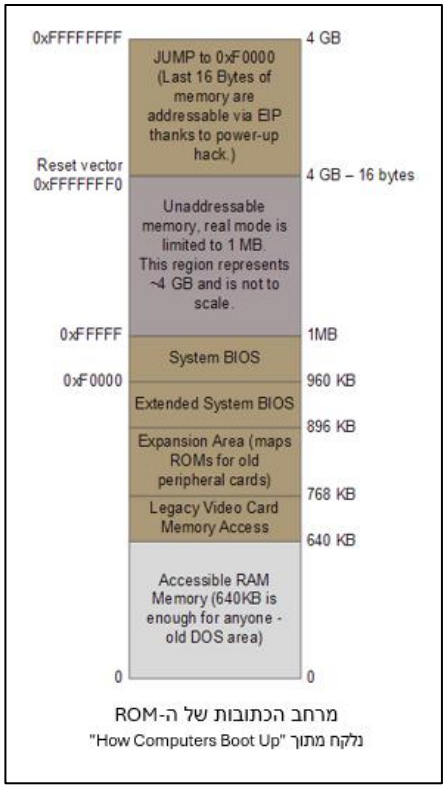
ניתן להיכנס למצב SMM על ידי קריאה ל-SMI - System management interrupt. ניתן לקרוא ל-SMI בעזרת פין פיזי של ה-CPU, כלומר אחד מהפינים הפיזיים של CPU מעיד על שליחה של SMI, וברגע שהוא נשלח ניכנס למצב SMM. ברגע שה-CPU נמצא במצב SMM כל interrupt אחר מבוטל. בכדי לצאת ממצב SMM

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.

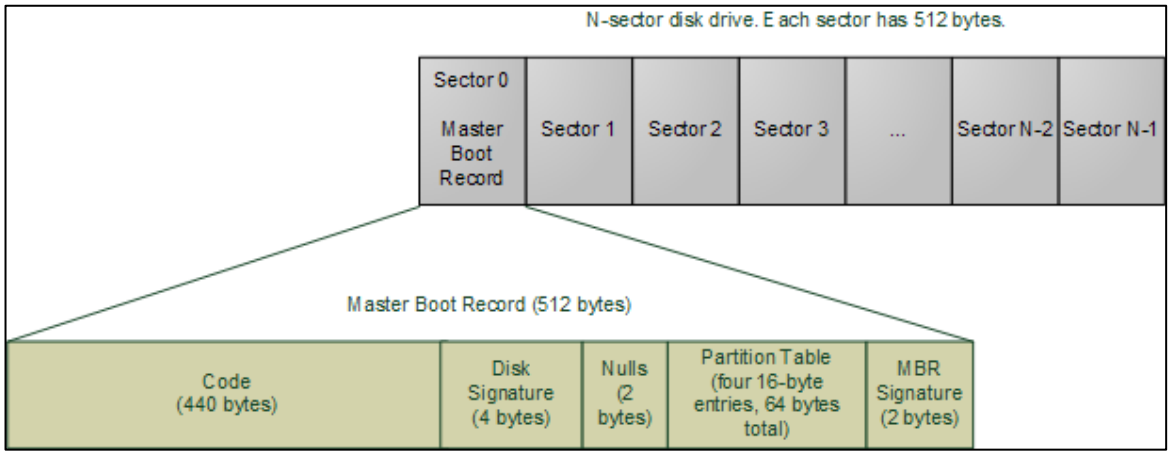
POST

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).



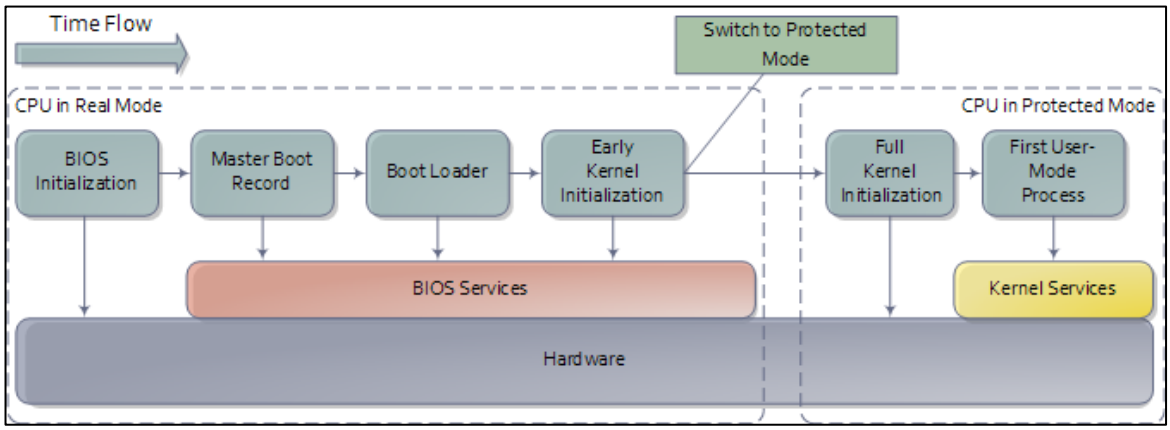
["How Computers Boot Up" נלקח מתוך]

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.

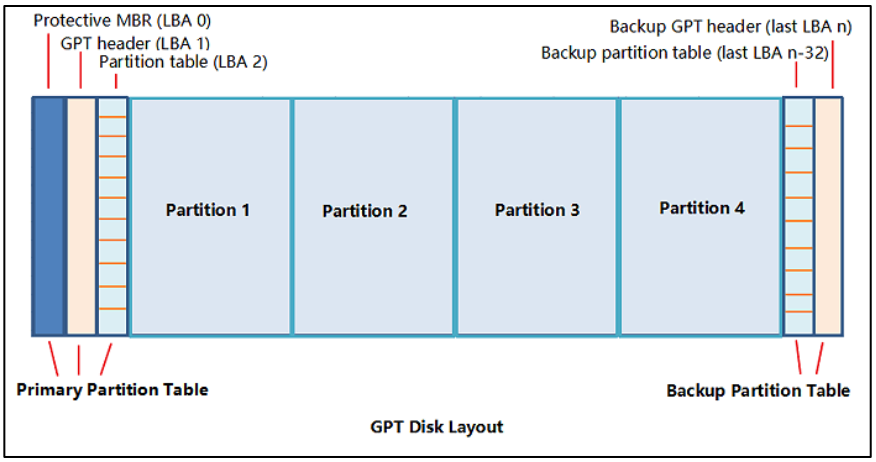
תהליך ה-BOOT עם UEFI הוא מעט שונה מאשר עם BIOS וגם יותר מורכב, ההבדל המשמעותי בין השניים הוא ש-UEFI לא צריך את ה-MBR או את ה-VBR. אלא הוא משתמש בקוד שמוטמע בתוך ה-NVRAM ו-GUID partition table (GPT).

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.

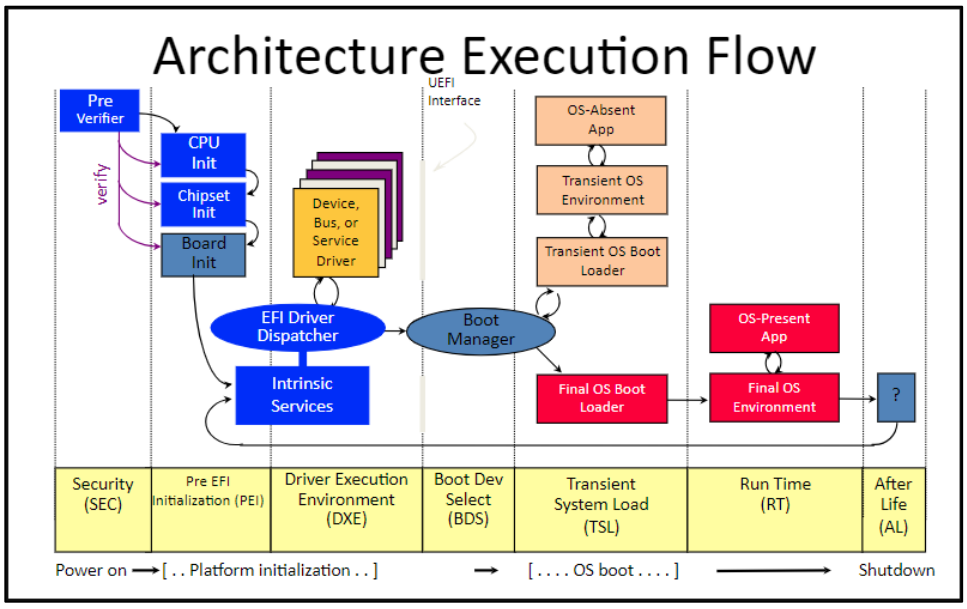
החלקים המרכזיים ב-GPT שחשוב להכיר:

- 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.

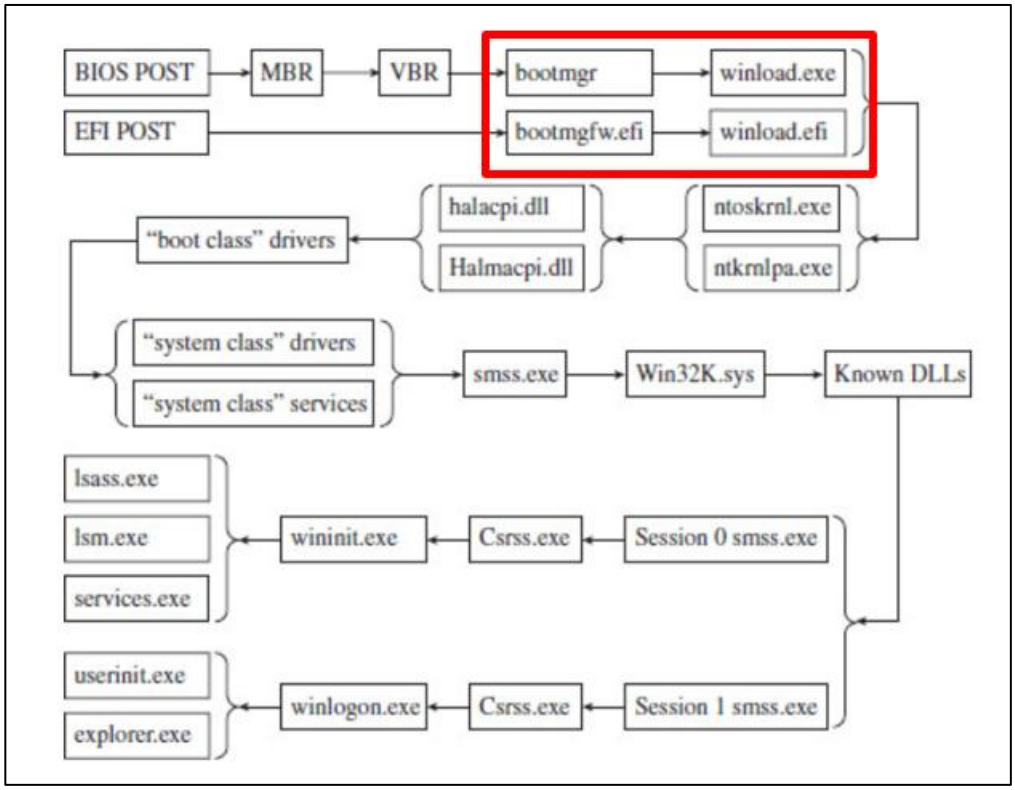
RT (Runtime)

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

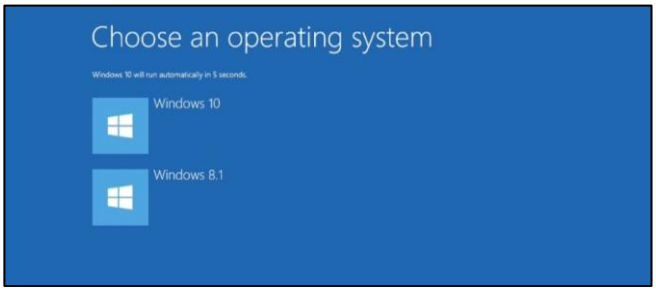
בכדי לתמוך גם ב-BIOS boot וגם ב-UEFI boot, windows מספקת ארכיטקטורת boot שמתאימה לשני הסוגים - בעיקר בכדי שיהיה ניתן להשתמש במערכת ההפעלה עם כמה שיותר סוגי חומרה. ב-BIOS לאחר הרצה של קוד ה-VBR, נגיע לקובץ bootmgr.exe. ב-UEFI לאחר שנפתח את ה-ESP partition ירוץ הקובץ bootmgfw.efi שנמצא בנתיב \EFI\Microsoft\Boot\bootmgfw.efi.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

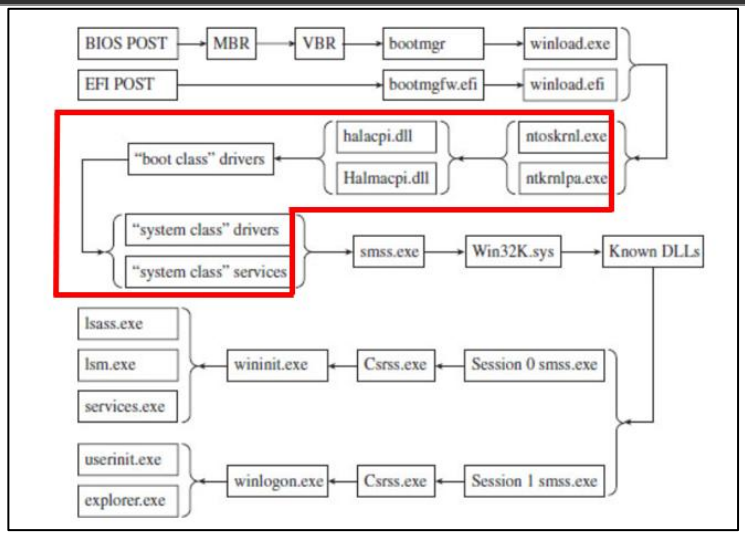
במקרה בו נשאיר את המחשב דלוק אבל לא ניגע בו הרבה זמן, המסך שלו "יהפוך לשחור" וניכנס למצב של hibernation - אפשר לחשוב על זה כמעין תנומת חורף. בגלל שלא באמת כיבינו את המחשב, כאשר נרצה לחזור לאותו מצב שבו עזבנו, תהליך ה-boot לא יהיה אותו הדבר. במצב הזה תהיה הגדרה ב-BCD על עליית מערכת מ-hibernation state, 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.

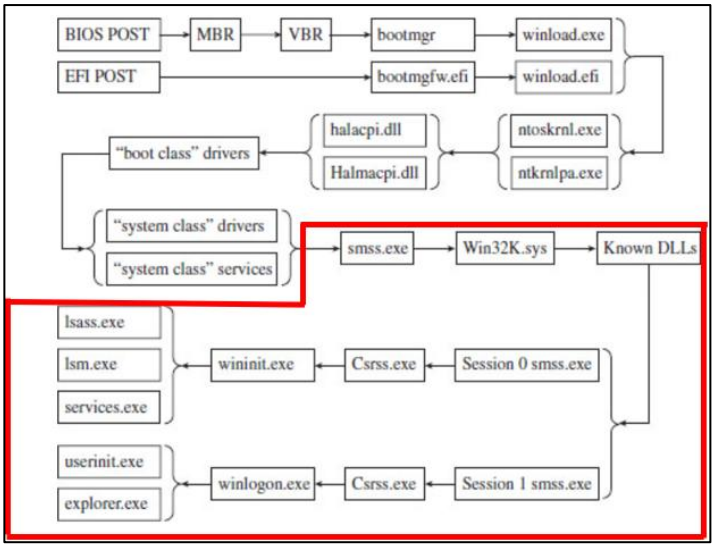
ה-windows boot loader הוא באמת הגורם שמעלה את מערכת ההפעלה במלואה, הוא מתחיל בלטעון את ה-registry hive - HKLM\SYSTEM שנמצא ראשית תחת system32. לאחר מכן הוא מוודא את החתימה של עצמו (nt5.cat) שנמצא תחת:

```
%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

smss.exe - session manager subsystem, 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".

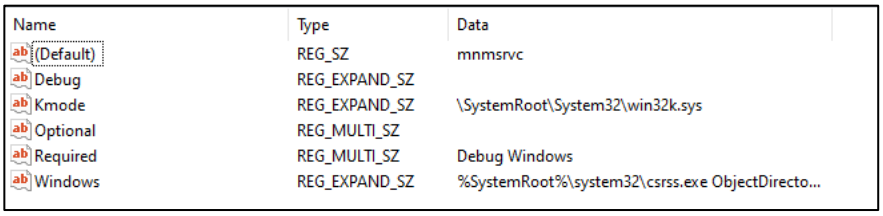
לכן smss יכול לבצע פעולות שלרוב התהליכים אין את היכולת לבצע, כמו יצירת security tokens - לכל משתמש יש token שמזוהה איתו, באמצעותו הוא מזדהה כאותו משתמש. smss לא משתמש ב-WinAPI בגלל שה-windows subsystem לא רץ כאשר smss רץ לראשונה, בעצם אחד התפקידים של smss הוא להריץ את ה-windows subsystem.

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.

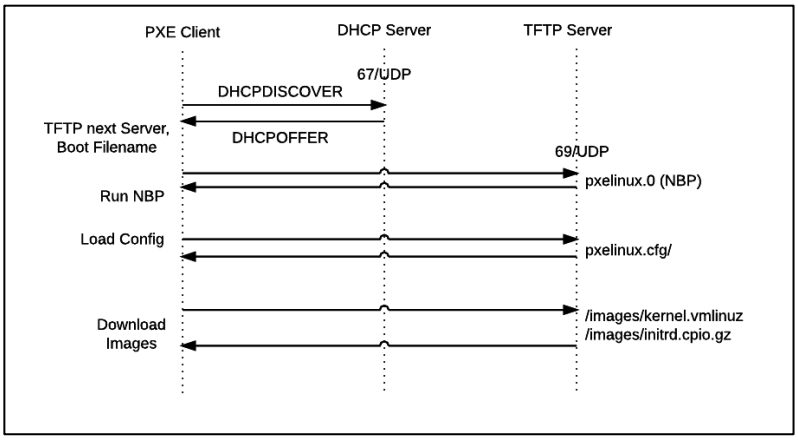
Winlogon.exe שרץ ב-session 1, אחראי לטפל ב-user logons. הוא מריץ את ה-logon interface - logonui.exe שמציג את מסך ההתחברות למשתמש. קובץ הרצה זה מעביר את ה-credentials שהמשתמש הכניס ל-lsass, אם תהליך האימות היה מוצלח, יורצו קבצי ההרצה הרלוונטיים תחת ה-values: UserInit & Shell שנמצאים תחת ה-key:

```
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.

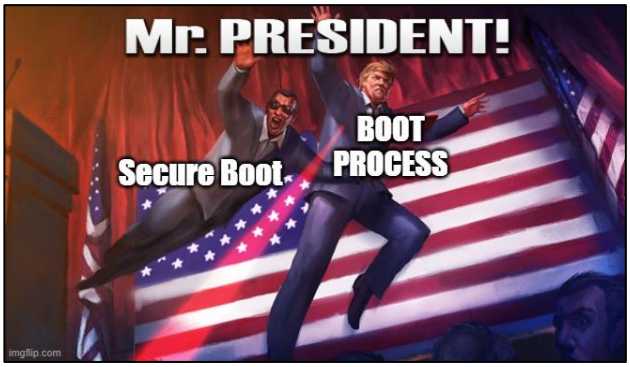
Network Boot (PXE)

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.



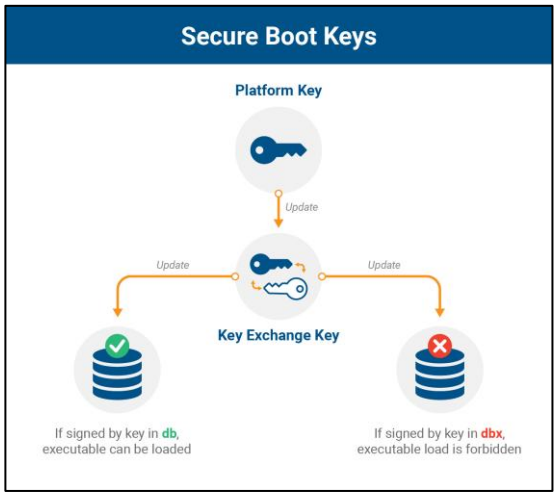
Secure Boot

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

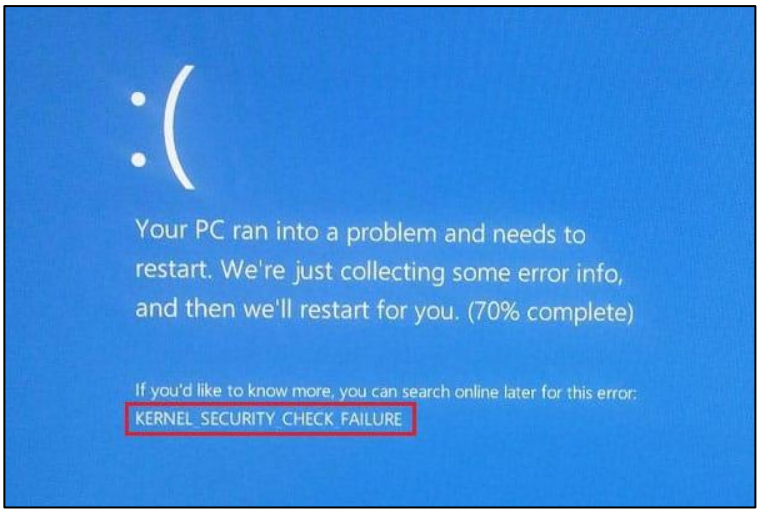
זה הוא פתרון הגנה מבית intel שנועד בכדי לוודא את תקינות ה-firmware שעולה. בהרבה מקרים ה- root of trust נקבע על ה-firmware מבלי לבדוק את החתימה שלו מראש, בגלל מחשבה שאולי זה הגורם הכי בטוח במערכת שלנו, אבל עם הזמן תוקפים התקדמו והצליחו גם להשפיע על החלקים הנמוכים ביותר של המערכת שלנו.

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.

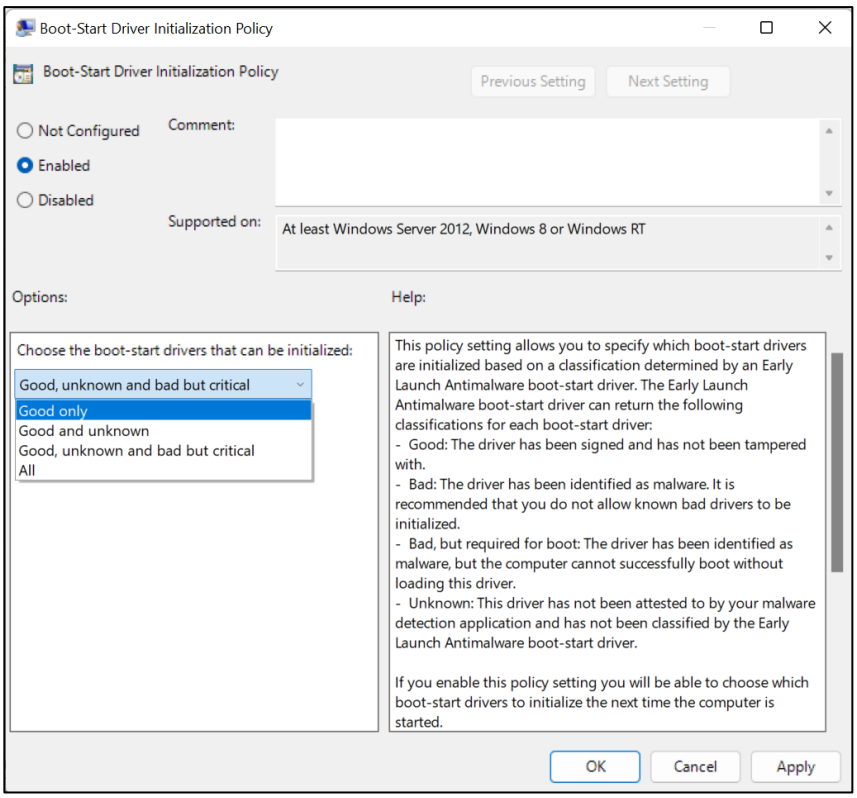
ELAM

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
```

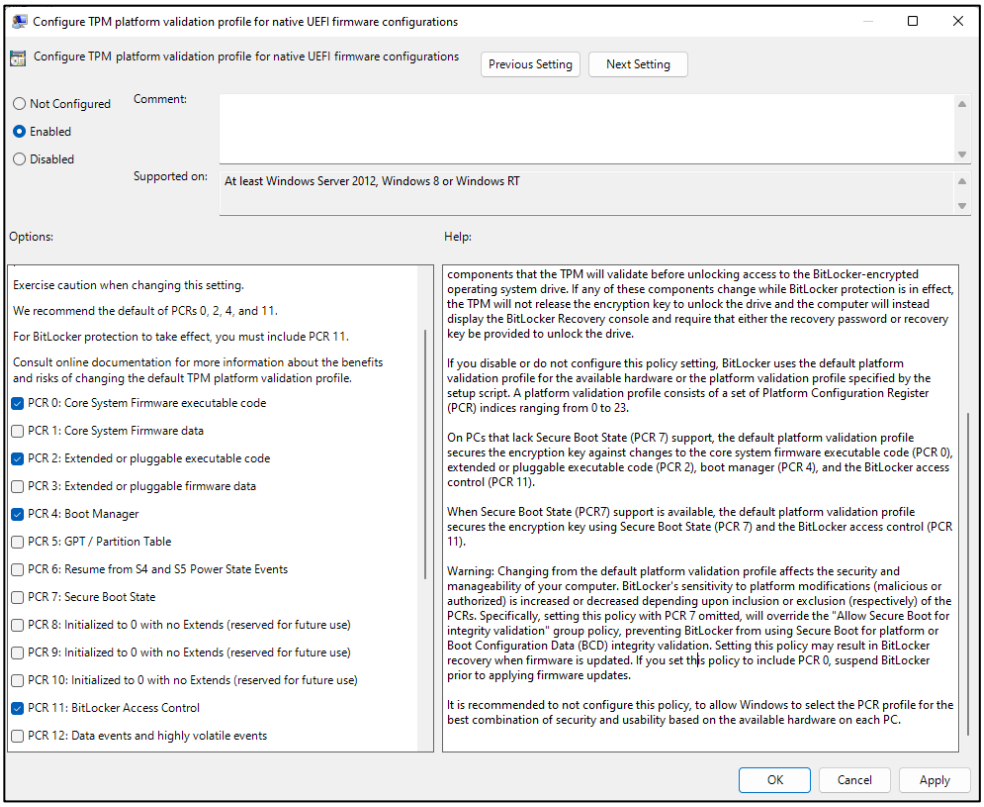
PCR Index	PCR Usage
0	SRTM, BIOS, Host Platform Extensions, Embedded Option ROMs and PI Drivers
1	Host Platform Configuration
2	UEFI driver and application Code
3	UEFI driver and application Configuration and Data
4	UEFI Boot Manager Code (usually the MBR) and Boot Attempts
5	Boot Manager Code Configuration 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.

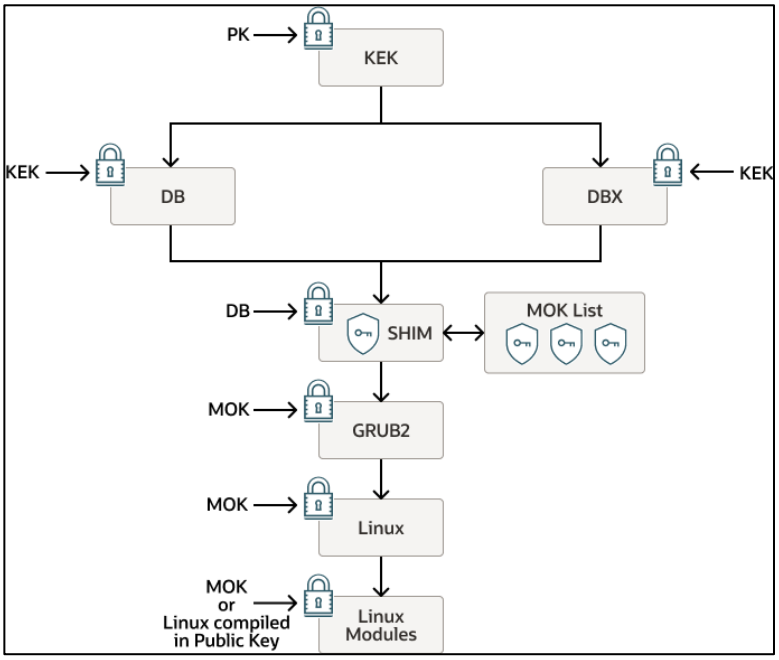


צירפתי גם תמונה שמראה כיצד באמת אפשר לקנפג את bitlocker, כלומר לפי אילו תנאים הוא משחרר את מפתח ההצפנה.

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:

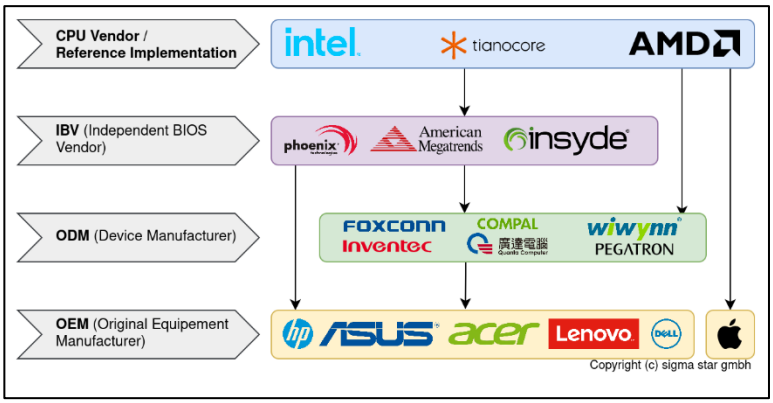
אבל אי פעם חשבתם איך בכלל מופיעה התמונה הזו על המסך שלנו? כי אנחנו עוד הרבה לפני מערכת ההפעלה. אם ניזכר בשלבי העלייה של UEFI וספציפית בשלב ה-DXE, נראה כי נטענים הרבה driver-ים שמאפשרים לנו את כל היכולות שיש ל-UEFI. אחד מהם הוא parser לתמונות bmp/jpg המאפשר להציגן על המסך.

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.

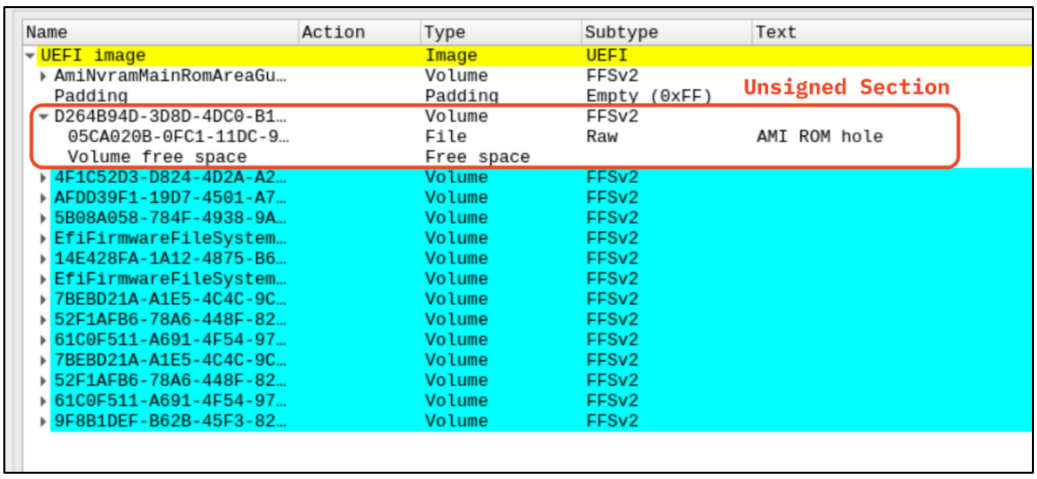
לכן ההשפעה היא הרבה יותר רחבה, כי גם אם נמצאה החולשה במחשב Lenovo אחד, היא יכולה להיות באותה המידה גם במחשבים של acer. זו הבעיה העיקרית בחולשות ב-firmware עצמן, גם היצרניות הגדולות בסופו של דבר משתמשות בקוד של חברה אחרת שמספקת שירותים לעוד עשרות חברות! בסופו של דבר binarly מצאו סה"כ 30 חולשות שונות על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



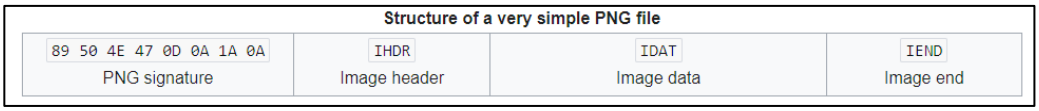
[דיאגרמה שמסבירה על חלוקת האחריות בפיתוח המחשב, נלקח מ-"Securely booting x86 using Heads - sigma star"]

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]

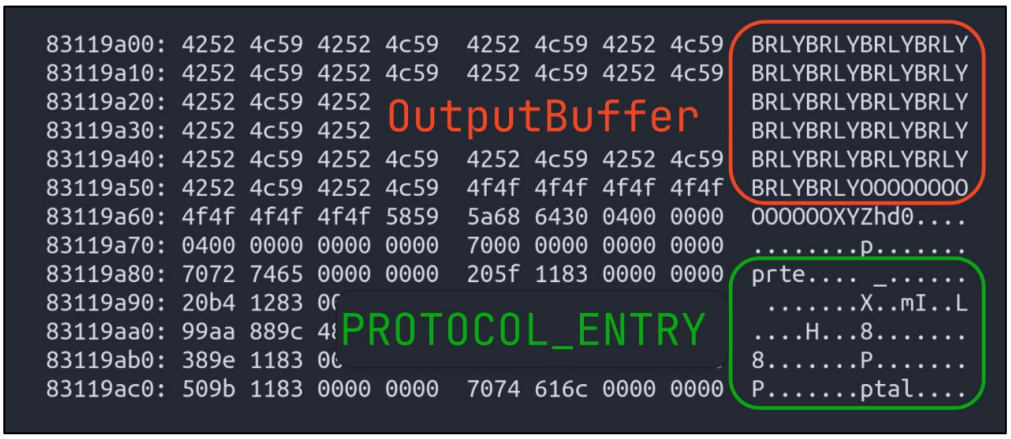
שני החלקים שהכי מעניינים אותנו הם IHDR שמכיל מידע על התמונה כגון Height and Width, ו-IDAT שמכיל את המידע עצמו של התמונה.

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).

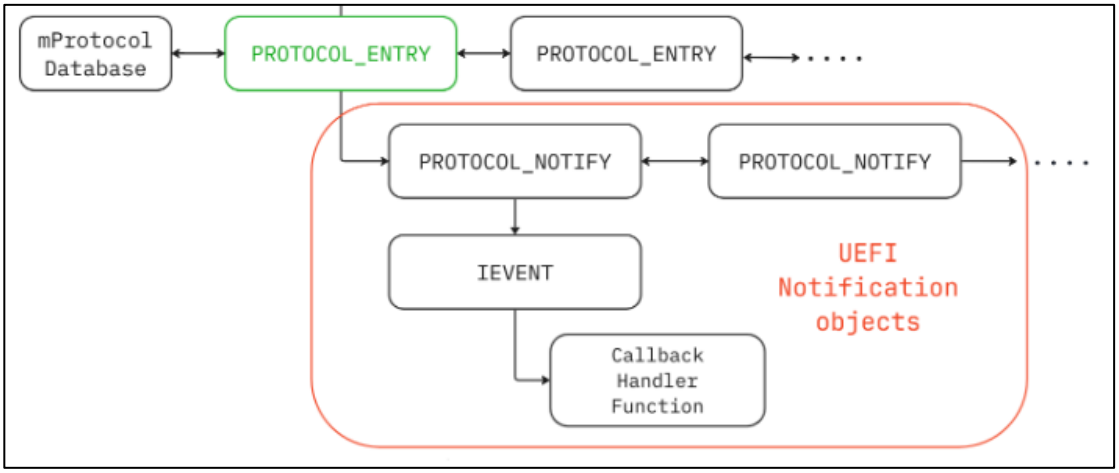
כעת עולה השאלה איזה מידע אנחנו דורסים באמצעות ה-heap overflow? לחוקרים ב-Binarly לא היו שום יכולות debugging על המכשיר, אז בכדי לראות את ה-state של ה-heap, ניסו לשמר את ה-state שלו לאחרי שמערכת ההפעלה עלתה. כלומר, גם לאחר ש-UEFI סיים את כל הפעולות שלו ומערכת ההפעלה עלתה לגמרי יהיה ניתן לחפש בזיכרון מידע 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.

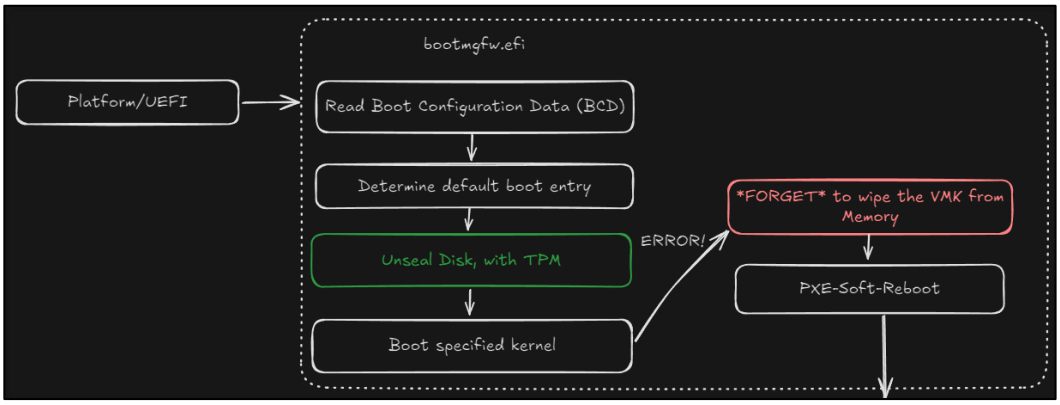
זהו! הצלחנו באמצעות תמונת PNG אחת להשתלט לגמרי על המערכת להשיג persistence שקשה מאוד לגלות. היה לי חשוב להציג את המתקפה הזו בכדי להראות כמה חזק תהליך ה-boot וגם כיצד ניתן להשתמש בדברים הכי טריביאליים בצורה זדונית. לקישור לסרטון שמראה את מהלך התקיפה לחצו פה.

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.

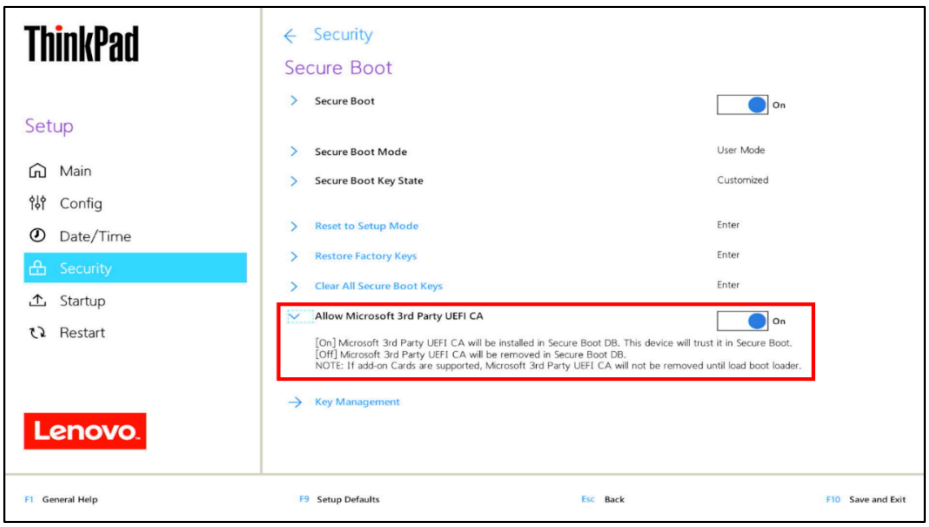


[נלקח מ- " Windows BitLocker -- Screwed without a Screwdriver "]

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

איך בונים פרימטר בענן

מאת יהב פסטינגר

הקדמה

כיום גופים רבים בוחרים שלא להוציא את המידע הרגיש שלהם למרחב הענן הציבורי מכיוון שבניגוד לרשתות הפנימיות שבהם הפרימטר (האיזור שחוצץ בין הרשת הפנימית לרשת הציבורית) נתון לפיקוח והגבלות רבות, בענן הציבורי הפרימטר אינו מוגדר היטב והסיכוי לטעות קונפיגורציה שעלולה לשבור את הפרימטר גבוהה בהרבה.

ספקיות הענן הבינו את גודל הבעיה וכדי לתת מענה לצורך הזה הכריזה AWS בשנת 2022 על הרעיון של Data Perimeter. הרעיון פשוט - ליצור איזור (בהמשך נבין איך נראה האיזור הזה) שבו יאוחסן המידע הרגיש. המטרה שלנו להחיל את כלל הפוליסות על האיזור הזה בשביל למנוע גישה של תוקף למידע. באמצעות סט של חוקים שמאפשרים רמת גרנולריות גבוהה ביותר נוכל למנוע תרחישים מורכבים שתוקפים עושים בהם שימוש ביום-יום.

החוזק המשמעותי של הקונספט הוא שבניגוד להרבה מכלי ההגנה המוצעים כיום בשוק שעובדים במוד של גילוי ולאו דווקא מניעה, הרעיון פה הוא למנוע את הגישה של התוקף למידע בזמן אמת כך שגם אם הצליח להגיע ליכולות משמעותיות ברשת שלנו, עדיין נוכל למנוע ממנו להדליף את המידע החוצה.

אז מה זה AWS Data Perimeter

הרעיון מדבר על כמה רבדים שונים מהם מורכב הארגון:

1. הפן הרשתי - הרשת ממנה אני עובד ביומיום (זו יכולה להיות רשת פרטית או רשת עם כתובת IP פומבית).
2. הפן הזהותי - מיהי הזהות שמבצעת את הפעולה. לדוגמא, משתמש SSO שמצומד ל-Role מסוים או IAM User. לכל זהות מצומדת סט של הרשאות שמגדיר את הפעולה אותן היא יכולה לבצע.
3. מה הזהות שמצומדת לי. לכל זהות מצומד סט של הרשאות שמגדיר את הפעולות אותן אני יכול לבצע.
4. הפן המשאבי - הענן מורכב מאוסף של שירותים שבכל שירות יש מבחר של משאבים (resources). לדוגמא, תחת השירות S3 שמהווה את שירות האחסון המרכזי ב-AWS ישנו משאב של Bucket שמייצג את יחידת האחסון הבסיסית. אל ה-Bucket נוכל לכתוב קבצים באמצעות סט של פעולות (APIs).

אלה הקומפננטות הבסיסיות שעליהן נרצה להחיל את סט החוקים שלנו.

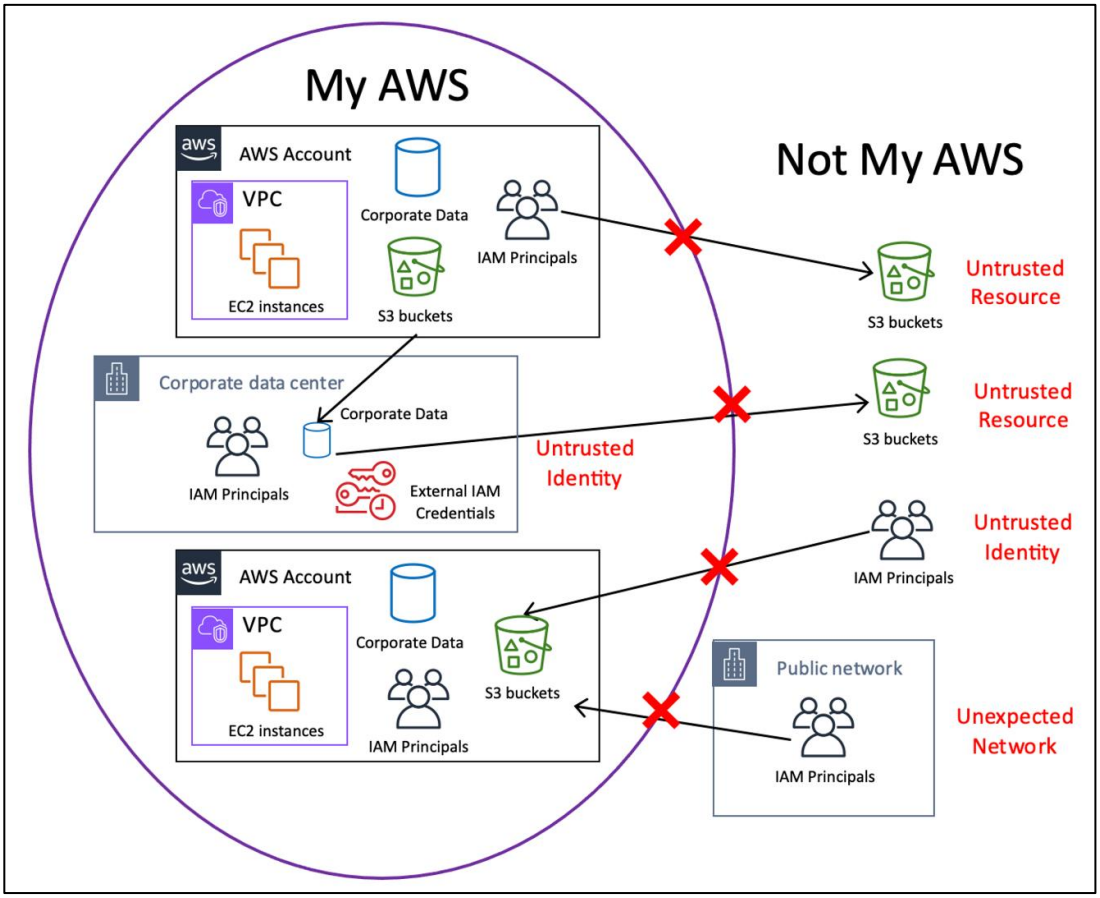
הרעיון פשוט מאוד ומגדיר את החוקים הבאים:

1. בפן הרשתי
   a. מהרשת של הארגון שלי ניתן לבצע פעולות רק באמצעות זהות של הארגון שלי
   b. מהרשת של הארגון שלי ניתן לבצע פעולות רק על משאבים של הארגון שלי
2. בפן הזהותי
   a. רק זהויות של הארגון שלי יכולות לגשת למשאבים של הארגון שלי
   b. זהות ארגונית יכולה לבצע פעולות אך ורק מהרשת שלי
3. בפן המשאבי
   a. ניתן לגשת למשאבים שלי אך ורק באמצעות זהות של הארגון שלי
   b. ניתן לגשת למשאבים שלי אך ורק מהרשת של הארגון שלי

בואו ננסה להבין מהם תרחישי התקיפה הרלוונטיים ולמפות בינם לבין החוק שמתמודד איתם:

1. תוקף שנמצא מחוץ לרשת שלי מנסה לגשת למשאבים שלי באמצעות ה-credentials (הטוקן איתו אני מבצע פעולות בענן) שלו.
   a. במקרה זה חוק 3.b ימנע את הגישה (מכיוון שהתוקף מנסה לגשת למשאב שלי שלא מהרשת של הארגון שלי)
2. תוקף גנב credentials ומנסה לפנות באמצעותם למשאב של הארגון שלי. דוגמא לתרחיש הזה: בשנת 2019 תוקף פרץ לענן של Capital One והשיג credentials לחשבון שבו היה מאוחסן מידע על גבי Bucket, ולאחר מכן ניגש לאותו ה-Bucket מהרשת הפרטית שלו (תוכלו למצוא פה חקירה מפורטת של אירוע התקיפה).
   a. חוק 2.b ימנע את הגישה (מכיוון שהתוקף מבצע פעולה עם הזהות שלי שלא מהרשת של הארגון שלי)
   b. חוק 3.a ימנע את הגישה (מכיוון שהתוקף מנסה לגשת למשאב של הארגון שלי שלא מהרשת של הארגון שלי)
3. תוקף השיג אחיזה ברשת שלי ומנסה לפנות למשאב שלו באמצעות credentials שלו. דוגמא לתרחיש הזה: בשנת 2022 מייקרוסופט זיהתה קבוצת תקיפה איראנית שמשתמשת בשירות OneDrive כשרת C&C (Command and Control) באמצעות credentials שהביאו מהסביבה העננית שלהם (תוכלו למצוא מידע מלא על התרחיש פה).
   a. חוק 1.a ימנע את הגישה (מכיוון שהתוקף מנסה לבצע פעולה מהרשת שלי שלא באמצעות הזהות של הארגון שלי)

ישנם עוד תרחישי תקיפה שנמנעים באמצעות סט החוקים שהגדרנו. ניתן למצוא סיכום של כלל תרחישי התקיפה באיור הבא:



[נלקח מהמקור הבא: https://docs.aws.amazon.com/whitepapers/latest/building-a-data-perimeter-on-aws/building-a-data-perimeter-on-aws.html]

איך זה נראה בפרקטיקה

אז ראינו עד כה איך הדברים נראים בתאוריה, אבל בואו נבין איך ניתן לקחת את זה לפרקטיקה (כלומר איך ניתן לממש את סט החוקים שהגדרנו).

על מנת לעבור לשלב הבא נצטרך להכיר עוד כמה קונספטים שקיימים ב-AWS:

1. SCP (Service Control Policy) - סט של חוקים שחלים על כלל הזהויות בארגון שלי ומאפשרים ניהול של כלל החוקים ממקום ריכוזי אחד (חשבון שנקרא Management). באמצעות המנגנון הזה נוכל להחיל את החוקים שרלוונטיים **בפן הזהותי**.
2. VPC Endpoint - VPC Endpoint Policy מאפשר לבצע פעולות מול שירות מסויים בלי הצורך לצאת לאינטרנט. לדוגמא, במידה ואנחנו רוצים לכתוב ל-Bucket נוכל ליצור VPC Endpoint של שירות S3

ובאמצעותו נוכל לגשת ולכתוב ל-Bucket שלנו. שימו לב שבאמצעות הפיצ'ר המשמעותי הזה למעשה נוכל למנוע גישה לאינטרנט ובכך לצמצם באופן משמעותי את ערוצי הדלף. בעת יצירת VPC Endpoint נוכל לשייך לו פוליסה. באמצעות המנגנון הזה נוכל להחיל את החוקים שרלוונטיים **בפן הרשתי**.

3. Resource Control Policy - ניתן לשייך פוליסה עבור משאב מסויים (Resource Based Policy). Resource Control Policy (פיצ'ר שיצא בשנה האחרונה) מאפשר לנו לנהל את ה-RBP של כלל המשאבים בצורה ריכוזית. אפשר לחשוב עליו כמעין SCP של עולם המשאבים (אם ב-SCP אנחנו אוכפים חוקים על כלל הזהויות אז ב-RCP אנחנו אוכפים חוקים על כלל המשאבים).

כדי להחיל את החוקים AWS פרסמה סט של conditions-keys שיעזרו לנו לכתוב את אותם החוקים:

1. ResourceOrgID - באמצעות התנאי הזה נוכל לאכוף את הגישה אך ורק למשאבים של הארגון שלי.
2. VpceOrgID (יצא ממש לאחרונה!) - באמצעות התנאי הזה נוכל לאכוף כי הגישה למשאבים תתאפשר אך ורק מ-VPC Endpoints של הארגון שלי.
3. PrincipalOrgID - באמצעות התנאי הזה נוכל לאכוף כי הגישה למשאבים תתאפשר אך ורק מזהויות של הארגון שלי.

וזהו! יש לנו עכשיו את כל הכלים כדי לממש Data Perimeter.

כך יראה ה-SCP שלנו:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DenyAccessNotToMineResource",
      "Effect": "Deny",
      "Action": [
        "*"
      ],
      "Condition": {
        "StringNotEquals": {
          "aws:ResourceOrgID": "MY-ORGANIZATION-ID"
        }
      }
    },
    {
      "Sid": "DenyAccessNotFromMineEndpoint",
      "Effect": "Deny",
      "Action": [
        "*"
      ],
      "Condition": {
        "StringNotEquals": {
          "aws:VpceOrgID": "MY-ORGANIZATION-ID"
        }
      }
    }
  ]
}
```

כך יראה ה-VPC Endpoint Policy שלנו:

```
{
    "Statement": [
        {
            "Sid": "AllowAccessOnlyToMyResourceAndFromMyIdentities",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "*",
            "Resource": "*",
            "Condition": {
              "StringEquals": {
                "aws:ResourceOrgID": "MY-ORGANIZATION-ID",
                "aws:PrincipalOrgID": "MY-ORGANIZATION-ID"
              },
            }
        }
    ]
}
```

כך יראה ה-RCP שלנו:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DenyAccessNotFromMineNetwork",
      "Effect": "Deny",
      "Action": [
        "*"
      ],
      "Condition": {
        "StringNotEquals": {
          "aws:VpceOrgID": "MY-ORGANIZATION-ID"
        }
      }
    },
    {
      "Sid": "DenyAccessNotFromMineIdentity",
      "Effect": "Deny",
      "Action": [
        "*"
      ],
      "Condition": {
        "StringNotEquals": {
          "aws:PrincipalOrgID": "MY-ORGANIZATION-ID"
        }
      }
    }
  ]
}
```

שימו לב שלכל אחד מה-condition keys יש וריאנט שמאפשר להחיל אותו על רמה ארגונית אחרית (Account או Organizational Unit). את כלל ה-condition keys תוכלו למצוא פה.

למה לא הכל מושלם

אז על פניו הכל נראה מושלם ופשוט. הצלחנו בצורה מאוד פשוטה להחיל חוקים שיוצרים לנו פרימטר סגור שממנו לא ניתן להדליף מידע החוצה.

אז למה לא הכל מושלם? ראשית, הקונספט הזה עדיין נמצא בראשיתו, ולכן לא כל הפעולות ולא כל המשאבים תומכים בכל היכולות שהוזכרו פה. לדוגמא:

- לא עבור כל השירותים יש תמיכה ב-VPC Endpoint. לדוגמא, השירות Route53 (השירות שאחראי על תעבורת ה-DNS ב-AWS) לא תומך ב-VPC Endpoint. מצד שני, לרוב שירותי ה-Data יש תמיכה ב-VPC Endpoint, ולכן הדבר האפשרי יהיה לפצל את השירותים שאין להם תמיכה ב-VPC Endpoint לאיזור רשתי נפרד.
- לא כל הפעולות נתמכות באמצעות ה-condition keys. ספציפית, יש רשימה שניתן למצוא בתיעוד של AWS אשר מפרטת את הפעולות שלא נתמכות באמצעות ResourceOrgID:

aws:ResourceOrgID

Use this key to compare the identifier of the organization in AWS Organizations to which the requested resource belongs with the identifier specified in the policy.

- **Availability** – This key is included in the request context only if the account that owns the resource is a member of an organization. This global condition key does not support the following actions:
  - AWS Audit Manager
    - `auditmanager:UpdateAssessmentFrameworkShare`
  - Amazon Detective
    - `detective:AcceptInvitation`
  - AWS Directory Service
    - `ds:AcceptSharedDirectory`
  - Amazon Elastic Block Store – All actions
  - Amazon EC2
    - `ec2:AcceptTransitGatewayPeeringAttachment`
    - `ec2:AcceptVpcEndpointConnections`
    - `ec2:AcceptVpcPeeringConnection`
    - `ec2:CopySnapshot`
    - `ec2:CreateTransitGatewayPeeringAttachment`
    - `ec2:CreateVpcEndpoint`
    - `ec2:CreateVpcPeeringConnection`
    - `ec2:DeleteTransitGatewayPeeringAttachment`
    - `ec2:DeleteVpcPeeringConnection`
    - `ec2:RejectTransitGatewayPeeringAttachment`
    - `ec2:RejectVpcEndpointConnections`
    - `ec2:RejectVpcPeeringConnection`

- עוד סיבה מרכזית מדוע לא הכל מושלם היא יישויות שמתנהגות באופן מוזר/חריג. נסתכל לדוגמא על הפעולה create-job תחת S3. פעולה זאת מאפשרת הרצה של פעולת S3 מול כמה אובייקטים שונים. לדוגמא, לקרוא מידע מכמה אובייקטים או למחוק כמות גדולה של אובייקטים.
  ניתן לראות כי אחד מהפרמטרים של הפעולה הוא role-arn:

```
  create-job
--account-id <value>
[--confirmation-required | --no-confirmation-required]
--operation <value>
--report <value>
[--client-request-token <value>]
[--manifest <value>]
[--description <value>]
--priority <value>
--role-arn <value>
[--tags <value>]
[--manifest-generator <value>]
[--cli-input-json | --cli-input-yaml]
[--generate-cli-skeleton <value>]
[--debug]
[--endpoint-url <value>]
[--no-verify-ssl]
[--no-paginate]
[--output <value>]
[--query <value>]
[--profile <value>]
[--region <value>]
[--version <value>]
[--color <value>]
[--no-sign-request]
[--ca-bundle <value>]
[--cli-read-timeout <value>]
[--cli-connect-timeout <value>]
[--cli-binary-format <value>]
[--no-cli-pager]
[--cli-auto-prompt]
[--no-cli-auto-prompt]
```

הפעולה עובדת בתצורה כזאת שכאשר היא מתחילה לרוץ היא מבצעת AssumeRole (פעולה אשר מאפשרת לקבל הרשאות של זהות אחרת) כדי לעבור ל-role שהעברנו לה בפרמטר. מה הבעיה? לאחר שהיא עוברת ל-role שלנו היא מבצעת את הפעולות מתוך רשת חיצונית ולא מתוך הרשת של הארגון שלנו, ולכן במידה ונחיל על הזהות הזאת את החוק כי ניתן לגשת אך ורק מהרשת הארגונית שלנו הפעולה תיכשל. לכן, נצטרך לשים החרגה ספציפית עבור זהויות מהסוג הזה.

- עוד סיבה היא שישנן פעולות שחורגות מההתנהגות הצפויה ועלולות לשבור את הפרימטר. לדוגמא, ישנה הפעולה put-bucket-notification-configuration. פעולה זו מאפשרת לנו להגדיר לאיפה נרצה לשלוח נוטיפיקציות על שינויים של אובייקטים שמתרחשים ב-Bucket. לדוגמא, נוכל לקבל התראה בכל פעם שנכתב אובייקט ל-Bucket. מה הבעיה? ניתן לבחור לשלוח את ההתראות הללו למשאב חיצוני (לדוגמא, ל-SNS Topic חיצוני), וכך למעשה כלל ההתראות ישלחו אל מחוץ לארגון (או חלילה לסביבה של התוקף). בהתראות הללו נוכל למצוא גם את שם ה-bucket או שם האובייקט, מידע אשר עלול להיות רגיש עבור גופים מסויימים. AWS עשו מאמצים לתעד את ההתנהגויות הלא צפויות הללו, וניתן למצוא את המידע הזה (ועוד הרבה פוליסות אחרות) ברפו הזה.

AWS עדיין משפרת דברים

אז למרות שהקונספט נראה שלם ובשל למעשה הוא עדיין נמצא בראשיתו והספקית נמצאת בעבודה מתמדת על מנת לשפרו ולאפשר בקרה חזקה והדוקה יותר על הפרימטר. כמה פיצ'רים מגניבים שיצאו ממש לאחרונה:

- RCP - המנגנון אשר מאפשר להחיל חוקים על כלל המשאבים בארגון זהו פיצ'ר שיצא בשנה האחרונה. למעשה עד אז היינו צריכים להחיל את החוקים שלנו על כל אחד מהמשאבים בנפרד בלי שתהיה לנו היכולת לנהל את כלל החוקים הללו בצורה ריכוזית (נשמע הזוי נכון?).
- Aws:VpceOrgID - מדובר על condition-key שהגיע אלינו באוגוסט השנה. אם עד היום היינו צריכים להשתמש ב-condition-key בשם aws:SourceVpce שבו יכולנו לציין את ה-VPC Endpoints הספציפיים שאנחנו רוצים לאפשר גישה מהם, כעת עם ה-condition-key החדש אנחנו יכולים לאכוף גישה מכל endpoint שנוצר בתוך הארגון. דרך נוספת לעשות משהו באופן ריכוזי.

AWS-ל יש תיעוד ממש טוב

אז AWS עושה מאמצים כבירים על מנת להנגיש את הקונספט הזה בצורה הנוחה ביותר. כחלק מהמאמצים הללו היא מתחזקת רפו בגיטהאב שבו היא מספקת דוגמאות לפוליסות שיכולות להוות את הבסיס עבור תפיסת ה-Data Perimeter. תוכלו למצוא שם החרגות כלליות שיש צורך לעשות, פירוט על התנהגויות חריגות של פעולות/שירותים מסויימים ועוד המון מידע מועיל למי שמעוניין להקים Data Perimeter בענן.

סיכום

אז אם עד לאחרונה לבנות פרימטר בענן הייתה משימה בלתי-אפשרית, נראה שאנחנו יותר ויותר מתקרבים לעולם שבו נוכל לנהל סיכונים גם על מידע רגיש יותר שיעלה לענן. היום, רוב מערכות הארגונים מתבססות על כלי ניטור שלאו דווקא ימנעו את דלף המידע אלא יאפשרו לנו לגלות זאת לאחר הדליפה. עם הקונספט שראינו פה הדברים יכולים להיראות אחרות - על ידי הגדרה מקדימה של סט חוקים נוכל למנוע מראש תרחישי תקיפה רבים ובכך למנוע מתוקף להוציא את המידע שלנו מחוץ לענן.

עם זאת צריך לומר שתחזוק סט החוקים הזה כיום היא משימה לא פשוטה. הדבר הרבה פעמים דורש להכיר את מנגנוני ההגנה של הספקית לעומק, התנהגויות שונות ומשונות של שירותים שונים והכרה רחבה של תרחישי תקיפה בענן. נראה שהספקיות הולכות לכיוון שבו הן ינגישו לנו יותר ויותר בקלות את הדברים שפעם היה כמעט בלתי-אפשרי לממש, ולכן גם אם כיום נראה לכם שיהיה זה קשה לתחזק את הרכיבים

השונים בפתרון לאורך זמן, אין זה אומר שהדבר לא יהיה קל יותר בעתיד, ולכן שווה לחכות ולראות איזה יכולות נוספות יצאו בהמשך.

במאמר דיברתי ספציפית על ספקית הענן AWS, אבל יכולות דומות קיימות גם בספקיות השונות. לדוגמא, ב-GCP קיים מנגנון בשם VPC-SC שמיועד להתמודד עם חלק מתרחישי התקיפה שהוזכרו כאן. אני מעודד אתכם ללמוד ולהכיר את המנגנון המתאים בהתאם לספקית בה אתם משתמשים ביומיום.

על המחבר

קוראים לי יהב פסטינגר, וכיום אני מתעסק במחקר למטרות הגנה בענן.

עבור יצירת קשר תוכלו למצוא אותי ב-Linkdein.

מקורות מידע

- https://docs.aws.amazon.com/whitepapers/latest/building-a-data-perimeter-on-aws/building-a-data-perimeter-on-aws.html
- https://dl.acm.org/doi/10.1145/3546068
- https://www.microsoft.com/en-us/security/blog/2022/06/02/exposing-polonium-activity-and-infrastructure-targeting-israeli-organizations/
- https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_scps.html
- https://docs.aws.amazon.com/vpc/latest/privatelink/what-is-privatelink.html
- https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_condition-keys.html
- https://github.com/aws-samples/data-perimeter-policy-examples
- https://cloud.google.com/vpc-service-controls/docs/overview

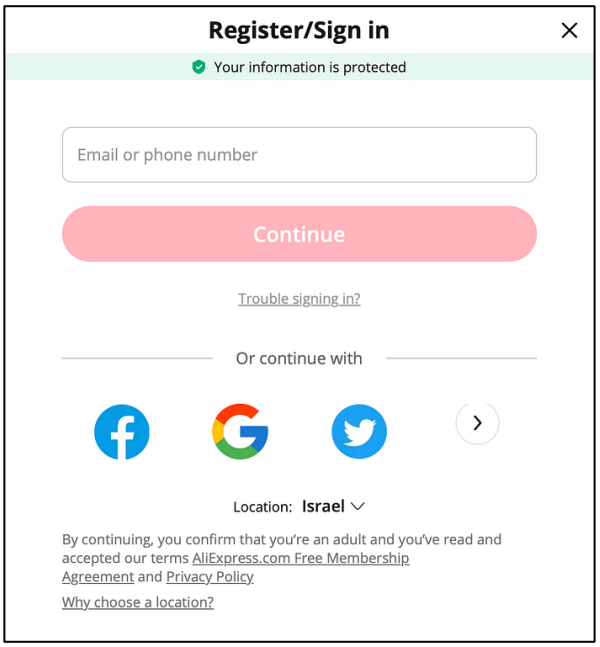
OAutheWHAT2.0

מאת סופיה שביקובסקי

הקדמה

כמה פעמים קרה לכם שנכנסתם לאתר רנדומלי באינטרנט והבנתם ששכחתם את הסיסמה לחשבון שלכם, אז עכשיו, במקום לגלול בנחת בטרמינל איקס או ב-Medium, אתם צריכים **שוב** להחליף סיסמה, בדיוק כמו בפעם שעברה שנכנסתם לאתר הזה. הסאגה הזאת לא נגמרת, מי זוכר את הסיסמה שלו בכלל בימינו?!

אז כן, הבעיה הזו הייתה רלוונטית המון זמן, עד שיום אחד פשוט הופיע הכפתור הבא, ומאז הכל השתנה.



אבל רגע – מה זה בכלל ואיך זה עובד?

בעיה ופתרון

תחשבו כך: יש לכם חשבון יציב באינטרנט, אולי זה גוגל, פייסבוק או אינסטגרם. אתם נמצאים בו קבוע ויודעים לעקוב אחרי כל מה שמתרחש בו. בתוך החשבונות האלו נמצא המידע המדויק עליכם כמו רשימת חברים, תחומי עניין, תמונות, מסמכים, הרשאות וכו'.

כדי שלא תצטרכו לעשות דופליקציה של אותו המידע בכל אתר שאתם נרשמים אליו, נוצר Framework שמאפשר התחברות ואימות בתצורת SSO וכן משיכת המידע הרלוונטי ישירות מהחשבון שלכם. הדבר הזה מתאפשר באמצעות טכנולוגיית OAuth, פרוטוקול שמאפשר לתת לאפליקציה גישה מוגבלת ומבוקרת למשאבים שלך, בלי לחשוף מידע פרטי כמו סיסמה.

במאמר הזה אני אתייחס יותר לשימוש ב-OAuth כפונקציה המשמשת את החשבונות לאימות, בתצורת SSO (Single Sign On), ולחולשות שיכולות להמצא בתצורה הזו.

OAuth

OAuth נועד לאפשר למשתמשים לתת גישה מוגבלת למשאבים שלהם באינטרנט לאפליקציות צד שלישי, מבלי לחשוף סיסמה, הפרוטוקול הזה פותח לראשונה ב-2006 על ידי צוותים מחברות כמו גוגל וטוויטר, כתגובה לצורך לפתרון של הבעיה שדיברנו עליה קודם, יותר ויותר שירותים רצו לאפשר אינטגרציה עם תוכנות חיצוניות, אבל בצורה מבוקרת.

הגרסה הנפוצה ביותר היא OAuth2.0, שפורסמה ב-2012. היא שינתה לגמרי את הדרך בה מתבצעת אותנתיקציה באינטרנט, במקום להעביר סיסמאות בין שירותים, המשתמש קיבל token (token) זמני שמאפשר לאפליקציה לבצע פעולות מסויימות בלבד ובזמן מוגבל.

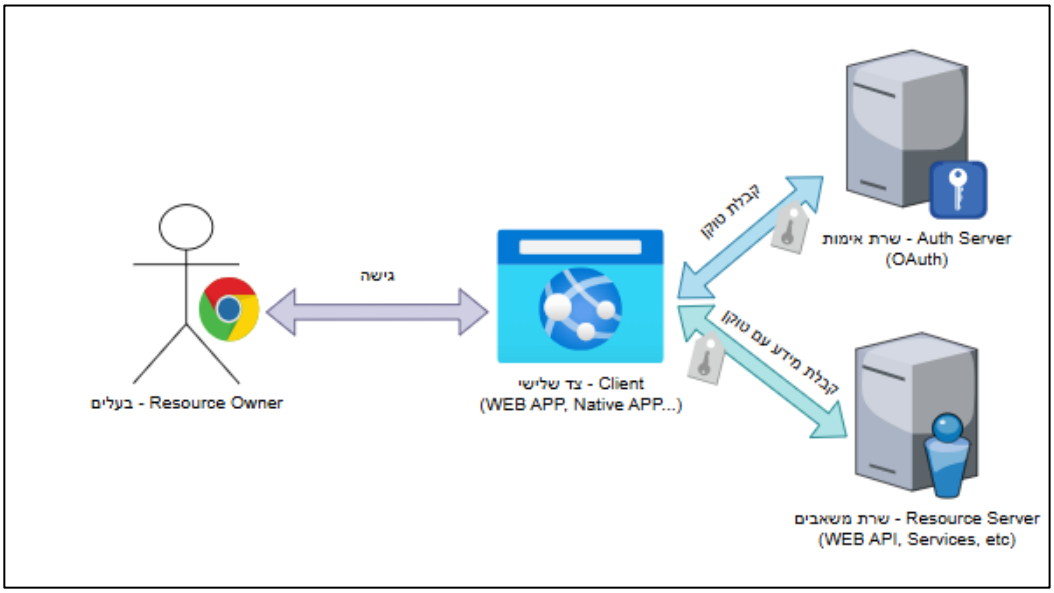
OIDC

בגלל שבמקור ה-Framework של OAuth היה שימושי לאותוריזציה בלבד, זאת אומרת אך ורק לגישה למשאבים ספציפים, ולא לצורך אותנטיקציה - אימות, נוצרה הרחבה שכן תאפשר לבצע את האימות באמצעות ה-OAuth והיא OIDC, בשמה המלא OpenID Connect, שמספקת שכבת אימות. OIDC מתבסס על OAuth ומוסיף אליו שימוש בtoken-ים מסוג JWT, סט scopes והרחבת מידע הרלוונטי למשתמש. בזכות ההרחבה הזו ההתחברות באמצעות SSO נהייתה הרבה יותר פשוטה ונפוצה באתרים ברחבי האינרנט, אבל יחד עם זאת, גם תצורת OIDC וגם OAuth מהווים חוליה שיכולה להיות מנוצלת אם לא מוגדרת היטב.

מבנה של ה-Framework

צדדים עיקריים (ACTORS):

- בעלים (Resource Owner) - המשתמש שמנהל את המידע והמשאבים (לדוגמא חשבון משתמש ב-GOOGLE).
- צד שלישי (Client Application) - הגוף שמבקש גישה למשאבי משתמש כדי לספק פונקציונליות כלשהי (לדוגמא "Sign in with GOOGLE").
- שרת אימות (Authorization Server) - הגוף שאחראי על אימות המשתמשים והנפקת Access Token, או Authorization Code.
- שרת משאבים (Resource Server) - שרת שאחראי על שמירת מידע של משתמשים וגישה אליהם דרך ה-token.



בעצם איך זה עובד בפועל:

בעלים (Resource Owner) הם המשתמש שמחזיק את החשבון או המידע הספציפי והרלוונטי. במקרה שלנו זה אני, משתמשת שיש לה חשבון Gmail שמכיל את רשימת אנשי הקשר שלי, מיילים, פרופיל וכו'.

אני רוצה להתחבר לאתר קניות כלשהו באינטרנט (Client Application), שמציע לי אפשרות להתחבר עם גוגל, הוא לא רוצה את הסיסמה שלי ולא שאמלא ידנית פרטים, אלא רק הרשאה לקרוא את הכתובת מייל שלי כדי ליצור חשבון או להתחבר אליו.

השרת שינהל את האוטנתקציה (הזדהות) שלי מול אותו האתר צד שלישי הוא השרת אימות (Authorization Server), לדוגמא שרת של GOOGLE שמבצע לוגין ומחזיר Access Token.

עם ה-token שקיבלנו מהשרת אימות נוכל לגשת למידע בשרת משאבים (Resource Server), שיכול להיות לדוגמא ה-API של Gmail, שמחזיק את הרשימת אנשי קשר שלי. האתר קניות יקבל את המידע הרלוונטי אך ורק אם ה-token שהוא קיבל מהשרת אימות תקף.

token-ים וסוגי ה-Grant Types

חלק מהותי בכל התהליך של קבלת גישה עם ה-OAuth הוא קבלת token, הדבר הזה מתבצע באמצעות ה-Grant Type, שזוהי בעצם "סוג דרך" שהאפליקציה (Client Application) מבקשת דרכה גישה לנתונים של המשתמש בשרתי המשאבים (Resource Server).

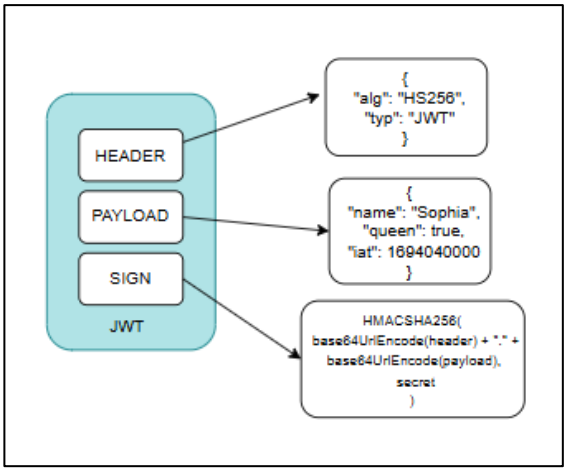
ה-Grant Type קובע את סדר הצעדים בתהליך, זאת אומרת מה האפליקציה שולחת, איזה token צריכה, איך הם מועברים ועוד. כמובן שזה משפיע ישירות על רמת האבטחה של ההחלפת מידע, כל OAuth Server חייב להיות מוגדר לתמוך ב-Grant Type מסויים, והאפליקציית צד שלישי בוחרת באיזה סוג היא רוצה להשתמש לצורך קבלת הרשאות.

סוג token-ים ב-Oauth:

בתהליך האימות עם ה-OAuth מופיעים מספר סוגי token-ים מרכזיים.

Access Token - token עם טווח חיים קצר שמאפשר ללקוח לבצע קריאות API לשרת משאבים. הוא מוגבל בזמן (לדוגמא למספר שעות), מקושר ל-SCOPE ספציפי (הרשאות ספציפיות שאפשר לקבל), בדרך כלל JWT.

- **JWT** או JSON Web Token הוא token דיגיטלי שמשמש להעברת מידע בין צדדים בצורה מאובטחת.



ה-JWT שימושי ונוח מכיוון שהוא מאפשר אימות (אותנטיקציה) כדי לזהות את המשתמש הספציפי והזדהות (אותוריזציה), בכדי לקבוע את ההרשאות שהמשתמש זכאי להן ואיזה פעולות הוא מורשה לעשות.

Refresh Token - token עם תכולת חיים ארוכה, שמאפשר ללקוח לקבל Access Token חדש מבלי שהמשתמש יצטרך להזדהות שוב. ה-token הזה נשמר בצד שרת ומאפשר חידוש token-ים בצורה מאובטחת.

Authorization Code - קוד חד פעמי שמתקבל אחרי שהמשתמש נותן הסכמה לאתר צד שלישי לגשת לנתונים שלו. בעצם הלקוח שולח את הקוד לשרת הרשאות יחד עם ה"סודות" של המשתמש, כמו Client_id ו-Client_secret, ובתמורה מקבל Access Token. זו דרך יחסית מאובטחת יותר, כי ה-Access Token לא עובר דרך הדפדפן.

סוגי Grant Types:

Authorization Code Grant Type:

סוג ה-grant הנפוץ ביותר באפליקציות Server-Side. היתרון שלו הוא רמת אבטחה הגבוהה, כי כל הנתונים הרגישים, כגון Access Token ופרטי המשתמש לא עוברים דרך הדפדפן אלא דרך ערוץ מאובטח בין שרתים.

ה-FLOW:

Authorization Request

הלקוח שולח בקשה לשרת ההרשאות לקבל הרשאה מהמשתמש, כולל פרמטרים כמו id, scope, state ועוד.

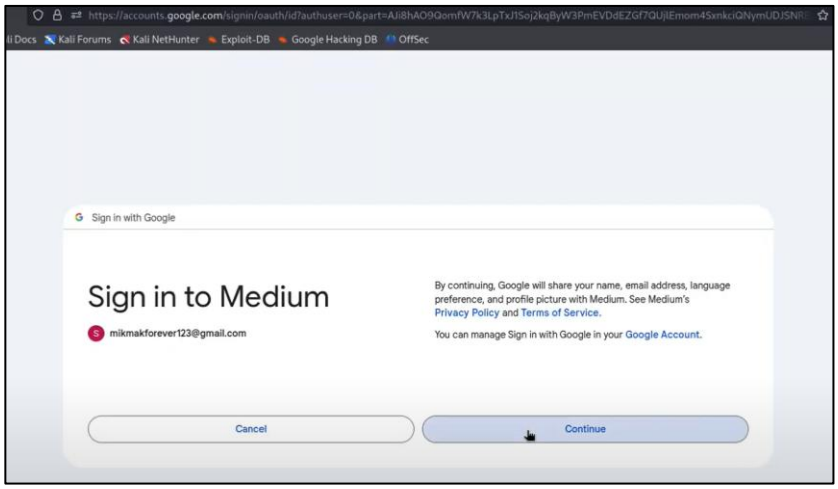
```
GET /authorization?client_id=12345 &redirect_uri=https://client-app.com/callback
&response_type=code
&scope=openid%20profile
&state=ae13d489bd00e3c24
HTTP/1.1
```

פרמטרים:

- Client id – מזהה של הצד השלישי אל מול השרתי אימות.
- Response type – מגדיר לאיזו תשובה מצפה הצד השלישי (קוד/token).
- Redirect uri - כתובת שאליה המשתמש יופנה אחרי התחברות או אחרי קבלת ההרשאות.
- Scope - רשימת הרשאות שהצד השלישי יכול לקבל על המשתמש, זה מגדיר איזה נתונים המשתמש מנגיש.
- State – ערך ייחודי שמקושר לסשן הנוכחי של המשתמש, זוהי הגנה מפני מתקפות CSRF.

User Login & Consent

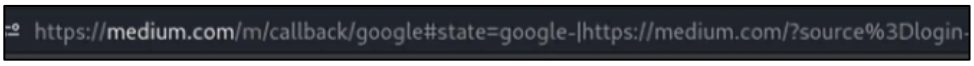
המשתמש נכנס לחשבון שלו ומסכים לאפשר ללקוח לגשת לנתונים שלו לפי ה-Scope שהוגדר. ברגע שהמשתמש אישר, האישור יהיה תקף וקיים כל עוד הסשן שלו פעיל.



Authorization Code Grant

אם המשתמש מסכים, הדפדפן מופנה לכתובת Callback, או בשמה השני – Redirect_URI, יחד עם ה-Authorization Code שקיבל וה-State.

```
GET /callback?code=a1b2c3d4e5f6g7h8&state=ae13d489bd00e3c24 HTTP/1.1
```



Access Token Request

עכשיו כשקיבלנו את הקוד, צריך להחליף אותו לtoken אמיתי. הלקוח שולח בקשה לשרת האימות כדי להחליף את הקוד ולקבל token. באמצעות בקשת POST מהשרת של האתר לקוח (אתר צד שלישי) לשרת אימות , המידע על המשתמש, כמו ה-Client_secret והקוד אימות עצמו עוברים ב"ערוץ מאובטח".

ערוץ מאובטח, משמעותו היא שהתקשורת הזאת מתבצעת Server-to-server, משמע הבקשה לא עוברת דרך הדפדפן, אלא בקשת HTTPS מתקבלת ישירות בשרת, כך שהמידע הרגיש לא נחשף בשום שלב, גם אם יאזינו לתעבורה של הדפדפן.

```
POST /token HTTP/1.1
Host: oauth-authorization-server.com
client_id=12345&client_secret=SECRET &redirect_uri=https://client-app.com/callback
&grant_type=authorization_code
&code=a1b2c3d4e5f6g7h8
```

Access Token Grant

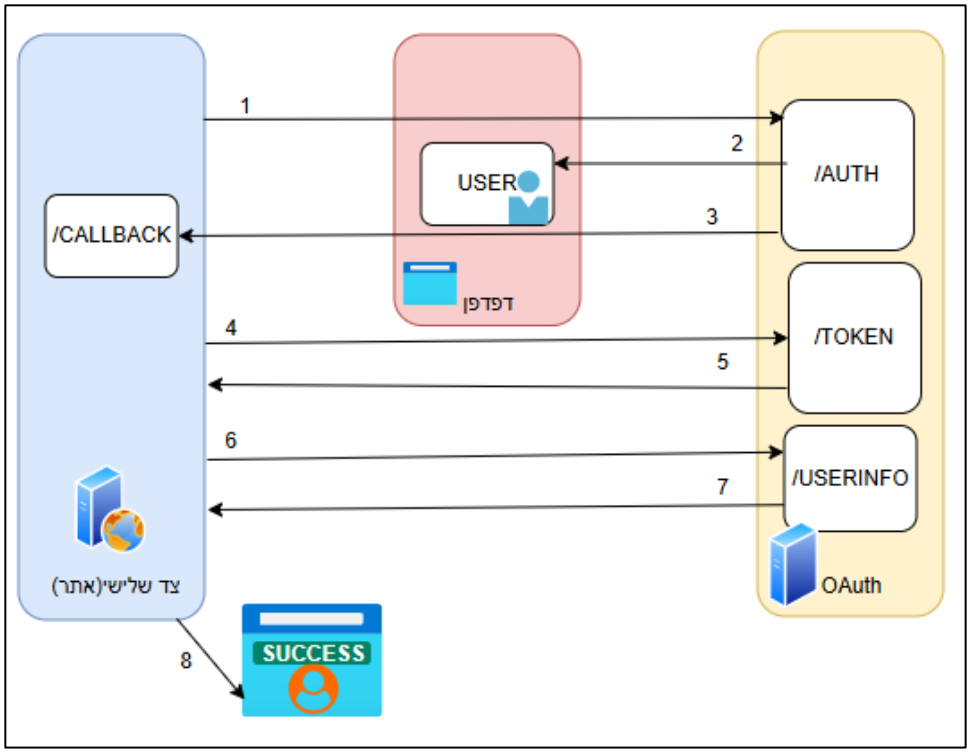
השרת מאשר ומחזיר Access Token לאתר צד שלישי (הלקוח), כולל ה-Scope.

API Call

הלקוח משתמש ב-Access Token כדי לגשת לנתונים של המשתמש בשרתי משאבים.

Resource Grant

שרת המשאבים מאשר את ה-token ושלח את הנתונים המבוקשים ללקוח.



Implicit Grant Type:

סוג ה-Grant שמתאים בעיקר ל-SPA, שזה Single Page App, או אפליקציות לא מאוד מתוחכמות ללא שטח אחסון. היתרון כאן הוא פשוט, אבל הפשטות הזו מגיעה גם עם קשיים אבטחתיים, מכיוון שכאן ה-token מגיע ישירות, ללא קוד "מתווך" כלשהו. הכל בדפדפן.

Authorization Request

הצד השלישי מבקש הרשאות מהמשתמש, אך הפעם הבקשה מוגדרת מראש על Response_Type=Token.

User Login and Consent

המשתמש מתחבר ומאשר לאתר צד שלישי לקבל את ההרשאות. התהליך כאן זהה כמו ב-Grant קודם.

Access Token Grant

במידה והמשתמש אישר את התהליך, הצד השלישי מקבל ישירות את ה-token בתוך ה-URL.

כאן ניתן לראות בפירוש שה-token חוזר אל הדפדפן ישירות מתוך ה-Callback.

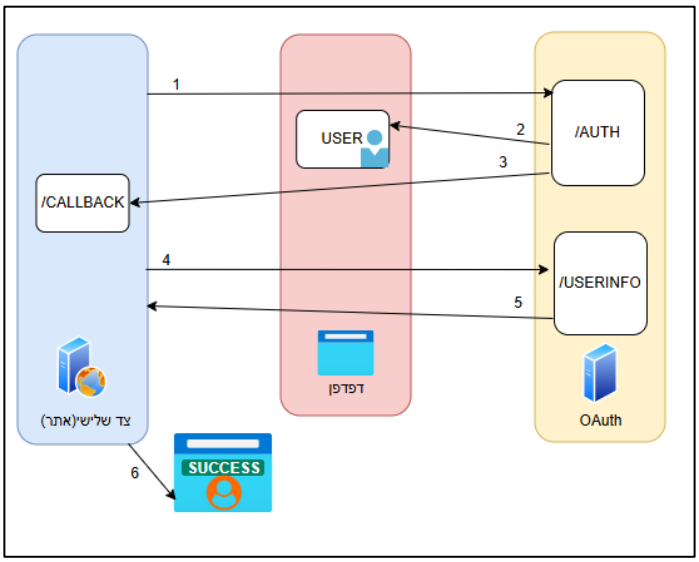
```
GET /callback#access_token=z0y9x8w7v6u5
&token_type=Bearer
&expires_in=5000
&scope=openid%20profile
&state=ae13d489bd00e3c24
HTTP/1.1
```

API Call

הצד השלישי מפענח את ה-token ומשתמש בו לצורך קבלת מידע מהמשתמש.

Resource Grant

שרת המשאבים מאשר את ה-token ושולח את הנתונים המבוקשים ללקוח.



:OpenID Connect

ה-OIDC משתמש ב-Oauth Authorization Code Flow, אבל מוסיף עליו שכבת אימות של זהות. ההבדל המרכזי, בנוסף ל-Access Token, הלקוח מקבל גם ID token שהוא JWT חתום עם Claims על המשתמש, שזה יכול להיות השם שלו, מייל, UID, מתי משתמש התחבר לאחרונה ועוד...

Authorization Request

כמו קודם, אך הפעם הבקשה תכלול Scope של OpenID.

```
GET /authorization?client_id=12345
&redirect_uri=https://client-app.com/callback
&response_type=code
&scope=openid%20profile%20email
&state=ae13d489bd00e3c24 HTTP/1.1
```

השוני כאן, לעומת התהליך הרגיל, הוא הוספת scope.

User Login & Consent

המשתמש מתחבר לחשבון גוגל שלו לדוגמא ומסכים לשתף את הנתונים שם (כמו מייל, שם ועוד.)

Authorization Code Grant

השרת מחזיר ל-Redirect URI את ה-Authorization Code עם ה-State.

```
GET /callback?code=a1b2c3d4e5f6g7h8&state=ae13d489bd00e3c24
```

Token Request

האתר צד שלישי שולח בקשת POST לשרת אימות (OAuth), ההבדל הוא שעכשיו הוא מבקש גם את ה- id token.

```
POST /token HTTP/1.1
Host: oauth-authorization-server.com
client_id=12345
client_secret=SECRET
redirect_uri=https://client-app.com/callback
grant_type=authorization_code
code=a1b2c3d4e5f6g7h8
```

Token Response

שרת אימות מחזיר את ה-Access Token ואת ה-ID Token.

```
{
"access_token": "z0y9x8w7v6u5",
"token_type": "Bearer",
"expires_in": 3600,
"id_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVC..."
}
```

API Call

הלקוח יכול להשתמש ב-Access Token כדי לגשת למשאבים בשרת (כמו אנשי קשר).

User Info via ID Token

או להשתמש ב-ID Token כדי לדעת מי המשתמש, מתי הוא התחבר וכו, בלי צורך בפניות נוספות לשרת משאבים.

חולשות ב-OAuth

אז כן, למרות שהפרוטוקולים OAuth ו-OIDC נועדו לאפשר אימות והרשאות בצורה מאובטחת ונוחה (ממש כמו Sign in with GOOGLE), הפרוטוקולים האלה עדיין חשופים לחולשות. הסיבה העיקרית היא שה-Framework מאוד מסתמך על הדפדפן, שרתי צד שלישי וtoken-ים שנעים על גבי הרשת. הנקודות תורפה של הפרוטוקול נובעות מחוסר אימות נכון, זליגת token-ים או פשוט תצורה לא מספיק נכונה של Redirect URIs ו-Scopes. בקצרה וכרגיל - החולשות נמצאות דווקא במערכות ואתרים שלא דואגים לעבוד עם קונפיגורציה נכונה.

סוגי חולשות:

Redirect URI Manipulation

במהלך תהליך האימות ב-OAuth, התגובה של השרת יכולה להיות רגישה, כי היא מכילה את החלק הקריטי ביותר, שהוא הקוד או ה-token של המשתמש. אם ניתן "ללכוד" את האלמנטים האלה, ניתן כך להשיג את המידע הרגיש ולהתאמת עם פרטים של משתמש אחר. הדרך הכי נפוצה לעשות את זה היא ניצול של הפרמטר Redirect_URI. אם השרת OAuth לא בודק את הפרמטר באופן מספיק קפדני או שהוא מוגדר בצורה כללית מידי (לדוגמא https://test.*), ניתן לנתב את המשתמש לפלטפורמה "זדונית" במקום לאפליקציית צד שלישי הרצויה.

לדוגמא, אם הייתה דרך לשנות את הכתובת Callback לכתובת של שרת חיצוני שאוסף את הקודי אימות\token-ים של כל מי שניגש אליו כפרמטר.

```
GET /authorization?client_id=1234
&redirect_uri=https://sus-app.com/collect
&response_type=code
&scope=openid%20profile
&state=ae13d489bd00e3c24 HTTP/1.1
Host: oauth-authorization-server.com
```

בדרך כלל החולשה נמנעת על ידי שרת OAuth, שבו מוגדרים הדומיינים הספציפיים אליהם ניתן לנתב את ה-token , אבל זה אכן אפשרי במקרה שבו יש באתר צד שלישי פגיעות ל-Open Redirect.

Open Redirect – הוא פגיעות WEB, מצב שבו אתר לגיטימי מאפשר להפנות משתמש לכתובת חיצונית בלי לוודא אם היא בטוחה, מה שיכול לאפשר לתוקף להפנות משתמשים לאתר זדוני, במקום כתובת לגיטימית.

```
GET /authorization?client_id=1234
&redirect_uri=https:
//client-app.com/redirect?next=https://sus-app.com/collect
&response_type=code
&scope=openid%20profile
&state=ae13d489bd00e3c24 HTTP/1.1
```

במקרה הזה המשתמש ינותב קודם ל-client-app.com/redirect, ואז ל-https://sus-app.com/collect. אם האתר צד שלישי פגיע, ה-Authorization Code יישלח לשרת של התוקף.

CSRF due to Missing State Parameter

ב-OAuth, פרמטר state הוא ערך אקראי וייחודי שנשלח מהאתר צד שלישי לשרת אימות בזמן בקשת ההרשאה. המטרה שלו היא לוודא שהבקשה שהתקבלה בשרת של האפליקציה היא אותה בקשה שהמשתמש התחיל. זה מונע מצב שבו ניתן להזריק קוד או token לא חוקי ולהתחבר בשם המשתמש. הבעיה היא שהפרמטר הזה הוא לא בדיוק חובה, לכן אם הוא חסר או לא נבדק כראוי, נוצרת חולשה קריטית שמאפשרת לנצל את החוסר אימות הזה.

לדוגמא כאשר משתמש רוצה להתחבר לאתר צד שלישי שמבצע אימות באמצעות גוגל, הוא שולח בקשת אימות לשרת אימות. השרת מאמת את המשתמש ומחזיר קוד לאתר צד שלישי, שאליו המשתמש מעוניין להתחבר. במצב תקין, הפרמטר state יישלח יחד עם הקוד, אך במידה והוא לא מוחל כראוי, הוא לא יהיה נוכח.

התוקף יכול לשלוח למשתמש קישור פישינגי זדוני, או בצורה כזו או אחרת לגרום למשתמש ללחוץ על הקישור כמו זה: "https://client-app.com/oauth?code=KJSUSIEGHBSHEOKJJSOIPE", במידה והמשתמש ילחץ על הקישור, הקוד יתקבל על ידי משתמש ויעבור לשרת של האפליקציה, השרת יחשוב שמדובר בקוד חוקי מהמשתמש עצמו ויקשר את החשבון של התוקף עם החשבון משתמש.

התוצאה - התוקף מתחבר לחשבון של המשתמש באתר צד שלישי עם הפרטי גוגל משלו.

Implicit Grant Exploitation

כפי שהבנו מקודם, תצורת Implicit Grant היא תצורת OAuth "מופשטת", שמאפשרת קבלת token ישירות אל הדפדפן, בלי החלפת קודים בין שרתים.

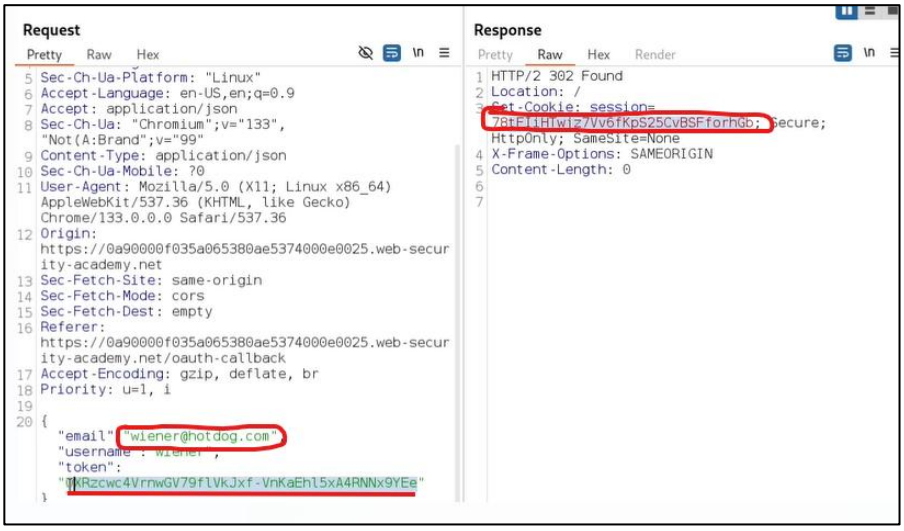
אם האתר צד שלישי לא מוודא בצורה נכונה שהמידע מגיע עם ה-Access Token שבאמת שייך למשתמש שמנסה להתחבר, מתקבלת אפשרות להזריק מידע לא נכון.

לדוגמא משתמש מנסה להתחבר עם גוגל לאתר צד שלישי, הדפדפן ישירות מקבלת Access Token:

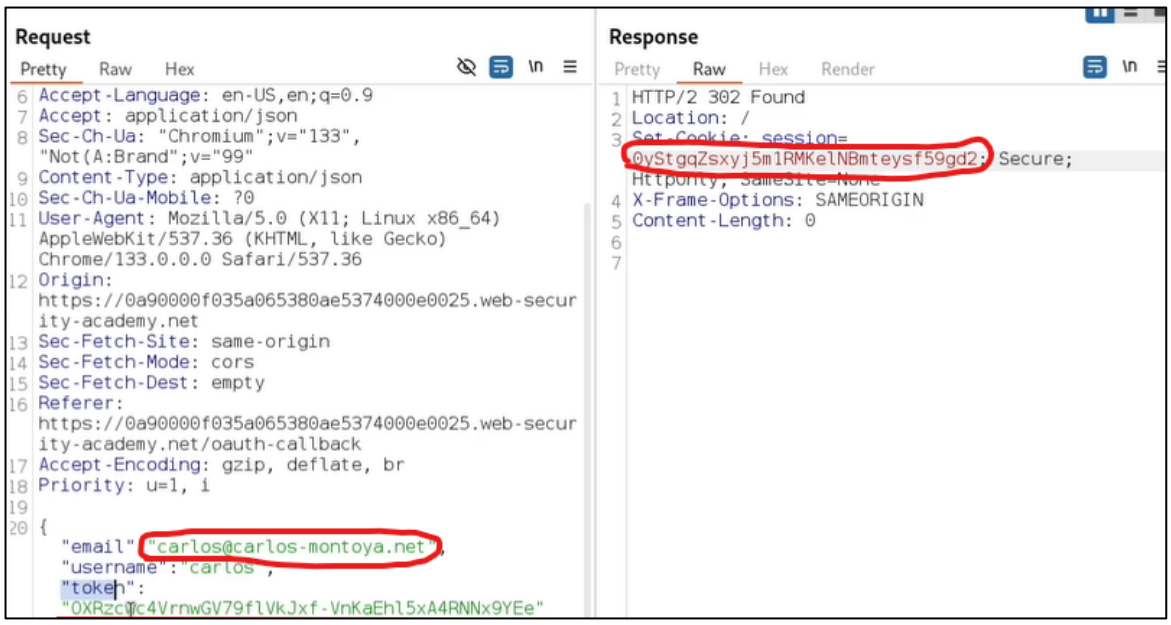
```
https://client-app.com/callback#access_token=1234&token_type=bearer&expires_in=3600
```

בגלל שכאן, השולח של ה-token הוא הדפדפן, אין שום תקשורת מאובטחת כמו תקשורת משרת לשרת וה-token חשוף, יש אפשרות להשיג token-ים של משתמשים אחרים. ניתו להעתיק token של משתמש אחר, או ליצור token מזוייף, לדוגמא באמצעות XSS או מתקפה אחרת, ה-token יישלח לאתר צד שלישי והוא יתקבל בלי אימות מסויים מול השרת אימות, מה שקורה מידי פעם באתרים שלא מוודאים את האמינות של token-ים לאחר קבלתם, ה-token המזוייף יוכל לאפשר גישה לחשבונות אחרים, כלומר – גניבת זהות.

במידה וה-token לא נבדק, ניתן לערוך פרטי משתמשים בתוך הבקשה וכך להשיג מידע, כמו coockies, וכך להשתלט על הסשן הקיים.



כאן ניתן לראות בקשת אותנטיקציה עם פרטי משתמש שיש לנו את ה-token שלנו (wiener, חובבי portswigger יזהו ישר), כתגובה קיבלנו את ה-cookie של המשתמש, וכך השגנו session בשמו.



אם נשנה רק את המייל והשם למשתמש קורבן שלנו, אם קיים session שלו במערכת, נוכל להשיג את ה-cookie שלו גם כן, בלי שהאתר צד שלישי יוודא את ה-token (שאפילו לא שינינו!) וכך נקבל גישה למשתמש שלו, בלי להשתמש לא בסיסמא ולא ב-token שלו.

Token Manipulation

במידה ונשלח JWT (לרוב ID Token) מזוייף לאתר צד שלישי והוא לא מוודא את התקינות/חתימה של ה-token מול המפתח הציבורי, האתר צד שלישי עלול לסמוך על כך שהמשתמש מזוהה ולהעניק לו גישה לנתונים או פעולות שהיו אמורות להיות מוגבלות למשתמש לגיטימי בלבד.

```
POST /oauth/callback HTTP/1.1
Host: client-app.com
Content-Type: application/x-www-form-urlencoded

id_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwiZW
1haWwiOiJhdHRhY2tlckBleGFtcGxlLmNvbSIsIm5hbWUiOiJBdHRhY2tlciIsImlhdCI6MTY5Nz
YyMjQwMCwiZXhwIjoxNzAwMjM4NDAwfQ.fakeSignature
```

אם האתר צד שלישי לא בודק את החתימה מול המפתח הציבורי של ה-OAuth Provider, הוא עלול להאמין שהמשתמש מזוהה ולהעניק לו הרשאות.

בנוסף, ניתן לשנות את הפרטים בתוך ה-Payload של ה-JWT וכך להשיג גישה למשתמש קיים.

Scope Abuse

אתר צד שלישי יכול לבקש הרשאות גבוהות מידי, או רחבות מידי (גישה ליותר מידי משאבים, ללא צורך), מעבר למה שהוא צריך. זה אפשרי מכיוון שה-Scopes מוגדרים בתור Strings והמשתמש לא תמיד יודע או שם לב או מבין אילו הרשאות הוא נותן.

Token Replay

ה-Access Token שנגנב יכול לשמש שוב ושוב לגישה לנתונים, גם אם פג תוקף, מכיוון שבאתר צד שלישי אין מנגנון של ניהול token-ים נכון.

Insecure Storage of Tokens

יש מצב שבו האחסון של הtoken-ים מצד האתר לקוח לא מיושם בצורה נכונה, לדוגמא אם מאוחסנים ב-Local Storage של השרת, או בקבצים לא מוצפנים. הדבר הזה מקל על גניבת token-ים.

OAuth2.1

כחלק מניסיון למנוע את כל החולשות האבטחתיות שנובעות משימוש ב-OAuth2.0, שזו הגרסה שדיברנו עליה לאורך כל המאמר, נמצא פתרון והוא שדרוג ה-Framework לגרסא משופרת, OAuth2.1, שמטרתה לקחת את כל הכשלים של הגרסא המקורית ולהפוך אותם כדרישות מחייבות, כדי לצמצם את המשטח תקיפה.

מהעדכונים שנוספו:

1. ביטול של תצורת Implicit Grant
2. חובת שימוש ב-PKCE, או בשמו המלא Proof Key for Code Exchange, מנגנון אבטחה ב-Oauth, שמוסיף סיסמה חד פעמית להחלפת הקוד בין השרתים, כדי לוודא שהקוד שנשלח לא יורט בשום שלב.
3. הגדרת Redirect_URI ספציפיים מראש.
4. חובת שימוש ב-nonce וב-state.
5. הקשחת ניהול ואחסון token-ים

מקורות מידע

- https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/05-Authorization_Testing/05-Testing_for_OAuth_Weaknesses
- https://astrix.security/learn/blog/part-2-how-attackers-exploit-oauth-a-deep-dive/
- https://portswigger.net/web-security/oauth#what-is-oauth
- https://www.vaadata.com/blog/understanding-oauth-2-0-and-its-common-vulnerabilities/
- https://oauth.net/2.1/

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-178 של Digital Whisper, אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב: למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה רבות כדי להביא לכם את הגליון.

ניתן לשלוח כתבות וכל פניה אחרת דרך עמוד "צור קשר" באתר שלנו, או לשלוח אותן לדואר האלקטרוני שלנו, בכתובת editor@digitalwhisper.co.il.

על מנת לקרוא גליונות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המגזין:

www.DigitalWhisper.co.il

"T4lk1n' 80ut a r3vo7u710n 5ounds like a wh15p3r"

הגליון הבא בתקווה ביום האחרון של חודש אוקטובר!

אפיק קסטיאל, ספיר פדרובסקי

30.09.2025