

Inside LSASS

מאת יהונתן אלקבס

הקדמה

הבסיס של כל פתרונות אבטחת המידע מכיל 4 אבני יסוד - אימות, הצפנה, בידוד וניתוח התנהגותי. למעשה, כמעט כל מוצר\פתרון הגנה שתבחרו עובד באמצעות וריאציה כזו או אחרת של אותם אבני יסוד. מבט זריז על מגוון פתרונות אבטחה מתחומים שונים יבהיר את הנקודה; בין אם ברמה התקשורתית כדוגמת VLANs ו-FW, ברמה התשתיתית כדוגמת NTLM / Kerberos ובין אם ברמה האפליקטיבית כדוגמת TLS ו-JWT. למעשה, גם אם ננסה לאחד מספר פתרונות לכדי EPP מוגמר כדוגמת EDR/AV או נעלה שלב לרמה הרשתית כמוצר MDR/XDR השואב לוגים מכל פינה ברשת ושולח את כלל המידע לניתוח בענן על ידי מודלים מבוססי AI לזיהוי אנומאליות בזמן אמת - הבסיס הוא אותו בסיס.

LSASS הוא מרכיב קריטי במערכת ההפעלה Windows. לאחר ביצוע מוצלח של logon לעמדת קצה, כתלות בסוג האימות, LSASS עשוי להכיל במרחב כתובות הזיכרון שלו סיסמאות clear text, NTLM hashes ואף TGTs במידה ונעשה שימוש ב-Kerberos. כאמור, הכל כתלות במנגנון האימות שבוצע. התהליך אחראי על אכיפת מדיניות אבטחה, ניהול אימות והרשאות משתמשים ושמירה על אבטחת מידע רגיש.

עקב תפקידו המכריע באבטחת המערכת, בין אם בסביבה הלוקאלית ובין אם בסביבת הדומיין, LSASS מהווה יעד פופולארי עבור תוקפים המעוניינים לנצל נקודות תורפה ב-process כדי לקבל גישה למשאבים נוספים. במידה ותוקף צלח בהשגת dump של התהליך, המרחק אל פיצוח סיסמאות ותזוזה רוחבית אינו עָנְף. במשך שנים מגנים ותוקפים היוו חתול ועכבר בנושא הנ"ל בדיוק בגלל הסיבות הללו.

מהצד האדום, פעולות כדוגמת memory injection, process hollowing, התקנת דרייברים זדוניים ושימוש בכלים המיירטים את בקשות האימות על ידי שימוש ב-Windows API כגון Mimikatz נוצרו ומימשו שיטות חדשות בכל עת שיצאו עדכונים וגרסאות מגננה חדשות עבור LSASS. מהצד הכחול, הפתרונות המובילים שמיקרוסופט הציגה בנושא היו Protected Process Light ו-Credential Guard.

הראשון משתמש בווירטואליזציה מבוססת חומרה כדי לבודד מידע רגיש באמצעות ה-Hypervisor והשני מגביל את היכולת של תוכנות צד שלישי לגבש אינטרקציה עם תהליכים מוגנים על ידי שימוש בחתימות.



במאמר הקרוב אגע בצד הכחול ובמאמר המשך אקדיש את תשומת הלב לאדום. לפיכך, לאורך המאמר אנחנו הולכים לדבר על המענה ההגנתי ולהציג כיצד רכיביו עזרו לעצב ולאבטח את ארכיטקטורת המערכת הנוכחית של LSASS (ותהליכים נוספים). אבל, בסופו של יום חשוב להבין כי הם מתבססים על אותם 4 אבני יסוד שהזכרנו בפתיח.

אתנחתא לפני שמתחילים: ניתן להבין לפי מספר ה-Buzzwords וראשי התיבות שנזרקו עד כה בהקדמה שהמאמר מיועד לאסכולה מתקדמת, עם זאת אעשה את המירב כדי להסביר בצורה הברורה ביותר עבור אלו שעדיין לא שם.

לפני שמתחילים

סייבר ואבטחת מידע לא נועדו לייצר כסף, אלא לחסוך ממנו. אני יודע שזה משפט קצת מוזר לשמוע בתחילת מאמר טכני על הגנת תהליכים בסביבת Windows אבל זו פרספקטיבה אלמנטרית שחשוב להבין. המודל העסקי של (כמעט) כל ארגון הוא למקסם את אחוזי הרווח והכוח של החברה. מתוך הבנה שנפילה למתקפת סייבר עלולה להשבית את סביבת הייצור והפרודאשן (Ransomware כדוגמה), להסגיר פטנטים רגישים עקב ריגול תעסוקתי ולזעזע את שווי מניות החברה - צמצום משטח תקיפה וניטור רשתי הם הסיבה מדוע מועסקים מיישמים, מגנים וחוקרי סייבר רבים.

המאמר הנוכחי מגיע במטרה להציג מענה לסט מאוד ספציפי של ווקטורי תקיפה - הוצאת מידע רגיש מתהליכים בפלטפורמת Windows בעת תקיפה אקטיבית לשם תזוזה רוחבית. כבר עכשיו חשוב לי לעצור ולומר כי פרקטיקות כדוגמת (JIT) Just-in-Time Admin, (JEA) Just Enough Admin המיישמים את אלמנט ההקשחה הבסיסי של Least Privileges לא יוצגו כלל במאמר. רעיון זה הוצג בצורה לא רעה (וטכנית) במאמר [Blue Hands-on BloodHound](#). ממליץ מאוד את הקריאה למי שעוד לא יצא.

במסגרת המאמר כעת אנחנו הולכים לשים דגש על הפתרונות ש-Microsoft הציגה בשביל להגן בצורה מלאה ואיכותית יותר על תהליכים רגישים בסביבת Windows ובמאמר המשך כיצד תוקפים מצליחים (או לא) לעקוף אותם. זה כנראה לא יפתיע אתכם שקיימים מספר רב של מנגנונים Built in לאבטחת מבני נתונים **בזיכרון** של תהליכים מסוגים שונים בפלטפורמת Windows.

הפתרונות הרלוונטיים בנושא הם:

- Security Reference Monitor (SRM)
- Protected Process Light (PPL)
- AppContainer Isolation, Session 0 Isolation & Secure Desktop
- Windows Resource Protection (WRP)
- Windows Memory Management



- Windows Integrity Control (WIC)
- Mandatory Integrity Control (MIC)
- Isolated User Mode (IUM) by Hyper-V
- Virtual Secure Mode (VSM) / Virtual Base Security (VBS)

בהחלט רשימה לא קצרה (וכנראה שיש עוד). אני לא הולך לנסות להסביר את כולם במאמר הקרוב מ-2 סיבות עיקריות - אחת, על כל אחד מהם אפשר לכתוב מאמר שלם, ושניים, אין לי את הידע לכך. עם זאת, אנחנו כן הולכים להכיר לעומק את אלו שקשורים בצורה ישירה ל-LSASS (חתן המאמר הנוכחי). קרי, רוב המאמר יתמקד ב-2 בפתרונות מהרשימה - PPL & VBS.

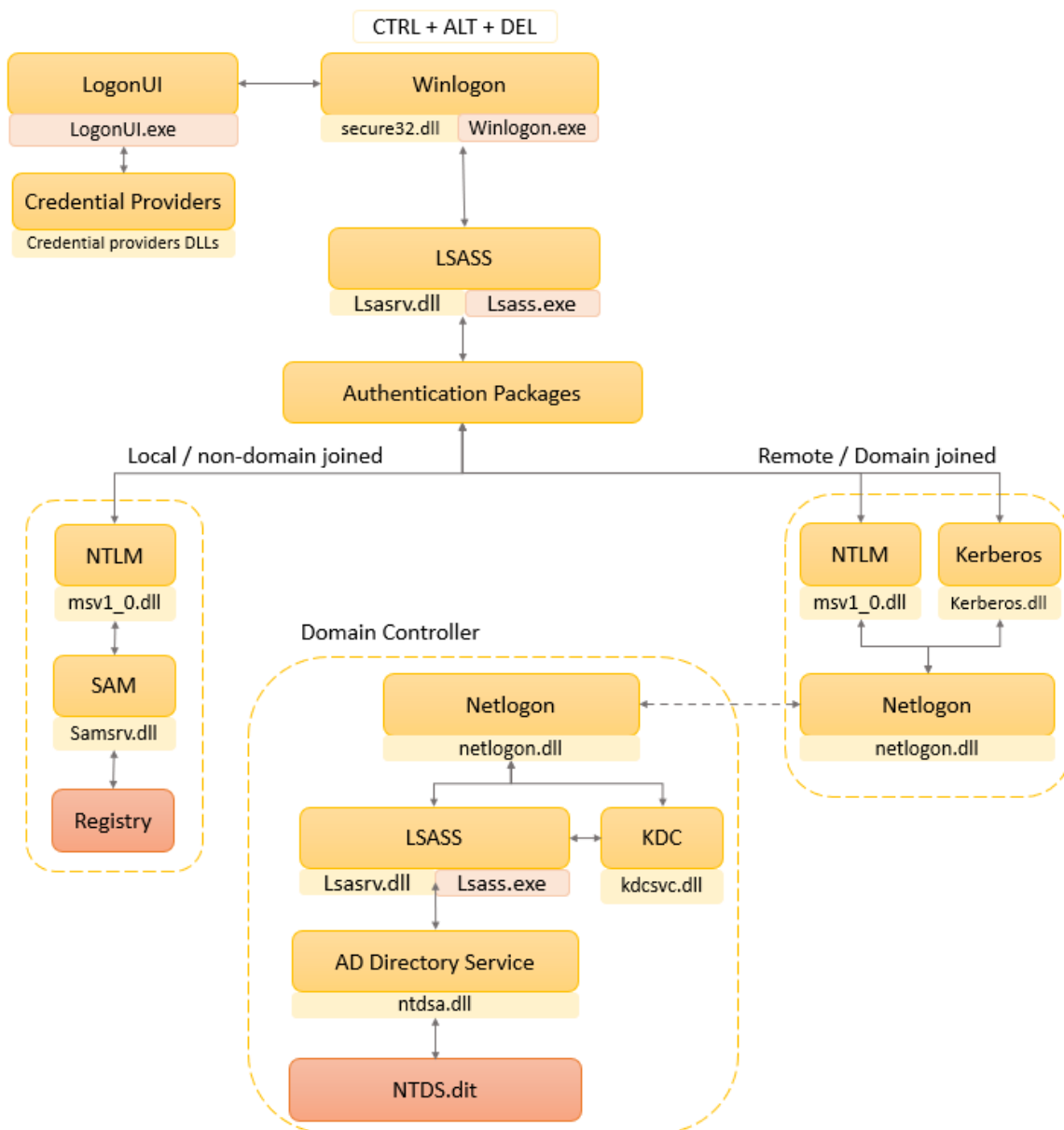
הערה: רוב החומר התיאורטי והרעיונות שאציג במסגרת המאמר כבר הוצגו [בדוקומנטציה](#) הרשמית של מייקרוסופט ובספר [Windows Internals 7th edition](#). מטרת המאמר אינה לכתוב אותם אלא לאגד את המידע הרלוונטי לנושא בליווי מחקר בסביבת דומיין שנבנה לצורך הדגמה והסבר המאמר.

על רגל אחת - Windows Credentials Lifecycle Internals

בשביל להבין את מורכבות הבעיה אנחנו צריכים קודם לקחת צעד אחורה ולהבין בצורה מלאה את שיטות האימות וההרשאה ב-Windows. למזלי, נכתבו במסגרת המגזין מאמרים מעולים בנושאים אלו בסביבת הדומיין אשר מציגים לפרטי פרטים את אופן פעולת הפרוטוקולים הלוקחים חלק בכל הסיפור. שני מאמרים מופתיים שאני מאוד ממליץ לקרוא בנושא הם [1-st Step to Tame a Kerberos: Know Your Enemy](#) ו-[2nd Step to Tame a Kerberos: Hit It Where It Hurts](#) מאת עדי מליאנקר ונתנאל כהן שמסבירים כיצד תהליכים ענפים כדוגמת Kerberos ו-NTLM עובדים בסביבה דומינית וכיצד תוקפים אוהבים לנצלם (המאמר השני מציג ליטרלי כמעט כל מתקפה שקיימת!).

תודות לצמד המאמרים (והחוקרים) הנ"ל, לא ניכנס לעומק ההבנה של פרוטוקולי Kerberos ו-NTLM. עם זאת, מחזור החיים של ה-Credentials בסביבה ה-Windows-ית החל מהתחברות משתמש ועד התנתקותו מהמערכת הוא נושא ששווה להציג (ולהבין) פעם אחת בצורה מלאה. לכן, על מנת לגבש תסריט צלול לכיצד פרטי משתמש עוברים במערכת ההפעלה לשם אימות והרשאה, ניכנס במספר עמודים הקרובים למבנה תהליכי האימות ב-Windows.

אני מאמין בקפיצה ישר למים ולכן בלי עיכובים נוספים - ארכיטקטורת תהליכי האימות ב-Windows:



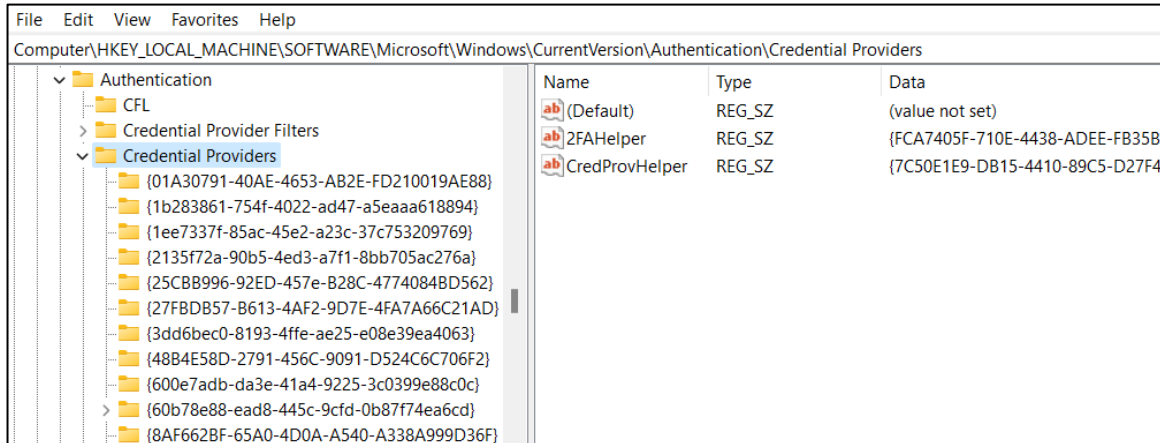
מה אנחנו רואים פה? בפסקה אחת: התחברות אינטראקטיבית לוקאלית מתבצעת על ידי תהליך כניסה (WinLogon) אשר מריץ את תהליך ממשק משתמש לכניסה (LogonUI) בשביל לקבל ממנו את נתוני היוזר באמצעות ה-Credential providers דרך Secure32.dll. משם, על ידי העברת הפרטים לתהליך LSASS, שימוש בחבילת אימות אחת או יותר, מתבצעת בדיקה במסדי הנתונים של SAM או Active Directory על פרטי המשתמש והאם הוא ראשי להיכנס למערכת. במידה וכן, Winlogon יוצר סביבת desktop ומכניס את המשתמש אליה. ה-Local Security Authority (LSA) היא תת מערכת ב-Windows שאחראית על האימות והתחברות של משתמשים בסביבה הלוקאלית ובדומיין והיא זו שמנהלת את כל התהליך.

זה היה הסבר על תהליך ההתחברות ב-Windows במינימום מילים. נקודה שחשוב להבין היא שתהליך אימות המשתמש לא מתבצע בקומפוננטה אחת אלא בשיתוף פעולה והעברת הנתונים של המשתמש בין מספר רכיבים במערכת ההפעלה (מיוצג על ידי חצים דו-כיווניים באיור).

הסבר קצת יותר טכני שמציג את הסיפור המלא למי שמתעניין:

Winlogon.exe הוא תהליך מאומת (trusted) שאחראי לניהול אינטראקציות משתמש הקשורות להתחברות, הרצת תהליכי מערכת, יצירת סביבות עבודה (Desktops) ואבטחה. במילים אחרות, הוא עושה מלא דברים, אחד מהם זה לתאם את כלל ה-flow בתהליך ה-logon של היוזר מול הממשקים (הפנימיים והחיצוניים) של מערכת ההפעלה. כך למשל, על מנת לקלוט את פרטי המשתמש (שם + סיסמה) הוא קורה אל **LogonUI.exe** או נועל את העמדה במידה ובוצעו יותר נסיונות חיבור כושלים ממספר הפוליסה. בצורה דומה, הוא אחראי על כך שאף תהליך לא מאומת לא יקבל גישה אל סיסמת המשתמש.

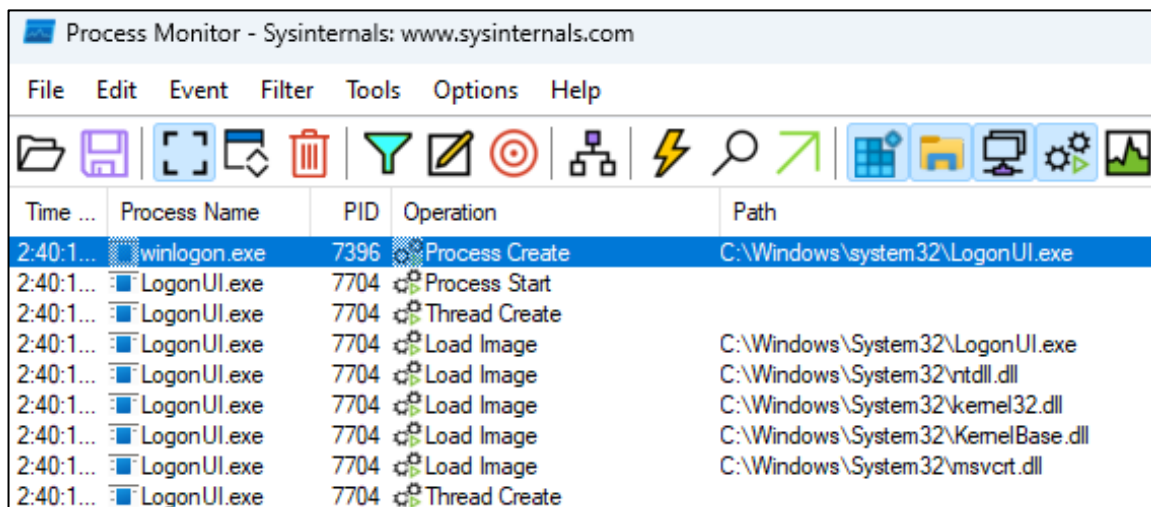
אבל אין זה אומר שהוא פועל לבד. למעשה, הוא מסתמך (בצורה די עיוורת) על **credential providers** שמותקנים על המערכת ששיגו לו את פרטי המשתמש. **credential providers** הם 'ברוקרים' לתהליך האימות אשר מיוצגים על ידי אובייקטי COM בתוך DLLs ומספקים שיטות אימות שונות כדוגמת **face recognition** או **smartcard**. ישנם 21 **credential providers** דיפולטיביים ב-registry על התקנה חדשה של Windows 11:



Name	Type	Data
(Default)	REG_SZ	(value not set)
2FAHelper	REG_SZ	{FCA7405F-710E-4438-ADEE-FB35B}
CredProvHelper	REG_SZ	{7C50E1E9-DB15-4410-89C5-D27F4}

Winlogon נחשב תהליך קריטי, במידה והוא קורס - כל המערכת תקרוס. לכן, בשביל להגן על מרחב הכתובות שלו מטעויות (תמימות ופחות תמימות) של **credential providers** מתבצע שימוש בתהליך שונה לטעינתם ולהצגת ממשק ה-logon למשתמשים - **LogonUI.exe**. אם מסיבה מסוימת הקוד של אחד ה-**credential providers** קרס, Winlogon פשוט יאתחל את **logonUI.exe** מחדש והוא בתורו יחל את תהליך איסוף נתוני המשתמש (ופעולות נוספות).

באמצעות Process Monitor ניתן לראות את רצף התהליך וטעינת קבצי DLL למימוש האימות בהמשך:



Time ...	Process Name	PID	Operation	Path
2:40:1...	winlogon.exe	7396	Process Create	C:\Windows\system32\LogonUI.exe
2:40:1...	LogonUI.exe	7704	Process Start	
2:40:1...	LogonUI.exe	7704	Thread Create	
2:40:1...	LogonUI.exe	7704	Load Image	C:\Windows\System32\LogonUI.exe
2:40:1...	LogonUI.exe	7704	Load Image	C:\Windows\System32\ntdll.dll
2:40:1...	LogonUI.exe	7704	Load Image	C:\Windows\System32\kernel32.dll
2:40:1...	LogonUI.exe	7704	Load Image	C:\Windows\System32\KernelBase.dll
2:40:1...	LogonUI.exe	7704	Load Image	C:\Windows\System32\msvcrt.dll
2:40:1...	LogonUI.exe	7704	Thread Create	

אז איך נראה תהליך ה-logon המלא ב-Windows?

תהליך ההתחברות למכונת קצה מתחיל כאשר המשתמש לוחץ על Ctrl+Alt+Del (רצף הנקרא [SAS](#) [בדוקומנטציה](#)), הקשה זו מטרגרת את Winlogon להתחיל את תהליך LogonUI אשר קורא להשגת שם משתמש וסיסמה מה-credential providers שבו נעשה שימוש והעברתם אל Winlogon בצורה מוצפנת. במקביל, Winlogon יוצר רשומת התחברות לוקאלית ייחודית SID עבור היוזר ומשתמש בה בשביל להקצות desktop למשתמש (מאופיין במקלדת, עכבר, מסך וכד').

לאחר מכן, Winlogon קורא אל תהליך lsass.exe בשביל לאמת את המשתמש ומעביר לו את ה-SID. LSASS בתורו מפעיל את ברוקר האימות (שנקבע לפי פוליסות בדומיין, תמיכה\מגבלות של תוכנת הקליינט וכו'), אלו הן **חבילות האימות**. חבילות אימות הן קבצי DLL המבצעים בדיקות אימות בזמן תהליך logon אינטראקטיבי. כשמדובר בהתחברות אינטראקטיבית לדומיין, לרוב נראה את Kerberos.dll נטען למרחב הכתובות של lsass.exe.

במידה והמחשב לא חלק בדומיין (התחברות לוקאלית) ו\או אין למכונה תקשורת ל-DC, החבילה שתטען ותנהל את אופרציית ה-challenge-response אל מול המשתמש היא msv1_0.dll (אשר מממשת את פרוטוקול האימות של NTLM).



אם כי שתי אלו הן חבילות האימות הסטנדרטיות לביצוע אימות אינטראקטיבי ב-Windows, חשוב להבין שהן לא חבילות האימות היחידות שקיימות, ניתן לראות זאת ברגיסטרי:

Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa			
Name	Type	Data	
(Default)	REG_SZ	(value not set)	
auditbasedirecto...	REG_DWORD	0x00000000 (0)	
auditbaseobjects	REG_DWORD	0x00000000 (0)	
Authentication P...	REG_MULTI_SZ	msv1_0	
Bounds	REG_BINARY	00 30 00 00 00 20 00 00	
crashonauditfail	REG_DWORD	0x00000000 (0)	
disabledomaincr...	REG_DWORD	0x00000000 (0)	
everyoneinclude...	REG_DWORD	0x00000000 (0)	
forceguest	REG_DWORD	0x00000000 (0)	
fullprivilegeaudit...	REG_BINARY	00	
LimitBlankPassw...	REG_DWORD	0x00000001 (1)	
LsaCfgFlags	REG_DWORD	0x00000000 (0)	
LsaPid	REG_DWORD	0x00000280 (640)	
NoLmHash	REG_DWORD	0x00000001 (1)	
Notification Pack...	REG_MULTI_SZ	scecli	
ProductType	REG_DWORD	0x00000003 (3)	
restrictanonymou...	REG_DWORD	0x00000000 (0)	
restrictanonymo...	REG_DWORD	0x00000001 (1)	
SecureBoot	REG_DWORD	0x00000001 (1)	
Security Packages	REG_MULTI_SZ	""	

במידה וחבילת האימות הצליחה לאמת את היוזר, LSASS מחזיר קריאה אל Winlogon אשר מצרף את ה-SID של המשתמש אל ה-token של תהליך ההתחברות למכונה ומפעיל logon shell בשם המשתמש.

חבילת האימות של MSV1_0 לוקחת את פרטי ה-logon של המשתמש ושולחת את שם היוזר עם ה-hash של הסיסמה שלו אל ה-SAM הלוקאלי (בהתבסס על שירות Samsrv.dll) בשביל לקבל את נתוני המשתמש - קבוצות שהיוזר חבר בהן, הגבלות וכד'. לאחר קבלת הנתונים, MSV1_0 בודקת שלא קיימות הגבלות (כגון - שעות התחברות, מכונות קצה ועוד) בנוגע לאותו משתמש. במידה וקיים cache של התחברות דומיננטית (קרי - המשתמש התחבר לעמדה בדומיין לפני ועדיין קיימת רשומה לגבי ה-session שלו) MSV1_0 יגש לנתונים שקיימים בו באמצעות הפונקציונאליות של LSASS לשמירת סודות.

בגלל לא מעט חולשות ארכיטקטורה בפרוטוקול NTLM (ובמימוש שלו ב-MSV1_0) הפרוטוקול שמתבצע בה עיקר השימוש בדומיין הוא Kerberos 5. השוני המהותי הוא העובדה שלהבדיל מ-NTLM, חבילת האימות של קרברוס משתמשת בסט אלגוריתמים מתקדם יותר, חותמת על הודעות חשובות ומערבת צד שלישי לביצוע האימות - שירות KDC שרץ ב-DC. לכן, לוגים להתחברות ירשמו הן בעמדת הקצה והן ב-DC (ולאחר מכן גם בשרת אליו מעוניינים לפנות כמובן). הספרייה שתהליך lsass.exe ב-DC מריץ בזיכרון שלו למימוש פרוטוקול קרברוס היא kdcsvc.dll.

מהאירור אפשר לראות כי LSASS רץ גם במכונת הקצה וגם בצד שרת. כלומר, בשני הצדדים, עושים שימוש ב-netlogon כברוקר לתקשורת בין תהליכי LSASS (על ידי שימוש בפורט מיוחד הנקרא ALPC, עליו נסביר בפרק אחר) וכך ניתן להעביר את פרטי האימות של המשתמש מצד אל צד.



DLL נוסף שנטען לטובת האימות הוא **Ntdsa.dll** שאחראי על ניהול ממשק ה-API מול **NTDS.dit** (מסד הנתונים של ה-AD), הוא מאפשר פונקציונאליות רבה על עץ המידע שמאפיין את מבנה הדומיין. **Ntdsa.dll** מאפשר לרכיבי המערכת ליצור, לשנות ולמחוק אובייקטי AD, לשלוח שאילתות בנוגע נתוני AD ולבצע רפליקציות בין DC-ים.

לאחר אימות היוזר, ה-Kdcsvc מחזיר credentials דומיינים אל LSASS והוא בתורו מחזיר את תוצאת האימות (הצליח או לא) אל העמדה בה ה-logon התרחש. LSASS מוסיף לאובייקט שמייצג את משתמש את ה-SIDs שהיוזר מקושר אליהם בנוסף (קבוצות), מסתכל במסד הפוליסות הלוקאלי LGPO של המכונה (אשר מתעדכן אל מול פוליסות הדומיין ב-DC מעת לעת) ולפי כל רשומות ה-SIDs שהיוזר מקושר אליהם מקבל access token שמייצג את ההרשאות ומשאבים שהוא זכאי אליהם. לאחר יצירת ה-token, LSASS משכפל אותו, יוצר handle שניתן להעביר אל Winlogon (ביחד עם הודעה שהאימות התרחש בהצלחה) וסוגר את ה-handle של עצמו.

LSASS מעביר אל Winlogon את הפרטים הבאים - קישוריות ל-access token, ל-LUID ל-session ההתחברות (כל session ב-Windows מקבל מזהה ייחודי לוקאלי כשהוא נוצר בהתאם לחבילת האימות באמצעותה הוא נוצר) ומידע על פרופיל המשתמש. ניתן לבחון את ה-sessions הפעילים בעמדת הקצה וב-DC באמצעות LogonSessions ו-Process Explorer בחבילת Sysinternals. כך למשל נוכל להסתכל על 2 רשומות לדוגמה במכונת הקצה WIN-CLIENT3 (בצד שמאל) וכדי להשלים את התמונה, נוכל לראות את ה-token-ים בתהליך LSASS שרץ ב-DC מחזיק אצלו באמצעות ה-handles (בצד ימין).

```

PS C:\Users\domain_admin\Desktop\SysinternalsSuite> .\logonsessions64.exe
LogonSessions v1.41 - Lists Logon session information
Copyright (C) 2004-2020 Mark Russinovich
Sysinternals - www.sysinternals.com

[0] Logon session 00000000:000003e7:
  User name: JON\WIN11-CLIENT3$
  Auth package: Negotiate
  Logon type: (none)
  Session: 0
  Sid: S-1-5-18
  Logon time: 2/19/2023 8:34:22 AM
  Logon server:
  DNS Domain: JON.NET
  UPN: WIN11-CLIENT3$@JON.NET

[8] Logon session 00000000:00085304:
  User name: JON\domain_admin
  Auth package: Kerberos
  Logon type: Interactive
  Session: 1
  Sid: S-1-5-21-1403467943-1367565381-54031474-1108
  Logon time: 2/19/2023 8:34:41 AM
  Logon server: DC01
  DNS Domain: JON.NET
  UPN: domain_admin@JON.NET
  WIN-CLIENT3

```

Token	NT AUTHORITY\SYSTEM:19871f
Token	NT AUTHORITY\SYSTEM:3e7
Token	NT AUTHORITY\SYSTEM:3e7
Token	JON\Administrator:623fa
Token	JON\Administrator:623fa
Token	JON\Administrator:623fa
Token	JON\Administrator:623fa
Token	JON\Administrator:623fa
Token	JON\Administrator:623fa
Token	NT AUTHORITY\SYSTEM:3e7
Token	NT AUTHORITY\SYSTEM:19871f
Token	NT AUTHORITY\SYSTEM:1c0bfff8
Token	JON\WIN11-CLIENT3\$:1e37359
Token	JON\WIN11-CLIENT3\$:1e3dfbd
Token	JON\domain_admin:1ed2c92
Token	JON\domain_admin:1e9e6ee
Token	JON\domain_admin:1e9e6ee
Token	JON\WIN11-CLIENT3\$:1e37359
Token	JON\domain_admin:1e9e6ee
Token	JON\domain_admin:1e9e6ee
Token	NT AUTHORITY\SYSTEM:1bde728
Token	JON\domain_admin:1ed2c92

המספר שמיוצג ב-Logon session (שמאל) ובסיומת כל יוזר (ימין) הוא ה-LUID של token המשתמש.

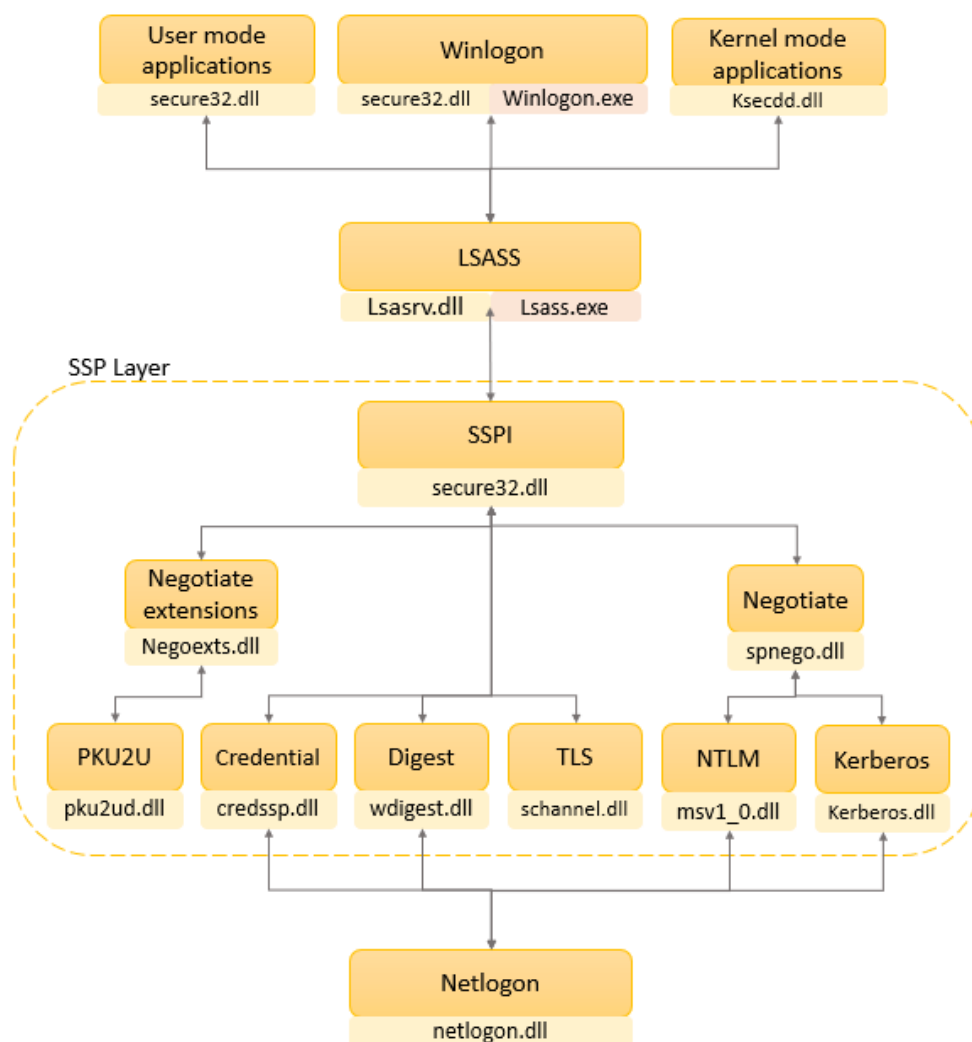
קבוצה חשובה שאי אפשר לא להציג כאשר מדברים על ארכיטקטורת האימות של Windows הם כלל ה-**Security Support Provider, SSP** בקצרה ו'ספקי אבטחה' בעברית שבורה. אפליקציות ב- Windows Server יכולות לאמת משתמשים על ידי שימוש בממשקים של SSPs בצורה אבסטקטית - ב-**SSPI**. Server מיוצגים על ידי ספריות DLL שתומכת בספפיקציה של ממשק ה-SSPI. לכן, מפתחים אינם צריכים להבין את המורכבות של פרוטוקולי אימות ספציפיים כדוגמת Kerberos/NTLM או לבנות פרוטוקולי אימות באפליקציות שלהם (כל מי שחושב שלהמציא את הגלגל מחדש זה פתרון טוב עתיד ללא מעט עליות לא צפויות בדרכו).

בדוקומנטציה שלה, Microsoft מתייחסת אל תשתית ה-SSPI כבסיס המרכזי לאימות במערכת ההפעלה. כל יישומי המערכת ב-kernel mode ואפליקציות user mode נדרשות לעבור דרכה על מנת לאמת משתמשים. SSPI זמין דרך מודול **Secur32.dll**, שהוא API המשמש להשגת שירותי אבטחה משולבים לאימות, שלמות ההודעות ופרטיות ההודעות. הוא מהווה שכבת אבסטרקציה בין פרוטוקולים ברמת האפליקציה לפרוטוקולי אבטחה.

מכיוון שיישומים שונים דורשים דרכים שונות לזיהוי או אימות משתמשים ודרכים שונות להצפנת נתונים בזמן שהם עוברים ברשת, SSPI מספק דרך לגשת לספריות DLL המכילות שיטות אימות ופונקציות הצפנה שונות. מכאן גם ניתן להבין את השוני הטרמינולוגי בין SSPs אל חבילות אימות (Packages Authentication) שהוצגו בתרשים הקודם - SSPs הם קטגוריה כללית של רכיבי אבטחה ב-Windows שמספקים פונקציונליות קריפטוגרפית ושירותי אבטחה.

כלומר, בעברית פשוטה, SSPI מאפשר קריאה אל פונקציות פנימיות בכל ספק אבטחה בהתאם לסוג הפרוטוקול שנבחר לאופרציית האימות.

אם לעקוב אחרי ארכיטקטורת ה-Credentials שאיתה פתחנו את הפרק, הרי שניתן לייצג את ארכיטקטורת ממשק SSPI בצורה הבאה:



כפי שמוצג באיור, ה-SSPI ב-Windows מספק מכניזם שמכיל tokens לאימות על פני ערוצי התקשורת הקיימים (sockets, קריאות RPCs וכד') בין מחשב הלקוח לשרת. כאשר יש צורך לאמת שני מחשבים או מכשירים כדי שיוכלו לתקשר ביניהם בצורה מאובטחת, הבקשות לאימות מנותבות ל-SSPI, אשר בתורו משלים את תהליך האימות, ללא קשר לפרוטוקול הרשת הנמצא כעת בשימוש.

כיצד זה קורה? כגוף אבסטרקטי, האובייקטים ש-SSPI מחזיר הם בינאריים והשרת והלקוח לא משנים אותם כלל בעת המעבר בין הבקשות. כך ניתן להעביר אותם לשכבת SSPI בצד שני לפירסור וניתוח. בצורה כזו, תשתית ה-SSPI מאפשרת לאפליקציות שונות להשתמש במודלי אבטחה שונים הזמינים במחשב (או ברשת) מבלי לשנות את הממשק לאותה מערכת אבטחה.

משפט מגניב שעוזר להבין באנגלית פשוטה את הנושא: **SSPI is used to Kerberize applications.**



בחבילות ה-SSP אפשר לראות את `msv1_0.dll` ואת `Kerberos.dll` שהרחבנו עליהם מקודם, על הבחירה מול מי לבצע את האימות מבין השניים (כאמור כתלות בחוקות דומיין ובתמיכה של מכונת הקצה) אחראית ספריית `Spnego.dll` כאשר דיפולטיבית היא תבחר בקרברוס.

מדוע TLS מוצג ומה הוא עושה שם? ובכן, קריפטוגרפיה של מפתח ציבורי היא פרקטיקה שימושית ובעיקר מאוד אלסטית. דיברתי על כך לא מעט במאמר [על תעודות דיגיטליות](#) לפני כמה שנים. הפלטפורמה של Windows ובפרט Windows server מאפשרת אימות מבוסס `web`. העברת זהויות על גבי האינטרנט, אימות זהויות אל מול שרתי `web` מאובטחים, שימוש ב-Azure Active Directory וחיבורו לסביבה הלוקאלית הן דוגמאות פרקטיות לכך. `Schannel.dll` משתמש במודל `client-server` בשביל לממש את היישומים השונים של TLS בגרסאותיו השונות.

בנושא אחר, `Wdigest.dll` היא גם חלק מחבילות האימות המוצעות. למרות ש-`wdigest` הוא פרוטוקול ישן, לא רלוונטי ופגיע. הסיבה העיקרית לכך שהוא לא מומלץ לשימוש היא מכיוון שהוא מעביר סיסמאות ו-credentials בתצורת `clear text` או `MD5`. הוא תומך `web` (על ידי ביצוע RPC לשרת) אז אולי הוא עדיין שם בשביל תמיכה לאחור ב-IIS או Internet Explorer (שהספיק גם להיות deprecated). בכל מקרה, למרות הדוקומנטציה של Microsoft בנושא, כשפתחתי את רשימת ה-DLLs שתהליך `lsass.exe` קורא להן `Wdigest.dll` הופיע במלוא הדרו.

שתי SSPs שפחות רלוונטיות לטובת המאמר הן CredSSP ו-PKU2U אבל אנחנו כבר פה אז נרחיב עליהן בקצרה בכל זאת. CredSSP מאפשרת Single Sign-on (SSO) למשתמש. מכירים את זה שאתם פותחים session ב-RDP ויש לכם גישה למשאבים שלכם גם שם? ובכן, CredSSP מאפשרת לאפליקציות לבצע דיליגציה ל-credentials שלכם ממחשב הלקוח לשרת היעד.

כחלק מתשתית ה-SSPI, חברת Microsoft מאפשרת תמיכה ב-SSPs extension, כלומר, הרחבות לתשתית האימות עצמה. אתם עובדים בסביבה היברידיית של Azure AD ורוצים להזדהות באמצעות ה-ID online שיש לכם ולא עם משתמש רשתי\לוקאלי? אתם יכולים באמצעות PKU2U. תאמת שיש מלא אופציות מגניבות בנושא אז למי שרוצה להרחיב שווה לעבור על [הדוקומנטציה](#).

אעצור כאן את ההסבר על אימות והרשאות ב-Windows. אנשים (שהם בבירור לא אני) יכולים לכתוב אינספור מאמרים על הנושא הספציפי הזה בלבד, אז ללא ספק יש עוד מלא חומר שאפשר ללמוד אבל גם ככה סטיתי מנושא המאמר יותר ממה שתכננתי.

מתחילים

אוקיי לקח לנו רק 12 עמודים לעבור את ההקדמה אבל הבנו ש-LSASS הוא תהליך חשוב בכל הקשור לאימות לוקאלי ודומיני. מה שכן, בסופו של דבר הוא רק חלק קטן ואינטגרלי מנושא רחב משמעותית, אז **למה להתעמק דווקא בו?** ובכן, כפי שכל הפרק הקודם מציג בצורה טובה, lsass.exe הוא בהחלט חלק מתמונה רחבה מאוד אבל הוא מרכז בתוכו המון מידע רגיש, פוליסות וסודות של sessions אקטיביים במערכת.

שאלה שחייבת להישאל היא **מדוע LSASS עושה את זה?** למה מייקרוסופט בחרו לכתוב תוכנה ששומרת את כל הסודות באופן מתמשך בזיכרון? התשובה, כמו בכל פתרון מבוסס מנגנון cache - חווית משתמש ואיכות חיים. בין אם זו גישה לקבצי רשת דרך SMB, הורדת מיילים באמצעות שרת Exchange, קבלת כתובת IP דרך שרת DHCP, נגישות אל Sharepoint - את כל אלו המשתמש מסוגל לבצע seamlessly ללא הצורך בהזדהות מול השרתים ע"י מנגנון SSO אל רוב השירותים בהם. זה אחד מהפיצ'רים שהופך את השימוש בדומיין לפרקטי.

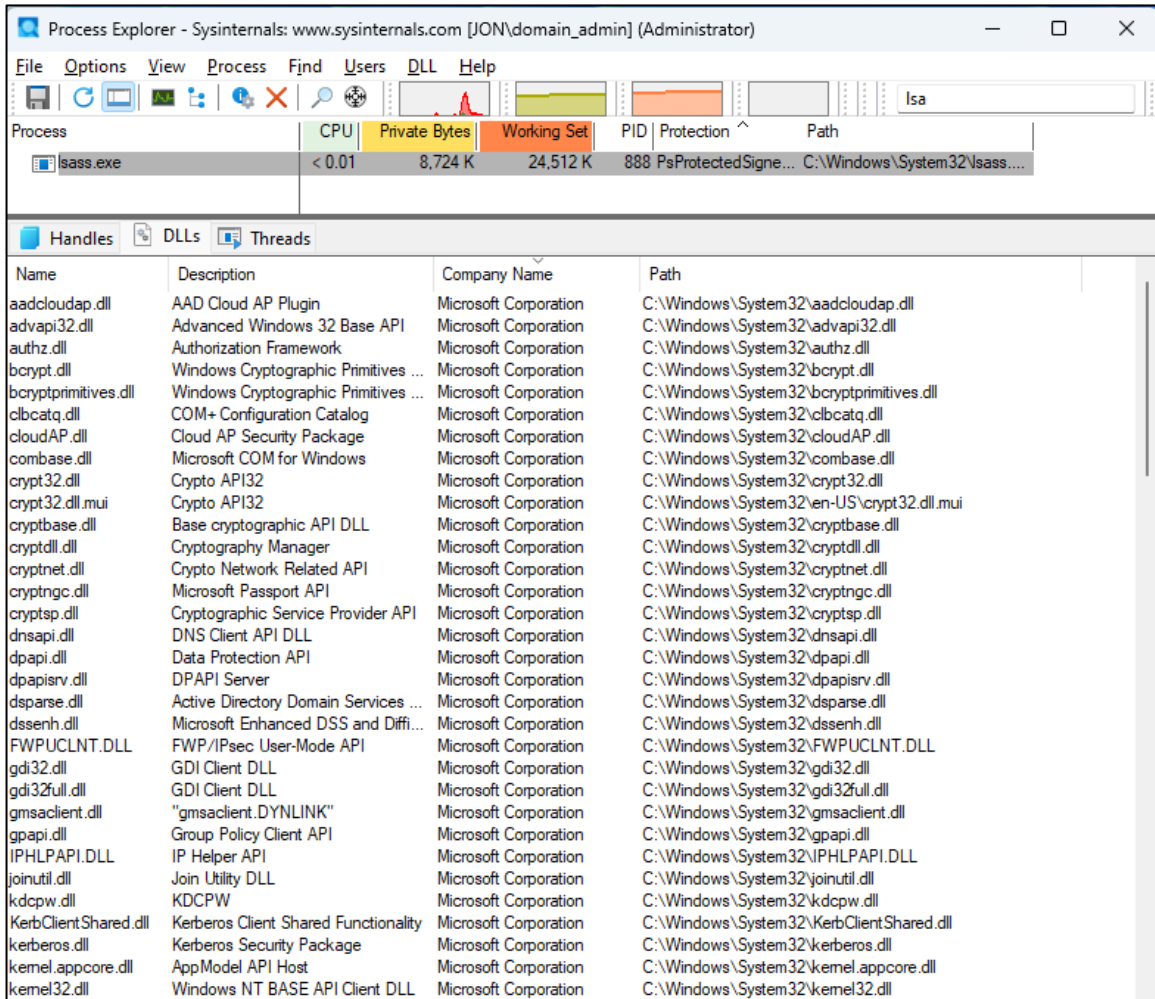
אז אילו פעולות של משתמש אקטיבי יגרמו לטירגור יצירת LSA sessions שישמרו את פרטיו כ- LSA credentials בזכרון של LSASS?

- התחברות לוקאלית (local session) והתחברות מרוחקת (RDP session) במכונת קצה.
- כל משימה שהמשתמש יריץ באמצעות RunAs.
- כל הרצה של Windows service אקטיבי על מכונת הקצה.
- הרצה של תהליך אוטומטציה (בעל הרשאות) כזה או אחר - משימה מתוזמנת או סקריפט מבוסס batch job.
- שימוש בכלים אדמיניסטרטיביים מרוחקים (כדוגמת TeamViewer, PowerShell, VNC) לשם ביצוע פעולה (בתצורה לוקאלית) על מכונת קצה.

LSASS הוא תהליך עצום עם לוגיקה מאוד מורכבת. מספיק מבט קצר על מספר ה-DLLs ה-Handles שהוא מרכז בתוכו בשביל להבין את גודל הסיפור שאנחנו ניצבים מולו.



באמצעות Process Explorer נוכל לראות את המאפיינים האלו הן בעמדת הקצה והן ב-DC עצמו:



אלה רק שליש מסך ה-DLLs. סה"כ באותו רגע היו:

עמדה	ספריות DLLs	Handles
WIN11-CLIENT01	112	275
DC01	139	981

הערה: מספר ה-handles ב-DC נמוך יחסית מכיוון שבאותו רגע חיו ברשת רק 2 מחשבי קצה עם 2 יוזרים פעילים. בסביבה אמיתית המספר גבוה משמעותית.



נקודה מעניינת ששווה להציג היא העובדה שאנחנו יכולים לראות את כל ה-DLLs של lsass.exe באמצעות Process Explorer אבל אם ננסה לבצע את אותה פעולה באמצעות Listdll (כלי נוסף בחבילת sysinternals) לא נצליח. מדוע זה אם שניהם רצים על ידי אותו משתמש ובאותם הרשאות? שמרו את השאלה, נוכל לענות עליה אחרי הפרקים הבאים.

```
Administrator: Windows PowerShell
PS C:\Users\domain_admin\Desktop\SysinternalsSuite> .\Listdlls64.exe lsass.exe

Listdlls v3.2 - Listdlls
Copyright (C) 1997-2016 Mark Russinovich
Sysinternals

Error opening lsass.exe(888):
Access is denied.
```

התמונה הגדולה

ברוב מערכות הפעלה מרובות משתמשים, האפליקציות מופרדות ברמת הריצה שלהן מקוד מערכת ההפעלה (אשר רץ בהרשאות גבוהות). המעבד של המחשב מודע לסטאטוס של כל תהליך שניכנס אליו ובהתאם מאפשר (או לא) גישה אל משאבי המערכת. על תהליך שמורשה לגשת ללב משאבי המערכת (הן בהיבטי תוכנה והן בהיבטי חומרה) נאמר כי הוא בעל הרשאות **kernel-mode** ועל תהליך המוגבל בפעולות ובממשקים העומדים לרשותו נאמר **user-mode**. בטרמינולוגיה הזו נשתמש לא מעט בפרק הבא.

בדומה למערכות UNIX גם Windows מבוססת על **מערכת הפעלה מונוליטית** בהקשר לכך שרוב הקוד של מ"ה והדרייברים משותף במרחב הזיכרון של הקרנל. קונספט שחשוב להבין הוא שלמרות שתהליכים חיים **זה לצד זה** במרחב כתובות הקרנל ותיאורטית יש לכל אחד מהם גישה ל-data structures של ראהו, זה ממש לא מקובל למשוך מידע בצורה הזו (קרי, על ידי פנייה ישירה לכתובת הזיכרון). במקום זאת, מתבצע שימוש בממשק API פורמאלי להעברת פרמטרים ובשביל לגשת לשנות מבני נתונים.

אבל (וזה אבל חשוב) עובדה זו מציבה תהליכים רגישים כדוגמת LSASS בבעיה מכיוון שכעת כל יצרן **מדפסת שאפאחד מעולם לא שמע עליו** יכול להגיש את הדרייבר שלו ולקבל גישה לאותו מרחב כתובות "מוגן" בקרנל. כמובן שזה לא כל כך פשוט אבל הבנתם את הרעיון.

את המאמר פתחנו במשפט חצי פילוסופי שמטרתו הייתה להכליל את תצורות האבטחה (4 אבני יסוד) כאשר אנו ניגשים לחפש פתרון לבעיה אבטחתית. כאשר מייקרסופט התמודדו עם הבעיה של מרחב הכתובות המשותף הם בחרו ב-2 פתרונות: **בידוד** (באמצעות Credential Guard) **ואימות** (חתימה על



תהליכים רגישים באמצעות Protected Process Light). בפרקים הקרובים אסביר על שני הנושאים האלו ובסופם נראה בדיוק כיצד Microsoft מפעילה את הצמד בשביל להגן על LSASS.

מאבטחים רק את מי שצריך - Protected Process Light

תהליכים רגישים ב-Windows רצים ברמת context מיוחדת הידועה בשם "Protected Process". תהליכים אלה מיועדים להיות מאובטחים יותר מתהליכים "רגילים"; מגוון האופרציות שהם יכולים לבצע דל משמעותית ויש עליהם מגבלות נוספות שלא חלות על תהליכים אחרים. העובדה שלא היינו רוצים ששירותי Microsoft Defender (MsMpEng.exe) ירוץ בצורה דומה אל שירותי ה-Fax (fxssvc.exe) היא ההמחשה הכי פשוטה לצורך הנ"ל.

Well, מודל אבטחת המידע של Windows מתבסס על העובדה שכל תהליך הרץ עם token בעל הרשאות debug (כמו זה שיש למשתמש Administrator) יוכל לבקש (ולקבל) גישה לכל תהליך הרץ במ"ה. הוא יוכל לקרוא ולכתוב לזיכרון התהליך, להזריק קוד, להשעות ולחדש ריצה של threads ולתשאל מידע על תהליכים אחרים. בצורה כזו תהליכים כמו Process Explorer ו-Tasks Manager עובדים בשביל למנף את הפונקציונאליות שלהם עבור המשתמש. ברור לנו שמכניזם זה מתנגש חזיתית עם מודלי אבטחת מידע נוספים בפלטפורמה.

המקור של Protected Process (PP) התחיל לפני די הרבה זמן עוד ב-Windows Vista / Server 2008 אבל שם כל המנגנון שתיארתי בפסקה מעל היה הרבה יותר הָדוּק ומנע פונקציונאליות רבה ברמת מערכת ההפעלה והמשתמש. למעשה המטרה שלשמה הוא הגיע לא הייתה בכלל להגן על credentials אלא יותר בכיוון של למנוע מהמשתמש לגנוב מידע מדיסקים עם זכויות יוצרים באמצעות נגן DVD פירטי.

לכן, ב-Windows 8.1 / Server 2012 R2 הוצג מנגנון חדש - Protected Process Light (PPL). מטרתו הייתה פשוטה - שחרור חלק מהמגבלות על אילו קבצי DLL מותרים לטעינה לתהליך מוגן והצגת מכניזם חתימה ברמות שונות עבור קובץ ה-executable הראשי.

ב-PPL קוד הרץ ב-user-mode (ואפילו אם הוא רץ בהרשאות גבוהות) לא יכול לחדור באמצעות הזרקת threads או השגת מידע על ה-DLLs שאותם תהליכים מוגנים טוענים. עם זאת, המודל של PPL מוסיף מימד חדש לכל הסיפור - attribute values. בסופו של יום, **עולם אבטחת המידע מתכנס לאמון**. בדיוק כמו בעולם ה-web המבוסס על TLS בו אנחנו (קרי, הדפדפנים שלנו) חייבים לסמוך על DigiCert Global Root CA בעיניים עצומות בשביל [לעשות עסקים באינטרנט](#), כך גם בעולם התשתיות והדומיין - **חייבת להיות נקודת מוצא של אמון** - Root of Trust; חלק מאותה נקודה הם התהליכים המוגנים.

בעוד שב-PP נעשה שימוש במכניזם בינארי (חתום \ לא חתום) ב-PPL מתבצע שימוש במודל עם 8 רמות. PPL ברמה אחת יכול לפתוח כל תהליך באותה רמת חתימה (או רמה נמוכה יותר) עם גישה מלאה אבל



מוגבל מאוד במידה והתליך המוגן עם חתימה ברמה גבוהה יותר. הטבלה הבאה מסכמת את הרעיון בצורה טובה:

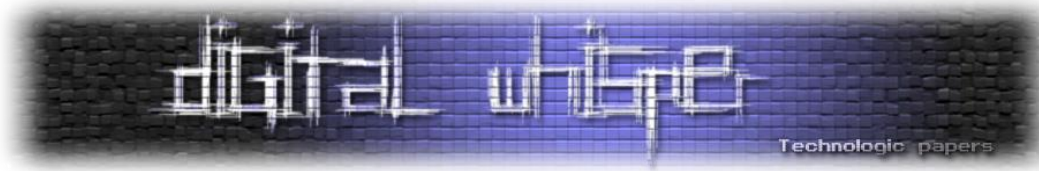
Signer Name (PS_PROTECTED_SIGNER)	Level	Used For
PSProtectedSignerWinSystem	7	System and minimal processes (including Pico processes).
PSProtectedSignerWinTcb	6	Critical Windows components. PROCESS_TERMINATE is denied.
PSProtectedSignerWindows	5	Important Windows components handling sensitive data.
PSProtectedSignerLsa	4	Lsass.exe (if configured to run protected).
PSProtectedSignerAntimalware	3	Anti-malware services and processes, including third party. PROCESS_TERMINATE is denied.
PSProtectedSignerCodeGen	2	NGEN (.NET native code generation).
PSProtectedSignerAuthenticode	1	Hosting DRM content or loading user-mode fonts.
PSProtectedSignerNone	0	Not valid (no protection).

PPL חתומים על ידי ישויות שונות במערכת ההפעלה עם רמת Trust שונה; כאשר 0 מציין את הרמה הנמוכה ביותר של חתימה (אין חתימה) ו-7 מייצג את הרמה הגבוהה ביותר (תהליכי מערכת). אם נסתכל [בדוקומנטציה](#) של Microsoft בנושא מבנה נתונים של PS_PROTECTION נוכל לראות כיצד אותה רמת הגנה (קרי, חתימה) מיוצגת ב-EPROCESS בקרנל:

```
typedef struct _PS_PROTECTION {
    union {
        UCHAR Level;
        struct {
            UCHAR Type : 3;
            UCHAR Audit : 1;           // Reserved
            UCHAR Signer : 4;
        };
    };
};
} PS_PROTECTION, *PPS_PROTECTION;
```

למרות שהוא מיוצג כמבנה (struct), כל המידע מאוחסן בשתי nibbles של byte בודד אחד. 3 הביטים הראשונים מייצגים של סוג החתימה (None, Light, Full), כלומר האם התהליך PP או PPL או כלל לא מוגן. ארבעת הסיביות האחרונות מייצגות את סוג החתימה. הייצוג של אותם enum-ים מוגדר בצורה הבאה:

```
typedef enum _PS_PROTECTED_TYPE {
    PsProtectedTypeNone = 0,
    PsProtectedTypeProtectedLight = 1,
    PsProtectedTypeProtected = 2
} PS_PROTECTED_TYPE, *PPS_PROTECTED_TYPE;
```

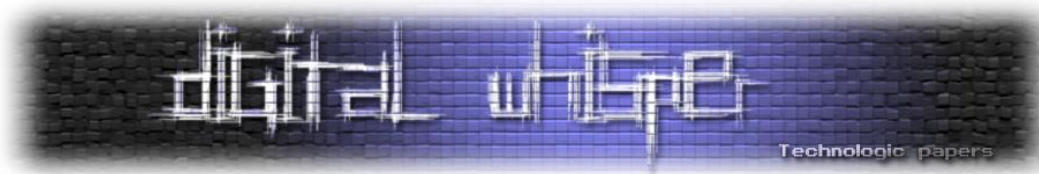



```
typedef enum _PS_PROTECTED_SIGNER {
    PsProtectedSignerNone = 0, // 0
    PsProtectedSignerAuthenticode, // 1
    PsProtectedSignerCodeGen, // 2
    PsProtectedSignerAntimalware, // 3
    PsProtectedSignerLsa, // 4
    PsProtectedSignerWindows, // 5
    PsProtectedSignerWinTcb, // 6
    PsProtectedSignerWinSystem, // 7
    PsProtectedSignerApp, // 8
    PsProtectedSignerMax // 9
} PS_PROTECTED_SIGNER, *PPS_PROTECTED_SIGNER;
```

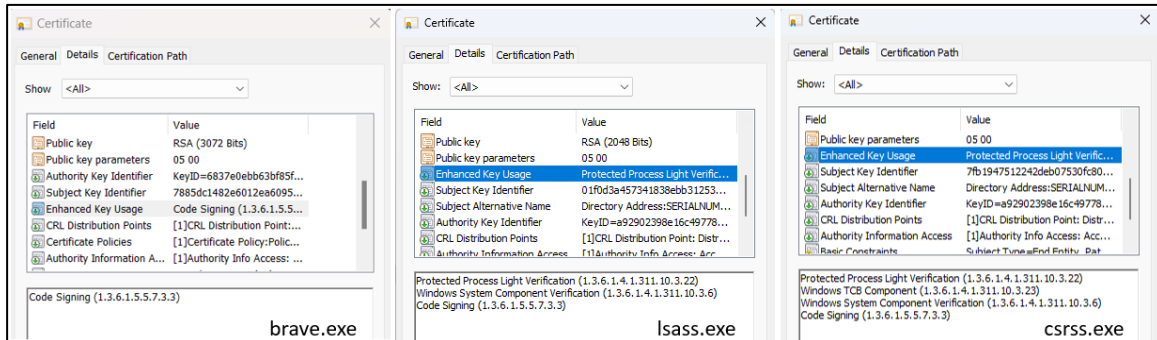
כאמור הצגתי כי בטלה הראשית ישנן 8 רמות של "חתימות" בסדר עולה. כל רמה מוגדרת באמצעות קומבינציה של שני הערכים הללו (PS_PROTECTED_SIGNER + PS_PROTECTED_TYPE):

Protection level	Value	Signer	Type
PS_PROTECTED_SYSTEM	0x72	WinSystem (7)	Protected (2)
PS_PROTECTED_WINTCB	0x62	WinTcb (6)	Protected (2)
PS_PROTECTED_WINDOWS	0x52	Windows (5)	Protected (2)
PS_PROTECTED_AUTHENTICODE	0x12	Authenticode (1)	Protected (2)
PS_PROTECTED_WINTCB_LIGHT	0x61	WinTcb (6)	Protected Light (1)
PS_PROTECTED_WINDOWS_LIGHT	0x51	Windows (5)	Protected Light (1)
PS_PROTECTED_LSA_LIGHT	0x41	Lsa (4)	Protected Light (1)
PS_PROTECTED_ANTIMALWARE_LIGHT	0x31	Antimalware (3)	Protected Light (1)
PS_PROTECTED_AUTHENTICODE_LIGHT	0x11	Authenticode (1)	Protected Light (1)

נקודה שחשוב להבין היא שלכל (!) תהליך של מ"ה יש תעודה חתומה של מייקרוסופט. במידה ומדובר באפליקציה (כגון דפדפן Brave.exe) החתימה תהיה על ידי CA authority כמו DigiCert. בתעודה, תחת השדה של Enhanced Key Usage (EKU) מופיע עבור כל תהליך מוגן כיצד הוא חתום ומה ההרשאות שלו.



על ידי שימוש ב-Properties על קובץ ה-exe של התהליך עצמו או באמצעות Task manager האהוב ניתן לצפות בכל החתימות הדיגיטליות של התהליכים. לשם הדוגמה נסתכל על 3 תהליכים (brave.exe, lsass.exe ו-csrss.exe) ברמות חתימה שונות, קל לראות את ההבדל בשדה EKU:

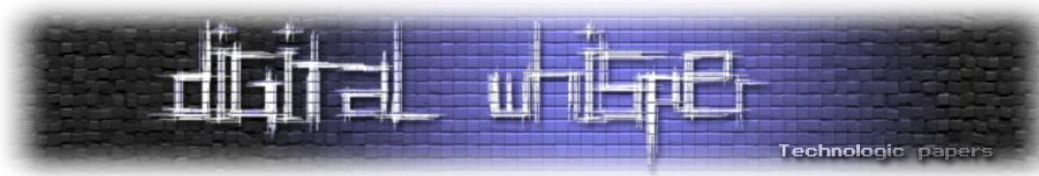


בנוסף, ה-protection level של התהליך גם תשפיע על אילו DLLs הוא מורשה לטעון (בשביל למנוע hijacking). כיצד זה קורה? כאשר תהליך מתחיל לרוץ נשמר לו ב-EPROCESS שדה של SignatureLevel וכאשר הוא בא לטעון DLL מתבצעת בדיקה ב-table lookup כלשהי לבדיקה האם ה-DLL הוא ב-level signature המתאים. זה נבדק על ידי מודל ה-Code Integrity באותה צורה בה ה-executable הראשי נבדק. ניתן לראות באמצעות משתמש חזק את התהליכים המוגנים באמצעות Process Explorer תחת עמודת "Protection":

Process	CPU	Private Bytes	Working Set	PID	Protection	Path
System	< 0.01	44 K	2,364 K	4		
Registry		5,808 K	34,320 K	92		[A device attached to the syst...
Memory Compression		500 K	70,884 K	2948		[A device attached to the syst...
SgmBroker.exe		5,048 K	8,520 K	10116	PsProtectedSignerWinTcb	[Access is denied.]
wininit.exe		1,312 K	6,312 K	704	PsProtectedSignerWinTcb-Light	[Access is denied.]
smss.exe		1,092 K	1,124 K	440	PsProtectedSignerWinTcb-Light	[Access is denied.]
services.exe		5,716 K	13,244 K	848	PsProtectedSignerWinTcb-Light	[Access is denied.]
csrss.exe		1,832 K	5,532 K	628	PsProtectedSignerWinTcb-Light	[Access is denied.]
svchost.exe	< 0.01	2,072 K	22,796 K	712	PsProtectedSignerWinTcb-Light	[Access is denied.]
svchost.exe		1,692 K	7,104 K	1328	PsProtectedSignerWindows-Light	[Access is denied.]
svchost.exe		2,508 K	9,928 K	9896	PsProtectedSignerWindows-Light	[Access is denied.]
SecurityHealthService.exe		3,632 K	14,764 K	8128	PsProtectedSignerWindows-Light	[Access is denied.]
lsass.exe		9,028 K	26,004 K	900	PsProtectedSignerLsa-Light	[Access is denied.]
NisSrv.exe		3,564 K	10,488 K	7488	PsProtectedSignerAntimalware-Light	[Access is denied.]
MsMpEng.exe	2.16	237,300 K	152,020 K	3724	PsProtectedSignerAntimalware-Light	[Access is denied.]
WUDFHost.exe	< 0.01	2,648 K	8,404 K	972		C:\Windows\System32\WUD...
WmiPrvSE.exe		11,968 K	27,056 K	4224		C:\Windows\System32\wbem...
winlogon.exe		2,320 K	10,288 K	772		C:\Windows\System32\winlog...

אם בתור משתמש חלש ננסה לראות את ה-DLLs של תהליך מוגן לא נמצא כלום, למעשה לא נוכל לראות שהתהליך מוגן בכלל. זה בגלל של Process Explorer עצמו עושה שימוש ב-APIs של user-mode על מנת לתשאל את המודלים שטעונים כרגע ובשביל זה נדרשת רמת גישה שלא מותרת על תהליכים מוגנים (במידה ואתם רצים תחת משתמש חזק ההרצה מתבצעת בצורה אחרת).

השיטות הפופולאריות בעבר להוצאת מידע מ-LSASS היו לטעון דרייבר או LSA plug-ins זדוניים, להצמיד אליו debugger ולהתבונן בקריאות שהוא מבצע. בדיוק כן זורח השימוש ב-lsass.exe כ-PPL. לא ניתן לדבג PPL בזמן ריצה בכלל. שיטה נוספת שקצת יותר קשה לתת לה מענה היא לטעון דרייבר ישן (וחתום) שידוע כי יש בו פגיעות.



אוקיי" אוקיי" יונתן השתכנענו שאנחנו רוצים ש-*lsass.exe* יוגדר כ-PPL, כיצד עושים את זה? אני אחלק את התשובה שלי ל-2 - אחת ברמת מחשב בודד באמצעות ה-registry או Local Group Policy (מיוחס אל ל-Windows 11 בלבד) והשנייה ברמת הדומיין באמצעות פוליסת GPO שאפיץ לכל מחשבי הקצה. נתחיל:

הגדרת PPL במחשב בודד דרך ה-Registry

- **אנו -** ניכנס למפתח שאחראי על LSA בריגסטרי:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa
```

- **דו"ס -** נגדיר את הערכים הבאים במפתח בצורה הבאה:

○ `"RunAsPPL"=DWORD: 1` במידה ובא לנו שיתבצע שימוש במשתנה תלוי UEFI.

○ `"RunAsPPL"=DWORD: 2` במידה ולא בא לנו UEFI.

אני ממליץ לעשות שימוש ב-UEFI (וגם ב-Secure Boot) כי אם לא אז ת'כלס שום דבר לא מונע מתוקף שהשיג גישה לעמדה לשנות בעצמו את הערכי הרגיסטרי.

- **טקס -** לבצע `reboot`.

○ אלטרנטיבית, אפשר באמצעות CLI של PS בצורה הבאה:

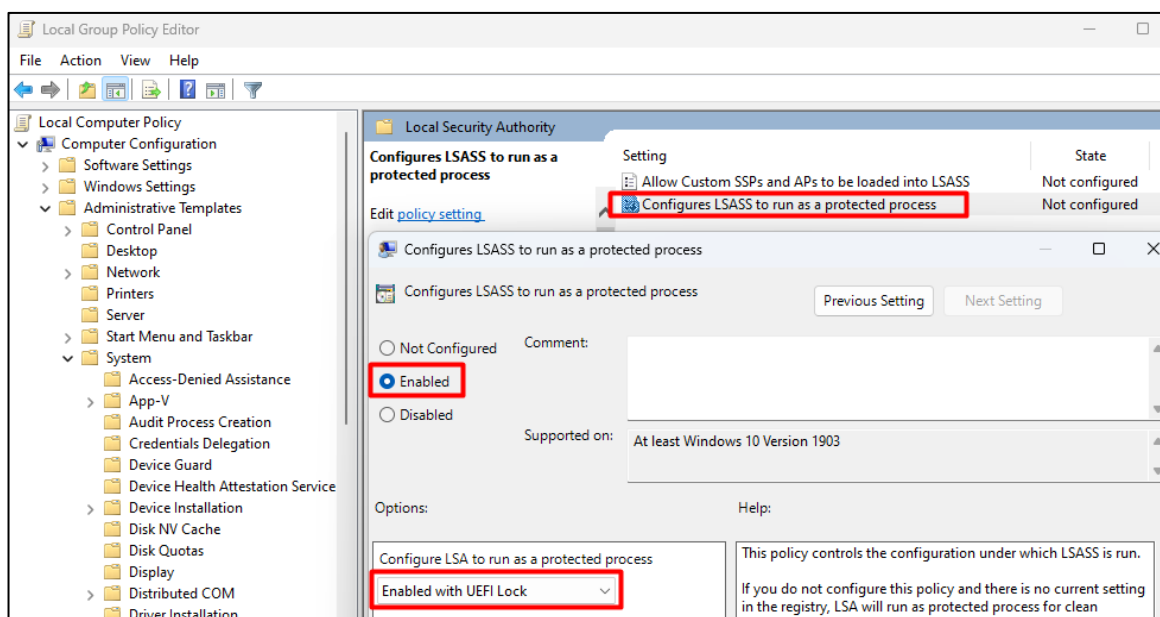
```
reg add "HKLM\SYSTEM\CurrentControlSet\Control\Lsa" /v "RunAsPPL" /t REG_DWORD /d 1 /f
```

הגדרת PPL במחשב בודד באמצעות LGPO

נפתח את `gpedit.msc` ניגש אל:

Computer Configuration → Administrative Templates → System → Local Security Authority

ונגדיר את החוק "Configures LSASS to run as a protected process" בצורה הבאה:



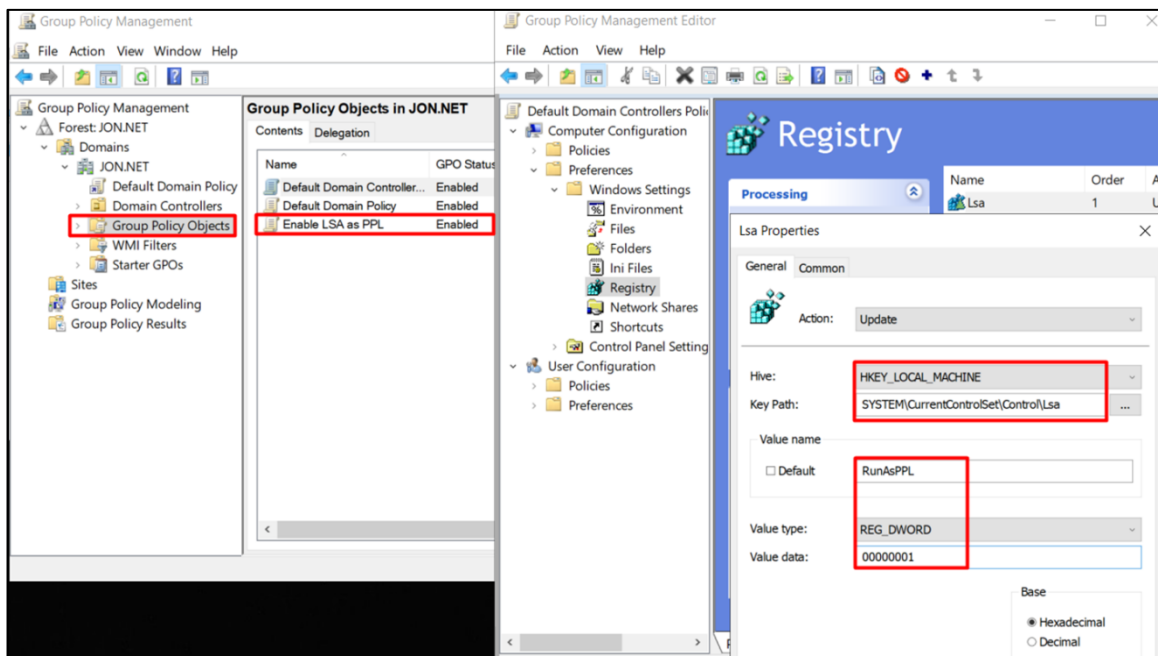
ריסטרט אחד וסיימנו.

גדרת PPL בדומיין באמצעות GPO

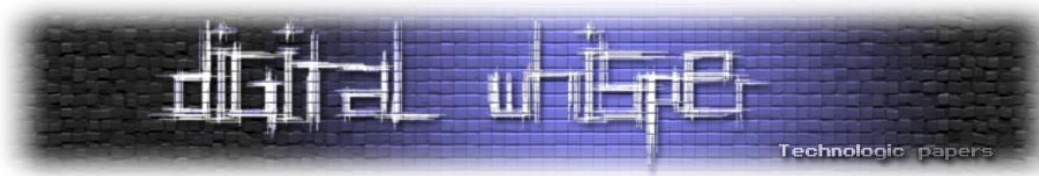
אם הבנתם את השלב הקודם אזי כנראה שלא תהיה לכם בעיה להבין את השלב הנוכחי כי הוא למעשה הצעד השני באינדוקציה. כלומר, כעת פשוט נגדיר באמצעות חוקת GPO את השינוי ערכים ברגיסטרי של כלל המחשבים.

ב-DC שאחראי על ניהול האובייקטים בדומיין נפתח את קונסולת Group Policy Management וניצור חוקת GPO חדשה שמקושרת ברמת הדומיין (או אל OU) שמכילה את כלל המחשבים (אפשר גם להתלבש על חוקה שכבר קיימת אליהם אבל אנחנו אוהבים סדר אז ניצור אחת חדשה).

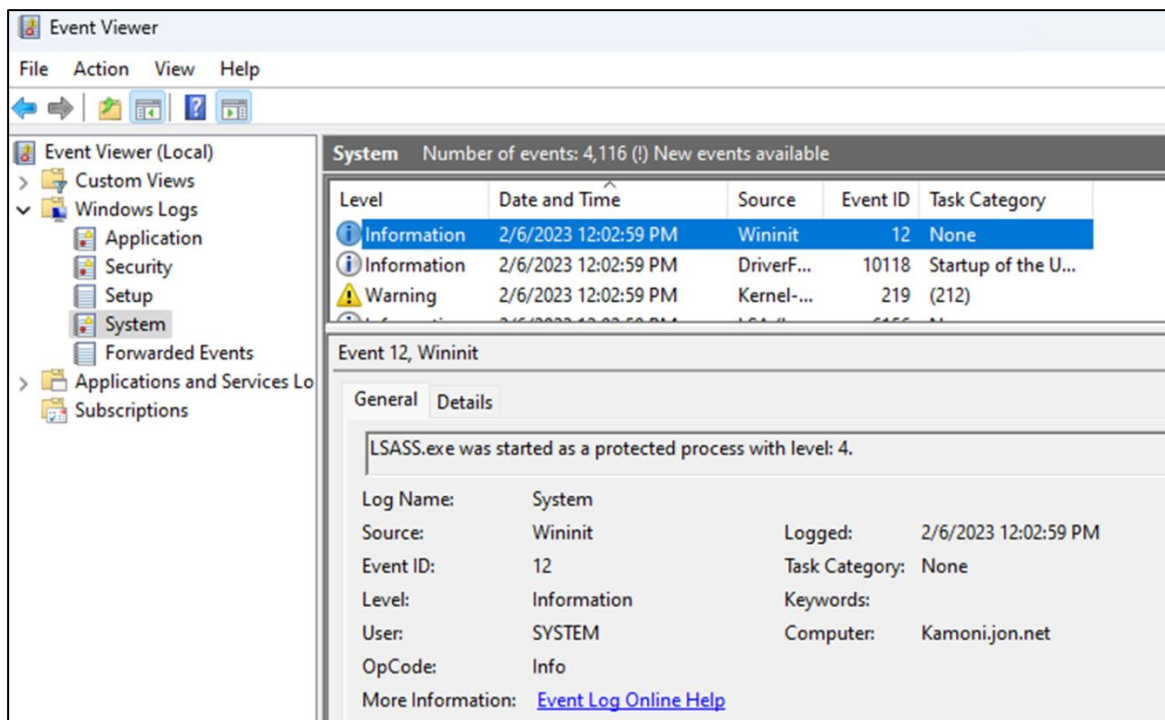
לאחר מכן נערוך אותה כך שתשנה את הערכים ברגיסטרי בהתאם למה שהראינו בהסבר מקודם. תמונה שווה אלף מילים אז נסתפק בתמונה הבאה:



אם המכונה מופעלת עם Secure Boot הערך עצמו יהיה מאוחסן בקושחה ולכן אותו ערך יהיה פעיל ללא קשר לשינויים ברגיסטרי עצמו.



אם הגדרתם את השימוש בו ואתם רוצים לבדוק ש-LSA Protection עובד, ניתן לברר אם הוא עולה באמצעות Event Viewer:



אפשר לראות כי ה-protection level שניתן ל-lsass.exe היא 4 (PsProtectedSignerLsa) לפי הטבלה הראשית ממקודם). במאמר המשך אציג יותר לעומק כיצד Mimikatz עוקפת את מנגנון PPL (אמל"ק - אחת הדרכים היא טעינת דרייבר חתום [mimidrv.sys] ואז באמצעות ה-service שלו הורדת ה-PPL של Lsass.exe. דרך אגב זה לא עובד יותר כי מייקרוסופט גנזו את התעודה שלו [דה..] אבל יש כלים אחרים שעושים שימוש בתוכנות כמו procexp64.exe בשביל לבצע משהו דומה).

אבל רגע רגע! לפני שאתם רצים להטמיע את השינויים האלו בכל הסביבה הארגונית שלכם, קיים שיקול שחייבים לקחת אותו בחשבון והוא **האם אתם משתמשים במודל אימות של צד שלישי?** בפרק על SSPi הסברנו שקיימים Authentication modules שניתן להוסיף.

במידה והארגון שלכם עושה שימוש באחד כזה, הוא **חייב** להיות חתום דיגיטלית עם החתימה של Microsoft (לא נתון הכי טריוויאלי).

כמו כן, חשוב לציין כי [בדוקומנטציה](#) של Microsoft בנושא אבטחת LSA הם מציינים בנוסף גם כמה פעולות לביצוע Audit וניטור אחר פעולות חשודות (כמו טעינת דרייברים אל lsass.exe). אני חושב שזה חסר טעם להעמיק בנושא במאמר כי כנראה לכל מערכת SIEM כבר יש 100 חוקות בנושא.



Third-party PPLs

ניתן להגדיר כי תהליכים נוספים (שלא נכתבו על ידי מייקרוסופט) יהיו גם חתומים בתצורת PPL. תוכנות anti-malware (AM) ותוכנות המבצעות דיפלוימנט בהיקף רחב הן המחשה טובה לצורך הנ"ל. לרוב, מוצר AM יהיה מורכב מ-3 מרכיבים:

- קרנל דרייבר שמפרש את בקשות ה-I/O למערכת הקבצים ו\או הרשת, ומיישם מנגנוני חסימה בהתאם ל-traffic שהוא קולט.
- שירות user-space (לרוב שרץ תחת משתמש עם הרשאות חזקות) שיכול לערוך את חוקות הדרייבר, לקבל הודעות מהדרייבר בנוגע ל-events ולתקשר עם שרת לוקאלי או האינטרנט.
- תהליך GUI ב-user-mode שמתקשר עם המשתמש להעברת מידע ומאפשר למשתמש לקבל החלטות. כלומר, אותו פאנל גדול כזה עם pie charts שמעדכן תמונות סטטוס שכולנו מכירים.

הסיבה להוספתם אל משפחת ה-PPL ברורה - גם במצב בו משתמש זדוני השיג הרשאות גבוהות, הוא לא יוכל להפסיק את ריצת ה-AM ו\או להזריק לה קוד (לפחות כל עוד הוא לא עושה שימוש באקספלויטים ברמת הקרנל). כמובן שעל מנת לקבל את האישור של מייקרוסופט הם נדרשים לעבור תהליך אימות מנהלתי כאשר בסופו שדה ה-EKU בתעודה יכול את שדה ה-PPL של:

`PS_PROTECTED_ANTIMALWARE_LIGHT (0x31)`

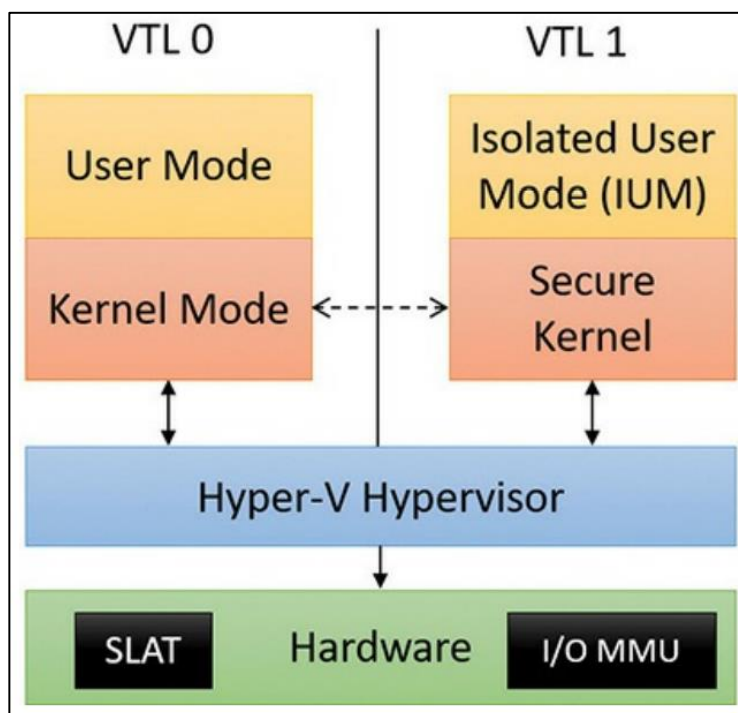
שכאמור נמצא ברמה אחת לפני האחרונה. ולכן גם במידה והשתלטו על תוכנת ה-AM, ההרשאות שלה יהיו יחסית נמוכות ביחס ל-PPLs אחרים (לשם דוגמה, היא לא תהיה מסוגלת לבצע dump אל lsass.exe).

חלוקה לשני עולמות - Virtual Trust Levels

על מנת להסביר על Credential Guard אנחנו צריכים לעשות עיקוף קטן ולהסביר קודם על Virtual Trust Levels (רמות אמון וירטואליות בעברית שבורה) בקרנל(ים) של Windows. ה-Hyper-V של מיקרוסופט הוא חלק בלתי נפרד מתהליך עליית מערכת ההפעלה ואף נטען לפני הקרנל (לעומת מימושים אחרים בהם ה-Hypervisor רץ כולו בקרנל). למעשה, חלקכם אולי יופתעו לגלות כי עקב היותו Hypervisor type 1, מערכת ההפעלה Windows שכולנו מכירים רצה כמכונה וירטואלית בצורה דומה לכך שבאמצעות WSL2 ניתן להריץ מכונת Ubuntu על גבי התשתית של Hyper-V. ליתר דיוק, כל מחיצה ב-Hyper-V מהווה **יחידה לוגית המיישמת מכונה וירטואלית מבודדת** משאר המכונות. לכל מחיצה יש מעבד וירטואלי שמנהל את ריצת התהליכים - מה שגורר ניהול context switch ברמת המעבד האמיתי של ה-Hyper-V עצמו. הגמישות שרמות אבסטרקציה מאפשרות הוא אינסופי.

המאמר [Hyper-V Architecture for Security Researcher](#) מאת דני אודלר בגליון 114 מדבר בדיוק על זה. שווה מאוד את הקריאה למי שרוצה להבין לעומק כיצד Hyper-V עובד. רמות אמן וירטואליות מדברות על אימפלמנטציה של 2 שכבות אבסטרקציה שה-Hypervisor מיישם עבור eco-system של קוד ברמות אבטחה ואמון שונות - **VTLO & VTL1**. בעברית פשוטה, מערכת ההפעלה והרכיבים שלה רצים ב-VTL0 בעוד שטכנולוגיות כדוגמת VBS (Virtual Based Security) רצות ב-VTL1 עם יכולות שונות לחלוטין. שום קוד ב-VTL0, אפילו אם הוא של הקרנל עצמה, לא יוכל להשפיע על התוכנות שרצות תחת VTL1 מכיוון שהוא לא מסוגל להיות מודע לקיומן.

קוד שרץ ב-VTL0 ישאר ב-"מימד" הזה ויוכל להשפיע רק על תהליכים שרצים באותו מימד. מצד שני, חשוב להבין ש-VTL1 מכיל את קוד הקרנל משלו שרץ ב-processor mode עם הרשאות גבוהות (ring 0) ב-x86/x64 וגם, בצורה דומה, אזור user-mode (שנקרא **IUM** - Isolated user mode) קיים. ניתן לחשוב על VTLs כפתרון אורתוגונאלי אל רמות ההשראות שבמעבד: user mode ו-kernel mode **בתוך כל VTL** וה-hypervisor מנהל את ההרשאות לאורך כל ה-VTLs.



[מקור - [Window internals](#)]

המון מילים שוברות שיניים נאמרו עד כה אז נקפוץ ישר לפרקטיקה בשביל להמחיש את הנושא - הקרנל המאובטחת של VTL1 מכילה בינאריים משלה ויושבת בדיסק בקובץ **securekernel.exe**. בצורה דומה, שחקן נוסף ששווה להזכיר בכל הקשור ל-VTL1 הוא **lumdll.dll** שמקביל אל **Ntdll.dll** (ב-VTL0) ומאפשר ביצוע system calls עבור הקרנל של VTL1.

נקודה ששווה לתת עליה את הדגש היא שלפי השרטוט ניתן לראות כי אין אפשרות בכלל לתקשר בין kernel mode ב-VTL0 אל user-mode ב-VTL1 (בעל הרשאות גבוהות יותר). עם זאת, גם הצד ההפוך נכון - אי



אפשר לגעת עם קוד מבוסס kernel mode ב-VTL1 בקוד user mode ב-VTL0 מכיוון שהחוקיות של משתמש (ring 3) לא יכול לגעת בקוד קרנל (ring 0) נשמרת.

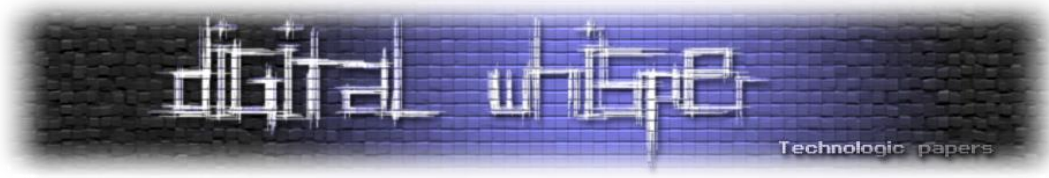
דרך טובה להבין את כל הבאלגן הזה בטרמינולוגיה היא להסתכל על כל עניינין של ה-VTLs בפרספקטיבה של בידול ולא מפרספקטיבה של כוח. אפליקציית user-mode ב-VTL1 לא יותר חזקה מקוד אפליקציה או דרייבר של VTL0 אלא היא מבודדת ממנו.

למעשה, שם אלטרנטיבי בדוקומנטציה ל-secure kernel של VTL1 הוא "proxy kernel". כלומר, לא רק שיישומי VTL1 אינם חזקים יותר מיישומי VTL0, במקרים רבים יש להם הרבה פחות יכולות. מה הכוונה? ובכן, מכיוון שהקרנל של VTL1 לא מיישמת מגוון שלם של יכולות מערכת, היא הלכה למעשה בוחרת ידנית אילו system calls היא תעביר לקרנל של VTL0. כל סוג של פעולות I/O, כולל קבצים, רשת ו-registry אסורים לחלוטין. מה עם פעולות גרפיקה? פפפפ הצחקתם, אינן באות בחשבון; לשום דרייבר לא מותר לתקשר ולבנות GUI. קיצר, ב-VTL1 הכל מתבצע ומתנהל בתצורה סטטית.

ה-secure kernel, על ידי ריצה גם ב-VTL1 וגם מהיותה kernel mode למעשה בעלת רמת אמון גבוהה מאוד. משם כך, יש לה גישה מלאה לזיכרון ולמשאבים של VTL0. הקרנל, באמצעות ה-hypervisor, יכולה להגביל את הגישה מערכת ההפעלה של VTL0 אל מרחב כתובות זיכרון על ידי מינוף תמיכת חומרת ה-CPU. פעולה זאת נקראת בדוקומנטציה Second Level Address Translation, או SLAT בקיצור (אם תשאלו אותי היו צריכים לשנות את הביטוי אל Second Level Unconstrained Translation, ללא ספק ראשי תיבות היו מעניינים יותר). SLAT הוא (היא?) הבסיס של טכנולוגיית Credential Guard שמאפשרת לשמור סודות במרחב כתובות כזה. נרחיב על Credential Guard בפרק הבא ונראה כיצד הוא משתמש באותה וירטואליזציה בשביל להגן על LSASS.

כדי למנוע מדרייברים רגילים למנף התקני חומרה ולגשת ישירות לזיכרון, נעשה שימוש בחומרה נוספת המכונה I/O memory management unit (MMU), אשר עושה וירטואליזציה לגישה לזיכרון עבור התקנים. זה יכול לשמש כדי למנוע התקני דרייברים להשתמש בגישה ישירה לזיכרון (DMA), כדי לגשת ישירות ל-hypervisor או לאבטח את אזורי הזיכרון הפיזיים של הקרנל. בכך למעשה יעקפו את SLAT מכיוון שלא מעורב זיכרון וירטואלי כלל בתהליך.

מכיוון שתהליכי user-mode הפועלים ב-VTL1 מבודדים, קוד זדוני פוטנציאלי עשוי לנסות לבצע system calls (שיאפשרו לו לחתום על סודות ובכך להיראות לגיטימי), וכן עלול לגרום לאינטראקציות זדוניות עם תהליכי VTL1 אחרים או עם ה-secure kernel. ככזה, רק מחלקה מיוחדת של קבצים בינאריים חתומים במיוחד, הנקראת Trustlets, מותרת להפעלה ב-VTL1. לכל Trustlet יש מזהה וחתומה ייחודיים, ולקרנל יש ידע מקודד של אילו Trustlets נוצרו עד כה. בצורה כזו, אי אפשר ליצור Trustlets חדשים ללא גישה לקרנל (שכאמור רק מייקרוסופט יכולה לשחק איתו), ולא ניתן לבצע patch אל Trustlets קיימים בשום צורה (מכיוון שהפעולה תבטל את החתימה).



Windows Defender Credential Guard

אם בעבר היה ניתן לשמור את סודות המשתמש רק בזכרון LSASS כעת על ידי שימוש בוירטואליזציה המצב קצת שונה. כאשר Credential Guard מופעל במכונת Windows מעתה ואילך יהיו 2 תהליכי LSASS - הרגיל, lsass.exe, זה שאנחנו מכירים ותהליך מבודד נוסף, lsaiso.exe, אשר רץ בתוך מכונה וירטואלית של Hyper-V. כלומר תוקף **יצטרך לעבור את ה-hypervisor** במידה והוא רוצה לגשת לנתונים של המשתמש (ולעשות dump ל-credentials) - וזו משימה אחרת לגמרי מאשר תקיפה של תהליך בודד.

מכיוון ש-LSAIsso מסוגל לתקשר רק עם מספר בינאריים מאוד מצומצם של מערכת ההפעלה שחתומים על ידי Microsoft בלבד והוא לא משתמש בשום דרייבר חיצוני, משטח התקיפה עליו מצטמצם משמעותית.

כמו כן, כאשר מפעילים את Credential Guard ניתן לבצע SSO רק עם Kerberos. זה עתיד ליצור לא מעט אי-נוחות מכיוון שחבילות אימות אחרות כגון NTLMv1, CredSSP ו-WDigest (עליהן הסברנו מקודם מפרק של SSP) נחסמות לשמירת Credentials ולכן לא ניתן לבצע SSO באמצעות הפרוטוקולים שהן מייצגות (אבל כן ניתן להזדהות באמצעותן). בנוסף, פרוטוקול Kerberos לא מאפשר יותר מעבר של unconstrained delegation וגם במצב PKINIT הוא ישתמש ב-RSA במקום ב-Diffie-Hellman. אך עם זאת, אימות מבוסס תעודות (כמו פרוטוקולים EAP-TLS ב-802.1x) מתאפשר ו-Credential Guard לא חוסם כלל.

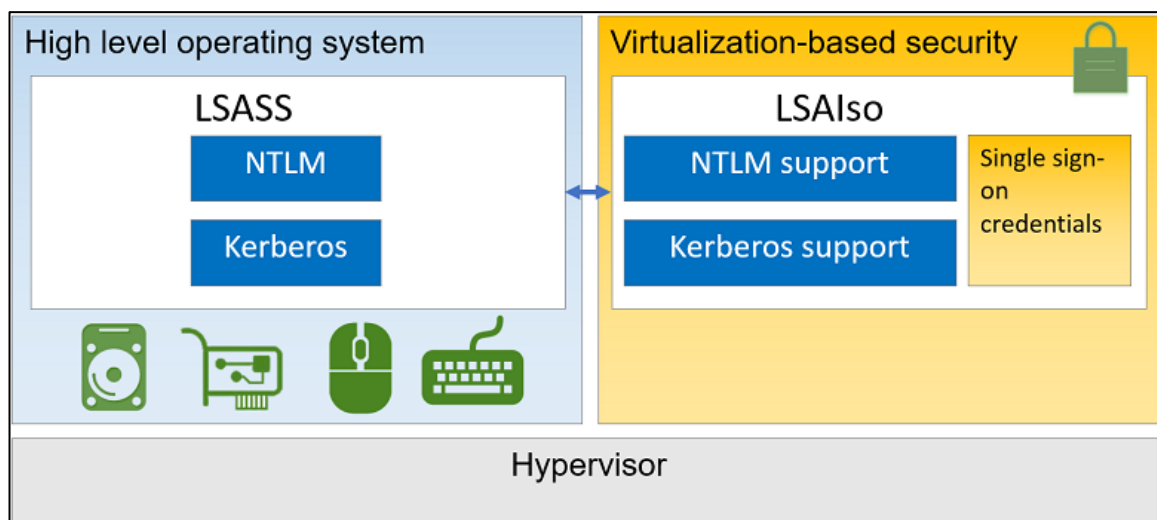
מכניזם נוסף ששווה להרחיב עליו בכמה מילים הוא **Device Guard**. בעוד ש-Credential Guard מתעסק ברובו בהגנת ה-Credentials של המשתמש (ומשם גם מגיע השם שלו), Device Guard מגיע במטרה להגן על כלל מערכת ההפעלה.

הוא מבוסס מנגנון whitelisting של יישומים מבוססי חומרה כדי להגן על devices (ומשם מגיע השם שלו) מפני תוכנות זדוניות ואיומים אחרים. הוא משתמש בשילוב חוקות מבוססות תוכנה בשביל להבטיח שרק אפליקציות מורשות יוכלו לרוץ במערכת ההפעלה ועם Secure Boot ו-VBS

הערה: בחלק מהדוקומנטציות מתייחסים אל VBS בתור VSM. למרות שקיים שוני טרמינולוגי קטן בין השניים, במהלך המאמר אתייחס אליהם בצורה זהה תחת השם VBS מכיוון שהוא השם הכללי של הטכנולוגיה.

דוגמה פרקטית ל-Device Guard היא היכולת שלו למנוע טעינה לזכרון של DLL-ים לא חתומים או untrusted ובכך למנוע DLL injection.

תמונה מפורסמת שחוזרת כמעט בכל בלוג אבטחת מידע שמתעסק ב-Credential Guard לקוחה מהדוקומנטציה של Microsoft בנושא:



[מקור - Microsoft]

מדובר באיור הממחיש ב-high level כיצד מתבצע הבידוד של LSASS באמצעות ה-hypervisor. בפרק הבא כאשר נדבר על נקודתית על הפתרונות, ניכנס יותר לעומק של המנגנון וכיצד הוא "מגן באמת" על נתוני המשתמש. אז כיצד מגדירים את השימוש ב-Credential Guard? בדומה ליישום ב-PPL, גם כאן אני אחלק את התשובה שלי ל-2 - אחת ברמת מחשב בודד באמצעות ה-registry או Local Group Policy והשנייה ברמת הדומיין באמצעות פוליסית GPO שאפיץ לכל מחשבי הקצה.

חשוב לציין כבר מעכשיו שהחל מ-Windows 11 22H2 המנגנון מאופשר אוטומטית. אירוני בהתחשב בכך שכבר שנים לא מעט חוקרים (כולל כותב שורות אלו) [מסתבכים להפעיל אותו במכונה וירטואלית](#). אז כיצד ניתן להפעיל אותו?

הגדרת Credential Guard במחשב בודד דרך ה-Registry

כאמור מנגנון האבטחה הנ"ל עושה שימוש ב-VBS ולכן ראשית נדרש לוודא כי קודם Device Guard מאפשר שימוש בו ורק אז להגדיר את Credential Guard.

1. ניכנס לנתיב שאחראי על Device Guard בריגסטרי:

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\DeviceGuard`

ונוסיף שני מפתחות:

- הראשון `"EnableVirtualizationBasedSecurity"=DWORD: 1` בשביל לאפשר VBS.
- השני `RequirePlatformSecurityFeatures` גם מסוג DWORD. במידה ואנחנו רוצים רק שימוש ב-Secure Boot ניתן לו את הערך "1". במידה ואנחנו מכוונים גם ל-Secure Boot וגם אל DMA Protection ניתן לו את הערך "3".

2. ניכנס לנתיב שאחראי על LSA בריגסטרי:

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa`

ונוסיף ערך DWORD בשם `LsaCfgFlags` אשר יכול לקבל שלושה ערכים:

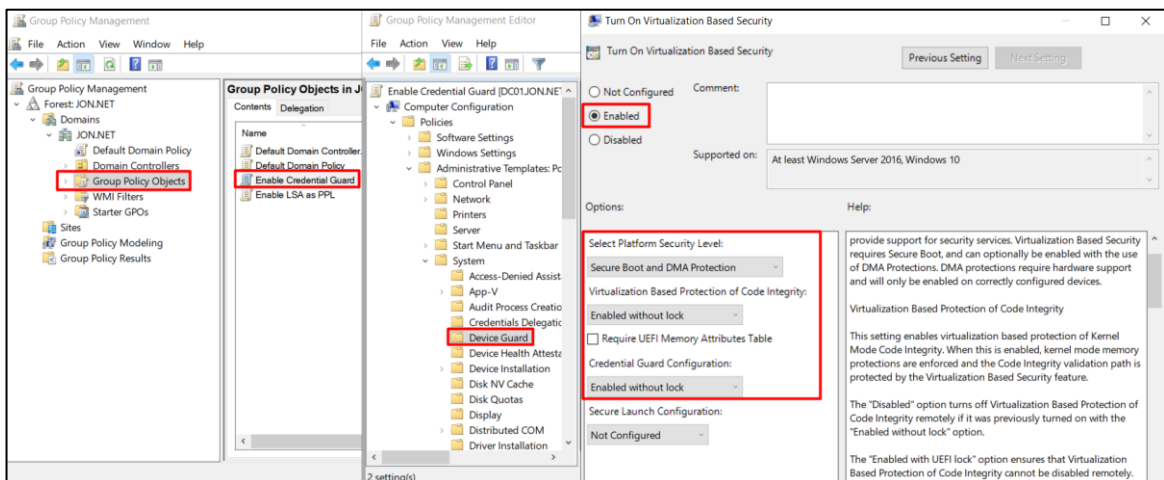
- "0" - בשביל לבטל את Credential Guard
- "1" - בשביל לאפשר את Credential Guard עם UEFI lock
- "2" - בשביל לאפשר את Credential Guard ללא UEFI lock

אישית אני ממליץ לאפשר VBS ו-Secure Boot אבל לא UEFI lock למקרה ויהיה צורך לשנות את ההגדרה ממחשב מרוחק (ללא גישה ל-firmware). מצד שני, אני גם ממליץ לא להקשיב למאמרים רנדומאליים באינטרנט ותמיד לעשות את המחקר לכדאיות של כל שינוי בעצמכם. אלטרנטיבית, אפשר פשוט להגדיר את הכל בריגסטרי דרך CLI של PS בצורה הבאה:

```
reg add "HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard" /v "EnableVirtualizationBasedSecurity" /t REG_DWORD /d 1 /f
reg add "HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard" /v "RequirePlatformSecurityFeatures" /t REG_DWORD /d 3 /f
reg add "HKLM\SYSTEM\CurrentControlSet\Control\DeviceGuard" /v "Locked" /t REG_DWORD /d 0 /f
```

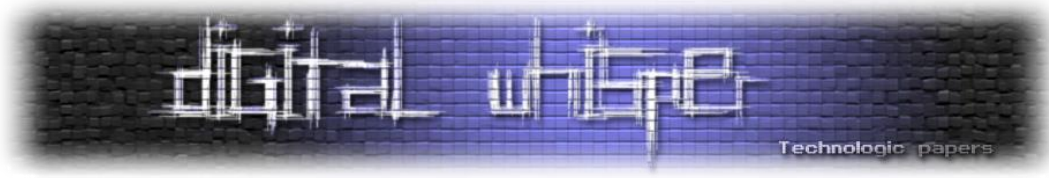
הגדרת Credential Guard בדומיין באמצעות GPO

כעת נגדיר באמצעות חוקת GPO את השינוי ערכים בריגסטרי של כלל המחשבים בדומיין. ב-DC שאחראי על ניהול האובייקטים בדומיין יש לפתוח את קונסולת Group Policy Management וליצור חוקת GPO חדשה שמקושרת ברמת הדומיין (או אל OU) שמכילה את כלל המחשבים. כמו שכבר אמרתי מקודם בהסבר על PPL, תמונה שווה אלף מילים אז נסתפק בתמונה:



לאחר מכן כל שנשאר הוא לעדכן את חוקות ה-GPO על ידי הרצת הפקודה:

```
gpupdate /force
```



על מנת לבדוק ש-Credential Guard אכן מוגדר במחשב ניתן להריץ את פקודת PowerShell הבאה:

```
(Get-CimInstance -ClassName Win32_DeviceGuard -Namespace root\Microsoft\Windows\DeviceGuard).SecurityServicesRunning
```

כאשר הפלט שמתקבל הוא "1" - Credential Guard מאופשר ובמצב ריצה. ובמידה הפלט הוא "0" - אז Credential Guard לא עובד.

הפרטים הקטנים - איך הכל מנגן ביחד

דיברנו על מלא נושאים שונים החל מכיצד נראה תהליך אסיפת ה-credentials ב-Windows, אילו פרוססים משתתפים בתהליך ומהם מנגנוני ההגנה שאמורים להגן עליהם. אבל לא הסברנו כיצד זה קורה - כיצד PPLs ושימוש ב-Credential Guard באמת מצליחים (והאם הם באמת מצליחים?) להגן על LSASS. ובכן, זה הזמן לחבר את כל הנושאים האלו ביחד.

אחרי תהליך logon מוצלח לעמדה, תהליך lsass.exe יחזיק בזיכרון לעיתים את **סימת המשתמש** בתצורת clear text, עם זאת במידה ומדובר באימות מבוסס NTLM הוא יחזיק בתוכו את ה-hash של סימת המשתמש (נקרא גם **NTOWF** - NT one-way function) אשר מבוסס על MD4. בצורה דומה כאשר תהליך Logon מבוסס על Kerberos, תהליך lsass.exe יכיל בכתובות הזיכרון שלו את מבני הנתונים של TGT (Ticket Granting Ticket). כאשר הודעת KRB_AS_REP חוזרת היא מכילה את ה-TGT חתום על ידי סימת היוזר krbtgt ביחד עם מפתח session שחתום על ידי ה-hash של סימת המשתמש שהגיש את הבקשה. זה מדוע במסגרת מתקפת AS-REP roasting ניתן להשיג את ה-hash של היוזר (ולשבור אותו כנגד מילון ב-offline). לאחר מכן, המשתמש יכול להגיש את ה-TGT שוב ל-KDC בשביל לקבל TGS מוצפן באמצעות ה-service hash שאליו הוא רוצה להזדהות. כך, השרת יוכל לקלף את ההצפנה ולאמת את ה-TGS. כאן נכנסת מתקפת kerberoast שפועלת על אותו מנגנון על מנת לשבור את ה-hash של ה-service account. ממליץ על המאמר [2nd Step to Tame a Kerberos: Hit It Where It Hurts](#) לאלו המעוניינים להבין טוב יותר את הנושא.

בשביל להגן על סימת המשתמש, על ה-TGT ועל ה-NTOWF, נעשה בפרוסס של LSASS שימוש ב-Credential Guard. כיצד הוא עוזר להגן על כל סוג credential?

הגנה על סימה - סימת המשתמש מוצפנת באמצעות מפתח סימטרי (AES) בשביל לאפשר SSO באמצעות פרוטוקולים כמו RDP. מכיוון ששירותים אלו עושים שימוש בסימת clear text היא חייבת להישמר בזיכרון התהליך - מה שמאפשר לתוקפים לבצע code injection, שימוש בכלי debugger, שיטות אקספלוטציה נוספות ופיענוח. Credential Guard לא יכול לשנות את אופן היישום של אותם פרוטוקולים לא בטוחים. לכן, "הפתרון" היחיד שהוא מספק במקרה של סימאות clear text הוא **לבטל את הפונקציונאליות של SSO** (כמובן שזה לא כיף ומכריח ביצוע re-authenticate). זו גם הסיבה מדוע



מייקרוסופט ממליצים לבטל לחלוטין את השימוש בסיסמאות ולעבור לעשות שימוש במזהים ביומטרים כמו Windows Hello שעובד עם תווי פנים\ טביעת אצבע. בצורה כזו, כאשר מתבצע אימות ניתן להוריד את משטח התקיפה משמעותית מפני מתקפות keyloggers, kernel sniffing/hooking tools ואפליקציות user-mode שמבצעות spoofing. כמו כן, שימוש ב-MFA חזק כמו YubiKey או Smartcard בצמוד ל-PIN יכול לעשות את העבודה.

לפי [STIG](#), על מנת לבטל את השימוש ב-Wdigest כשיטת אימות ניתן לשים ב-registry תחת:

```
KEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\WDigest
```

את הערך `UseLogonCredential` עם DWORD שווה 0.

על מנת להגן על NTOWF או TGT שמאוחסנים ב-LSASS נדרש להגן על LSASS בשלמותו - כאן נכנסים לתמונה תהליכי PPL עליהם דיברנו מקודם. אם התהליך חתום ומוגן, תוקף פוטנציאלי לא יוכל לגשת אליו. עם זאת, חשוב להבין כי אופציה זו מספקת הגנה מפני תהליכים זדוניים במרחב ה-user-mode; היא ממש לא מספקת הגנה מפני מתקפות kernel או מתקפות user-mode שמנצלים חולשה ב-kernel drivers (שכאמור קיימים לא מעט שניתן לטעון, ארחיב על הנושא במאמר אחר).

ובכן, Credential Guard מבטל את משטח התקיפה הנ"ל בכך שהוא טוען תהליך נוסף - `Isaiso.exe` אשר רץ בתור Trustlet בתוך VTL1. לכן, LSASS למעשה לא מחזיק את הסודות של המשתמש - אלא `Isaiso` מחזיק ואליו התוקף לא יכול בכלל לגשת כי הוא נמצא ב-"מימד קוד" אחר.

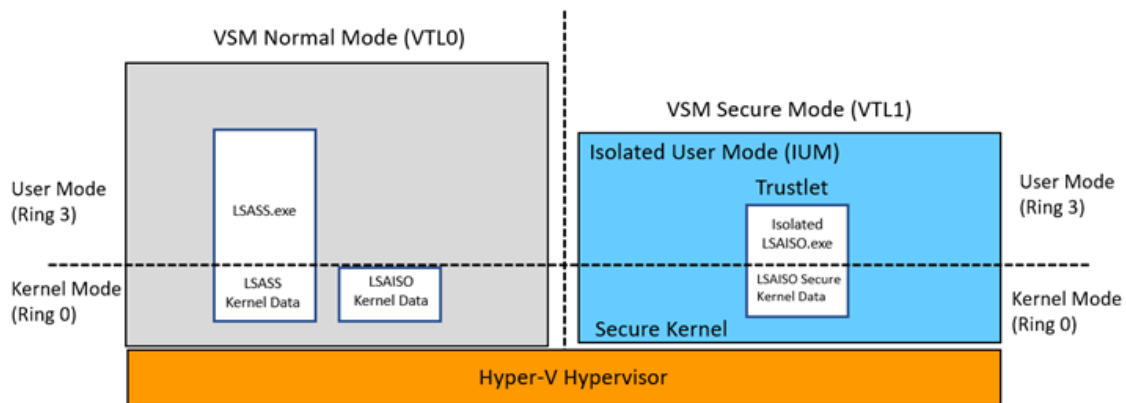
ל-VTL1 יש משטח תקיפה מינימלי (קטן משמעותית לעומת VTLO שמוכרת לנו), מכיוון שאין לה את ליבת ה-"NT" הרגילה המלאה, וגם אין לה מנהלי התקנים או גישה ל-I/O של חומרה מכל סוג שהוא. ככזו, LSA מבודד (שהוא VTL1 Trustlet), אינו יכול לתקשר ישירות עם KDC.

זוהי עדיין אחריותו של `lsass.exe`, המשמש כיישום ופרוטוקול proxy על מנת לתקשר עם ה-KDC כדי לאמת את המשתמש ולקבל את ה-TGT (ואת המפתח) וה-NTOWF. LSASS גם אחראי על התקשורת עם שרת אליו אנחנו מזדהים באמצעות ה-ticket הרלוונטי (TGS).

אם עקבתם אחרי הפסקה האחרונה, כנראה שתפסתם את הבעיה העקרונית בארכיטקטורה של התהליך הזה - ה-TGT והמפתח/NTOWF שלו עוברים באופן זמני דרך LSASS במהלך האימות, וה-TGT והמפתח שלו זמינים איכשהו ל-LSASS להפקת ה-service ticket.

זה מוביל לשתי שאלות:

- איך LSASS שולח ומקבל את הסודות מ-LSA המבודד?
- כיצד נוכל למנוע מתוקף להתחקות בצורה לגיטימית? הרי חייבת להיות תקשורת בין lsaiso.exe אל lsass.exe. מה מונע מתוקף לבצע מתקפות MITM?



[מקור - Microsoft]

לגבי השאלה הראשונה, התשובה היא **Advanced Local Procedure Call (ALPC)**. מדובר במכניזם של תקשורת העושה שימוש בסט קריאות מערכת שתומכות בביצוע RPC לוקאליות על ידי [ncalrpc](#) מרחוק. אופן העבודה של הפרוטוקול הוא על ידי פתיחת פורטים מבוססי ALPC שמאפשרים תקשורת בצורה דינאמית בין תהליכים ברמות שונות של אמון.

על ידי שימוש ב-ALPC ה-secure kernel מסוגלת לעביר את קריאות המערכת אל הקרנל הרגילה. לאחר מכן, ה-user-mode של VTL1 מיישם תמיכה ב-RPC על גבי פרוטוקול ALPC אשר מאפשר לאפליקציות VTL1 ואפליקציות VTL0 לתקשר ביניהן בדיוק כמו שהיו עושות עם RPC לוקאלי.

למעשה אם נפתח את ה-Process Explorer שוב נוכל לראות את תהליך lsaiso.exe עם ה-handle ופורט ALPC שלו (אלו למעשה מתקשרים עם lsass.exe):

Lsalso.exe	1,600 K	3,472 K	936
lsass.exe	10,632 K	23,976 K	952 Local Security Authority Process
fontdrvhost.exe	1,356 K	3,528 K	1084 Usermode Font Driver Host
csrss.exe	4,140 K	7,396 K	860 Client Server Runtime Process
winlogon.exe	2,528 K	10,888 K	1504 Winlogon

Type	Name
ALPC Port	\RPC Control\LSA_ISO_RPC_SERVER
Directory	\BaseNamedObjects
File	C:\Windows
Thread	<Access is denied.>

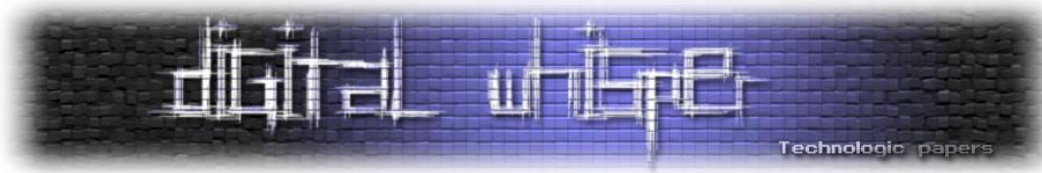
בשביל לענות על השאלה השנייה (MITM) נדרש לערב קצת קריפטוגרפיה ומודל של Challenge-Response בכל הסיפור. מניעת מתקפות MITM בין ה-KDC ופרוטוקול ה-LSA המבודד דומה במיוחד לאופן פעולת SSL/TLS ואופן פעולת האינטרנט (כבר ציינתי כמה פעמים שרשמתי [מאמר](#) קצרצר על כך).

למרות ש-LSASS מתנהג כמו proxy, הוא למעשה רק רואה ומעביר **תעבורה מוצפנת** בין ה-KDC ותהליך ה-LSA המבודד. שני הצדדים מגבשים תקשורת מבוססת על session key (מפתח סימטרי) אשר נמצא רק ב-VTL1 וב-KDC. ה-KDC מצפין את session key עם מפתח פרטי שרק לו יש, בצורה הזו ה-KDC יכול לענות עם TGT והמפתח שלו אחרי שהוא הצפין את התעבורה עם ה-session key של ה-LSA.

המודל הנוכחי הנ"ל מסוגל להגן אפילו על אימות מבוסס NTLM, כיצד? כאשר המשתמש מבצע logon עם plain text, LSA שולח אותה אל תהליך ה-LSA המבודד, אשר בתורו מצפין את הסיסמה עם ה-session key שלו ומחזיר את ה-credentials אל lsass.exe. לאחר מכן, כשנדרש לבצע את ה-Challenge-Response תהליך ה-LSASS שולח את ה-NTLM challenge ביחד עם ה-credential המוצפן אל ה-LSA המבודד. הוא בתורו מפענח את התכולה, מוודא שהיא נכונה ושולח את ה-NTLM response על בסיס ה-challenge בתמורה.

המודל הנוכחי לא מושלם. למעשה, אפשר לחלק את פגיעותיו ל-4 קטגוריות של מתקפות:

- במידה והמכונה כבר נמצאת תחת שליטת התוקף, ניתן ליירט את הסיסמאות plain text בזמן שהן מוקלדות (Keylogger) ו\או בזמן שהן נשלחות לתהליך ה-LSA המבודד (במידה ו-LSASS נתון לשליטת התוקף).
- מכיוון שאין ל-NTLM מנגנון anti-replay ניתן לתפוס את ה-NTLM response ולענות באמצעותו שוב על אותו ה-challenge. לחילופין, במידה ותוקף הצליח להשתלט על LSASS לאחר ביצוע ה-logon הוא יוכל לתפוס את ה-credentials המוצפנים ולשלוח אותם ל-LSA המבודד בשביל ליצור NTLM response חדש עבור NTLM challenges אקראיים. עם זאת, שיטת התקיפה הזו רלוונטית רק עד ביצוע reboot מכיוון שבזמן עליית המערכת ה-LSA המבודד יוצר session key חדש.
- במקרה של התחברות Kerberos-ית, ניתן ליירט ה-NTOWF (שכאמור לא מוצפן) ואז לעשות בו שימוש מחדש - בדיוק כמו במתקפה סטנדרטית של pass-the-hash. כמובן שגם פה נדרשת השתלטות מבעוד מועד על המכונה.
- Downgrade - תוקף עם גישה פיזית למכונה יכול לבטל את Credential Guard (חשוב לזכור שבסה"כ מדובר בהגדרה ב-registry במידה ולא מוגדר UEFI lock) ואז לעשות שימוש במודל אימות legacy. בקו מחשבה דומה, ניתן לבטל את הגדרת ה-Protected Process ולבצע Reboot למכונה. כמובן ששימוש ב-UEFI Secure Boot ו-UEFI ימנע את שינוי ה-Registry.



סיכום

נקודה שקצת הוזנחה במאמר היא ההיסטוריה של כל הסיפור. מאיפה הגיע הצורך להוסיף מגננות על מגננות, לבודד תהליכי אימות ולהכיל חתימה על כל דבר שזז? התשובה היא שפשוט עבר יותר מידי זמן. סט האימונים שפרוטוקולים כמו WDigest ו-NTLM גדלו לתוכם פשוט לא תואם אל ה-design שלהם (ובטח ובטח של AD בכלליותו). אם בעבר להעביר hash של הסיסמה עצמה נחשב לבטוח, הרי שהיום כרטיס מסך Nvidia GeForce RTX 4090 יכול לפצח [300 מיליארד](#) (!) גיבובי NTLM בשנייה. פרוטוקולים שנבנו בשנות ה-90 פשוט לא תוכננו לכזו רמה של עמידות.

מסיבה זו בדיוק הגיעו מנגנוני האבטחה PPL ו-Credential Guard לחיזוקו של LSASS וכלל מערכת ההפעלה. ותאמת, הם שמים את Microsoft במצב די טוב. Mimikatz כבר לא רלוונטי (לפחות לא לבדו) וללא ספק קשה היום משמעותית לחלץ מידע מ-LSASS. אין זה אומר שאלו יספיקו בשביל למנוע מתוקף להתפשט ברשת אבל הוא יצטרך להשתמש באמצעים מתקדמים (ומסובכים) בשביל לעקוף אותם, מה שבסופו של דבר **יגדיל את הסיכוי שלו להתגלות**.

ישנם כלים התקפיים שמשכו תשומת לב ברמת החידוש והיצירתיות שלהם. בין אם זה ניצול מנגנון פנימי של התהליך או סביבתו ובין אם זה גילוי\המצאת וקטור תקיפה חדש ברמת התשתית. משפט זה מוביל אותנו בצורה טובה לסיום המאמר וכהכנה להמשך.

במאמר הבא אני מתכוון להיכנס לחלק מהכלים הבולטים לתקיפת LSASS, להריץ אותם בסביבת דומיין ולהסביר ברמה טכנית כיצד הם פועלים. רשימת הכלים הנוכחית (אשר צפויה להתעדכן) הינה: [Mimikatz](#) (כמובן), [PPLdump](#), [NanoDump](#), [RToolZ](#), [PPLcontrol](#), [EDRSandblast](#), [pypykatz](#).

אשמח לנצל את הבמה ולומר שבמידה ואתם משתמשים/מכירים כלי **קוד-פתוח** שמצליח לחלץ נתונים חרף כל מנגנוני ההגנה של Microsoft ולא נמצא פה ברשימה - תשלחו לי כדי שאעבור על הקוד שלו ואולי נציג אותו במאמר הבא ☺

על הכותב

[יהונתן אלקבס](#), חוקר אבטחת מידע בחברה לא קטנה ולא פרטית. חובב סוקולנטים, קפה וטיולים.