Digital Whisper Technologic Papers

Threadmap - Detecting Process Hollowing

מאת קייל נס, שחף עטון וליעם שטיין

הקדמה

Process Hollowing הינה טכניקה ותיקה (Stuxnet השתמשה בה עוד בשנת 2010) המאפשרת לכלי להריץ קוד תחת מעטפת של תהליך אחר על מנת לשמור על חשאיות. למרות שמדובר בטכניקה ישנה, עדיין ניתן לראות שימוש רחב בטכניקה הזו ע"י כלים/פוגענים שונים. ניתן גם לראות שקיימת עלייה ברמת התחכום של הכלים מבחינת יכולות חשאיות בזמן ריצה, יכולת "להתקפל" בצורה יסודית ללא הותרת סימנים מעידים, יכולות Anti-Reverse, Anti-Forensics וכו'. לצד ההתקדמות של הכלים עולה גם רמת התחכום והיכולות של אנשי אבטחה לחקור, לסווג ולמצות "אירוע אבטחתי".

במאמר זה נתייחס לנושא של מציאת עדות להזרקת קוד מסוג Process Hollowing בעת חקירת זיכרון מנקודת המבט שלנו.

הצגת מטרה

בין אם טכניקות ה-Process Hollowing משתנות יחד עם הקדמה, או נשארות פחות או יותר אותו הדבר, ראינו כי רוב יכולות הזיהוי בעת חקירת זיכרון מתייחסות למקרה "הפשוט" של Process Hollowing, כלומר כנראה שנוכל לתפוס אימפלמנטציה של PoC בכלי, אך אם אנחנו מתעסקים עם כלי מתוחכם יותר בו נעשתה חשיבה עמוקה על מנת להסתיר שיטות הזרקה (מתוך הבנה של איך כלי חקירה תופסים אותן) נמצא את עצמנו במערכה קשה.

מטרת המאמר היא להציג דרך נוספת לאתר הזרקת קוד מסוג Process Hollowing בעת חקירת זיכרון. לשם כך נציג במהלך המאמר את התהליך שעברנו עד שפתיחנו את השיטה. ראשית, עלינו להכיר מהו Process Hollowing ואיך הוא משומש כיום - טכניקה זו מתוארת במספר מקורות באינטרנט, ספרים (*The Art of Memory Forensics*) ואפילו בגיליון ה-77 של Digital Whisper. לאחר מכן, נכיר את טכניקות חקירת הזיכרון שמכוונות למצוא Process Hollowing ונבחן כיצד אנחנו יכולים לממש Process Hollowing שיוכל להתחמק משיטות אלה. לבסוף, נציג את השיטה שלנו.

הסבר - Process Hollowing

טכניקת ה-Process Hollowing נוצרה על מנת לענות על צורך בקרב תוקפים שונים - הסתרה של פעולות לא לגיטימיות. מדובר בטכניקה נפוצה המשתייכת למשפחת הזרקות קוד אשר במהלכה תוקף יוצר תהליך לגיטימי בהשהייה, "מרוקן" אותו, מזריק את הקוד של התהליך הזדוני, ולבסוף מפעיל את התהליך (מבטל את ההשהייה). בדרך זו, התוקף יוצר תהליך זדוני אשר רץ תחת מעטפת לגיטימית. המשתמש הפשוט יראה תהליך שנראה לגיטימי, מוכר, ואף עם חתימה דיגיטלית (לדוגמה: Microsoft), כאשר בפועל מדובר בתהליך זדוני במסווה.

על מנת לדבר על Process Hollowing במובנים מקצועיים יש לדעת תחילה את המושגים הבסיסיים (לדוגמה: PEB[1], IBA[2]) ואת התהליך של טעינת תהליך לזיכרון אשר הוסברו בקפידה בספר *Windows Internals 6th edition part 1*. כעת, נעבור בקצרה על השלבים של Process Hollowing כפי שהוצגו בספר *The Art of Memory Forensics*:

1. התחלת תהליך לגיטימי (לדוגמה: C:\windows\system32\lsass.exe), אבל עם ה-Main thread בהשהייה. בנקודה זו, ה-ImagePathName כחלק ממבנה ה-PEB מציג את הנתיב המלא של lsass.exe לגיטימי.
2. השגה של הקוד הזדוני אשר עתיד להיות מוחלף. קוד זה יכול להיות מושג דרך הדיסק, מהזיכרון או דרך הרשת.
3. השגת ה-ImageBaseAddress של lsass.exe, ושחרור\הסרת מיפוי של איזור הזיכרון הרלוונטי. בנקודה זו, התהליך ריק מתוכן, כלומר - אינו מכיל DLL-ים[3], Heaps, Stacks. בנוסף ה-Handleים עדיין נשארים פתוחים, אך ללא קובץ הרצה קיים.
4. מיפוי סגמנט חדש של זיכרון בתוך מרחב הזכרון של lsass.exe, תוך כדי וידוי שהזיכרון שהוקצה הוא בהרשאות קריאה, כתיבה והרצה (READWRITE_EXECUTE). ניתן להשתמש באותה הכתובת של ה-ImageBaseAddress או בכתובת חדשה.
5. העתקה של ה-Header[4] PE של התהליך הזדוני לתוך הזיכרון הממופה החדש בתוך lsass.exe.
6. העתקה של כל אחד מה-Sections של הקובץ הזדוני אל תוך זכרון וירטואלי כהלכה בתוך lsass.exe.
7. השמה של כתובת ההתחלה של ה-Main Thread (זה שהותחל במצב השהייה) לנקודת הכניסה של התהליך הזדוני (למקום בו נמצא הקוד אותו הזרקנו, ואנחנו רוצים שירוץ).
8. התחלה של ה-Thread. בנקודה זו, התהליך הזדוני מתחיל לרוץ תחת מעטפת ה-lsass.exe. ה-ImagePathName בתוך מבנה ה-PEB עדיין מצביע ל-lsass.exe.

---

[1] PEB – Process Environment Block
[2] IBA – Image Base Address
[3] DLL – Dynamic Link Library
[4] PE – Portable Executable

שיטות זיהוי נפוצות

כאשר מדברים על שיטות זיהוי למתקפות שונות, נוכל למצוא דרכים רבות דרך כלים שונים. נוכל לזהות את טכניקת ה-Process Hollowing דרך זיהוי של פונקציות API שונות במוניטור מסויים, ב-strings של אותו קובץ, ועוד. במאמר נתרכז בדרכים של מציאת הטכניקה בזכרון דרך מבנים שונים ב-Windows.

בקהילת Volatility נוכל למצוא מאגר של כלי זיהוי להזרקות קוד ככלל, ול-Process Hollowing בפרט. הדרכים העקריות למציאת Process Hollowing הן:

1. ההרשאות של ה-VAD[5]. גישה קלאסית היא זיהוי של זכרון מוקצה תחת הרשאות של כתיבה,קריאה והרצה (PAGE_EXECUTE_READWRITE). זו גישה מאוד גסה עקב כל התראות השווא שהיא מספקת.
2. השוואה בין ה-ImageBaseAddress שנמצאת ב- PEB לבין ה-VAD-ים של התהליך.
   א. ה-ImageBaseAddress חייב להצביע אל VAD עם File Object.
   ב. השוואה בין שם הקובץ הנמצא ב- PEB לבין שם הקובץ הכתוב ב-VAD שאליו ה-IBA מצביע.
3. השוואה בין ה-LDR Lists[6] לבין המידע הנמצא ב-VAD.

עקיפת שיטות הזיהוי הנפוצות

לאחר בחינה של פלאגינים קיימים, הצלחנו לבצע Process Hollowing מבלי להיחשף. במהלך ה-PoC[7] הבא, נציג מעקף של כל אחת מהטכניקות אשר הוזכרו בחלק הקודם. יש לציין כי אף אחת מהטכניקות בהן השתמשנו לא דרשו הרשאות Kernel.

שינוי ההרשאות ב-VAD

ההרשאות הנמצאות ב-VAD הן ההרשאות איתן ה-VAD נוצר. כלומר, אם לאחר היווצרות ה-VAD הרשאותיו ישתנו (לדוגמה בעזרת פונקציית VirtualProtect) ההרשאות לא יתעדכנו ב-VAD (כך גם מתואר בספר *The Art of Memory Forensics*).

למעשה, נוכל למפות זיכרון בהרשאות של קריאה וכתיבה, להוסיף את ההרצה רק לאחר מכן. במצב זה החוקר ימצא את עצמו במצב שהוא מסתכל על VAD עם הרשאות של קריאה, וכתיבה כאשר בפועל, קיימות גם הרשאות הרצה.

---

[5] VAD - Virtual Address Descriptor

[6] LDR - מצביע למבנה נתונים הנמצא ב-PEB בשם _PEB_LDR_DATA שמכיל מידע לגבי ה-DLL-ים הטעונים בתהליך.

[7] PoC - Proof Of Concept – על מנת להדגים את השימוש ב-Process Hollowing נעזרנו בפרויקט ב-GitHub שנמצא ב:
https://github.com/m0n0ph1/Process-Hollowing
אנחנו רוצים לציין כי הפרויקט שייך ל-m0n0ph1 על זכויותיו. במהלך ה-POC הוספנו קטעי קוד שלנו על מנת להדגים כיצד אנחנו עוקפים את שיטות הזיהוי של היום. השימוש נעשה למטרות לימודיות ואנחנו לא לוקחים אחריות על כל שימוש זדוני בפרויקט ובהדגמות.

```
Process: svchost.exe Pid: 2640 Address: 0x430000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 14, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x00430000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x00430010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x00430020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00430030  00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00   ................
```

[תמונה 1-1, חיתוך הפלט של malfind ב-Volatility על ה-KSLSample.vmem[8]]

אם נסתכל על מיצג 1-1 נוכל למצוא אנומליות שונות ב-VAD:

- קיימת עדות לכתיבה של קובץ הרצה (PE) ע"י הימצאות ה-magic number - MZ
- ההרשאות על ה-VAD מסומנות כ-PAGE_EXECUTE_READWRITE
- חוסר תאימות בין קובץ הרצה טעון, לבין היעדר File Object

התהליך Svchost.exe עם ה-PID: 2640, היה נתון ל-Process Hollowing. כאשר הבנו את הבדיקות ש-malfind מבצע (בדיקה של ההרשאות של ה-VAD), חילקנו את ההקצאה של הזיכרון ליותר שלבים:

- הקצאה של זיכרון בתוך מרחב הכתובות של התהליך ללא הרשאות הרצה, (למשל PAGE_READWRITE ב-PoC שביצענו).
- כתיבה של הקובץ בזכרון המוקצה
- הוספה של אפשרות הרצה במרחב הזכרון המוקצה.
- הרצה של הקוד המוזרק

```
    PVOID pRemoteImage = VirtualAllocEx(
        pProcessInfo->hProcess,
        pPEB->ImageBaseAddress,
        pSourceHeaders->OptionalHeader.SizeOfImage,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE
    );
    DWORD old = 0 ;
    VirtualProtectEx(
        pProcessInfo->hProcess,
        pRemoteImage,
        pSourceHeaders->OptionalHeader.SizeOfImage,
        PAGE_EXECUTE_READWRITE,
        &old
    );
```

הקוד המוצג לעיל הוא הצצה לתוספות שלנו לקוד המקור של ProcessHollowing.exe. בעזרת תוספת זו, הצלחנו להתחמק מ-malfind.

[8] Memory Dump - KSLSample.vmem של Windows 7 64bit שהכנו לטובת המאמר. ה-Memory Dump מכיל בין השאר 5 תהליכים של svchost.exe עליהם ביצענו Process Hollowing בשיטות שונות להדגמות.

כאמור, לאחר השינוי בקוד, malfind לא מוצא שום פעילות חשודה לתהליך שבו בוצעה טכניקת ה-Process Hollowing (התהליך עם PID: 2784 בקובץ הזיכרון המצורף). אמנם הצלחנו להתחמק מבדיקה אחת, אך קיימים פלאגינים נוספים שיכולים לתפוס אותנו בינתיים.

השוואה בין ה-PEB ל-VAD

ה-IBA מצביע לקובץ ההרצה הטעון של התהליך, כלומר אם נפתח יישום של מחשבון (calc.exe) הקובץ עצמו יטען לזיכרון (במסגרת תהליך ההרצה במערכת ההפעלה של Windows). ה-IBA יצביע לאיזור בזיכרון בו טעון calc.exe. מבחינת מערכת ההפעלה, הקצאת קטע הזיכרון הזו תגובה ב-VAD עם File Object שכן הקובץ טעון שם. בנוסף ה-VAD Type יהיה Image File.

עד כאן תיארנו בקצרה "מה אמור להיות". ברגע שמתבצע Process Hollowing החלק המיועד להרצה (כאמור קובץ ההרצה הטעון) מוסר מהזיכרון (לפי שלב 3 בתיאור של Process Hollowing) ומתבצע מיפוי חדש לקוד הזדוני ושינוי ה-IBA לאיזור החדש אליו מופה בזיכרון (שלב 4 בתיאור של Process Hollowing). הקוד לא מגובה בקובץ וכתוצאה מכך נוצר VAD ללא File Object. הסתירה הזו יכולה לעזור לחוקרים לראות שיש כאן משהו לא תקין. אחת הבדיקות של hollowfind מבצע הוא למצוא בדיוק את חוסר ההתאמה של IBA המבציע ל-VAD שלא מגובה ב-File Object.

```
Hollowed Process Information:
        Process: svchost.exe PID: 2784
        Parent Process: ProcessHollowi PPID: 2088
        Creation Time: 2017-09-12 13:29:31 UTC+0000
        Process Base Name(PEB): svchost.exe
        Command Line(PEB): svchost
        Hollow Type: No VAD Entry For Process Executable

VAD and PEB Comparison:
        Base Address(VAD): 0x0
        Process Path(VAD): NA
        Vad Protection: NA
        Vad Tag: NA

        Base Address(PEB): 0x430000
        Process Path(PEB): C:\Windows\SysWOW64\svchost.exe
        Memory Protection: PAGE_READWRITE
        Memory Tag: VadS

0x00430000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x00430010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x00430020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00430030  00 00 00 00 00 00 00 00 00 00 00 00 f8 00 00 00   ................
```

[תמונה 1-2, חיתוך של הפלט hollowfind ב-volatility על KSLSample.vmem]

תמונה 1-2 מציגה את הפלט של hollowfind על תהליך PID: 2784 (התהליך שציינו בהדגמה הקודמת שהצליח להתחמק מ-malfind). Hollowfind הצליח לאתר את התהליך הזה כחשוד, מכיוון שה-IBA מצביע ל-VAD שלא מגובה ב-File Object. ניתן גם להבחין שההרשאות של ה-VAD הן PAGE_READWRITE למרות שהן שונו ל-PAGE_EXECUTE_READWRITE. במקום שה-IBA יצביע על קטע הקוד הזדוני שכתבנו נרצה לשנות אותו לאיזור אחר בזיכרון של התהליך עם File Object. אנחנו יכולים לשנות את ה-IBA מבלי לפגוע בריצה של התהליך, מכיוון שברגע שה-Main Thread חוזר לרוץ אין שימוש

בו יותר. הסיבה השניה שאנחנו יכולים לשנות את ה-IBA בכזו קלות היא שה-IBA (כחלק מה-PEB) נמצא בזיכרון User-mode, כך שאם יש לנו הרשאות לבצע שינויים בתהליך אנחנו יכולים לעשות כאוות נפשנו (במקרה הזה, אנחנו "התהליך" אז אנחנו יכולים לבצע שינויים על איזור הזיכרון השייך לנו).

```
    LONG_PTR ldr_addr;
    PPEB_LDR_DATA ldr_data;
    PLDR_MODULE LdMod;
    DWORD *bad_address;


    __asm mov eax, fs:[0x30]  //get the PEB ADDR - 32 bit
        __asm add eax, 0xc
    __asm mov eax, [eax] // get LoaderData ADDR
        __asm mov ldr_addr, eax


    ldr_data = (PPEB_LDR_DATA)ldr_addr;
    LdMod = (PLDR_MODULE)ldr_data->InLoadOrderModuleList.Flink->Flink;
// get the second dll that is loaded
    bad_address = (DWORD*) LdMod->BaseAddress;

    __asm mov eax, gs:[0x60] // using the GS register to get to the peb
    __asm add eax, 0x10 // get the image base address
    __asm mov edx, bad_address
    __asm mov[eax], edx // change IBA
```

בקטע קוד המתואר הוצאנו מאחת הרשימות של ה-LDR (InLoadOrderModuleList - הרשימה המכילה את ה-DLL-ים בסדר הטעינה שלהם בזיכרון) את המצביע (Pointer) לאחד ה-DLL-ים. לאחר מכן שינינו את הערך של ה-IBA שיצביע לאותו DLL.

אחרי ששילבנו את השינוי הנ"ל בקוד הצלחנו להישאר חבויים מ-hollowfind, אך עדיין malfofind הצליח למצוא אותנו.

```
Process: svchost.exe Pid: 2220 Ppid: 2696
Address: 0x77490000 Protection: PAGE_EXECUTE_WRITECOPY
Initially mapped file object: c:\windows\syswow64\svchost.exe
Currently mapped file object: \windows\SysWOW64\ntdll.dll
0x77490000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x77490010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x77490020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x77490030  00 00 00 00 00 00 00 00 00 00 00 00 d8 00 00 00   ................
```

[תמונה 1-3, חיתוך הפלט של malfofind ב-Volatility על KSLSample.vmem]

לפי תמונה 1-3, התהליך PID: 2220 נמצא חשוד ע"י malfofind, מכיוון שקיימת חוסר תאימות בין הנתיב המלא של הקובץ שאמור להיות ממופה, לבין הקובץ שה-IBA מצביע אליו (אנחנו יכולים להסיק גם שה-DLL אליו שינינו את ההצבעה הוא ntdll.dll). malfofind משיג את הנתיב המלא מ-ProcessParameters (גם הוא חלק מה-PEB שנמצא ב-User-mode ונתון לעריכה) ומשווה אותו לנתיב המלא של הקובץ מה-File Object של ה-VAD אליו מצביע ה-IBA.

מיפוי קובץ הרצה (Image File)

אחת הדרכים שניתנות לביצוע על מנת להתחמק מההשוואה בין הנתיבים היא למפות את הקובץ המקורי (במקרה שלנו svchost.exe) מחדש במרחב הזיכרון של התהליך ולאחר מכן לשנות את ה-IBA אליו. כך אנחנו שואפים למינימום "חתימות מפלילות" והחשדה של התהליך.

```
hFile = CreateFile("C:\\Windows\\SysWOW64\\svchost.exe", GENERIC_READ,
FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
hMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
LPVOID dw;
dw = MapViewOfFile(hMap, FILE_MAP_READ, 0, 0, 0); //map the svchost file
VirtualLock(dw, sizeof(&dw)); // make sure that it won't be paged

LONG_PTR ldr_addr;
PPEB_LDR_DATA ldr_data;
PLDR_MODULE LdMod;
DWORD *bad_address = (DWORD *)dw;

__asm mov eax, gs:[0x60] // using the GS register to get to the peb
    __asm add eax, 0x10 // get the image base address
__asm mov edx, bad_address
__asm mov[eax], edx // change IBA
```

בקטע הקוד המתואר כאן, מיפינו קובץ בעזרת CreateFileMapping ו-MapViewOfFile על מנת ליצור VAD שיהיה מגובה ב-File Object. זה כמובן שונה מאשר פשוט לבצע הקצאה לזיכרון בגודל הקובץ, לקרוא בעצמנו את הקובץ ולכתוב אותו לאיזור הזיכרון שהקצנו (במקרה זה נראה VAD ללא File Object עם התוכן של הקובץ ונהיה חשופים לזיהוי פשוט).

ברגע ששילבנו את קטע הקוד הנ"ל הצלחנו להישאר חבויים מ-malfind, malfofind ו-hollowfind. בין הדברים המעניינים שכדאי לציין הוא הפלט הבא:

```
Pid      Process              Base               InLoad InInit InMem MappedPath
-------- -------------------- ------------------ ------ ------ ----- ----------
    2828 svchost.exe          0x0000000076380000 False  False  False \Windows\SysWOW64\user32.dll
    2828 svchost.exe          0x00000000000f0000 False  False  False \Windows\SysWOW64\svchost.exe
```

[תמונה 1-4, חיתוך של ldrmodules ב-Volatility ב-KSLSample.vmem על התהליך svchost.exe PID: 2828]

לפני שנסביר למה הפלט מעניין אותנו, נסכם בקצרה על ה-LDR. ה-LDR הינו חלק מה-PEB והוא מכיל שלוש רשימות מקושרות:

- InLoadOrderModuleList - רשימה מסודרת לפי ה-DLL-ים שצריכים להיטען לתהליך.
- InMemoryOrderModuleList - רשימה מסודרת לפי הימצאות (מבחינת כתובות) ה-DLL-ים במרחב הזיכרון של התהליך.
- InInitializationOrderModuleList - רשימה מסודרת לפי הרצת ה-DLLMain של ה-DLL-ים הטעונים בתהליך. לא בכל DLL שנטען מורצת פונקציית ה-Main שלו, כך שהרשימה הזו לא תמיד תציג את כלל ה-DLL-ים.

מדובר באותם DLL-ים, אך הם מוצגים כל פעם בסדר שונה. קובץ ההרצה של התהליך יוצג בתחילת InLoadOrderModuleList כי הוא נחשב לקובץ הראשון שנטען לזיכרון, לעומת זאת הוא לא יופיע ב-InInitializationOrderModuleList כי אין אצלו פונקציית DLLMain והוא מורץ בדרך שונה.

Ldrmodules מוציא את כל ה-Image Files הטעונים בזיכרון של התהליך (ע"י הוצאת ה-VAD-ים עם File Object עם MZ magic) והוא משווה אותם לרשימות. אם נחזור לפלט של תמונה 1-4 נוכל לראות ש-svchost.exe לא נמצא כביכול באף אחת מן הרשימות. הסיבה היא שברשימות של ה-LDR לא מעודכן המיקום החדש של svchost.exe שנטען. כדי לשנות את תוצאות הפלט של ldrmodules נוסיף את הקטע קוד הבא:

```
hFile = CreateFile("C:\\Windows\\SysWOW64\\svchost.exe",
GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,0,NULL);

hMap = CreateFileMapping(hFile,NULL,PAGE_READONLY,0,0,NULL);
LPVOID dw;
dw = MapViewOfFile(hMap,FILE_MAP_READ,0,0,0); //map the svchost file
VirtualLock(dw,sizeof(&dw)); // make sure that it won't be paged

LONG_PTR ldr_addr;
PPEB_LDR_DATA ldr_data;
PLDR_MODULE LdMod;
DWORD *bad_address = (DWORD *)dw;

__asm mov eax, gs:[0x60] // using the GS register to get to the peb
__asm add eax, 0x10 // get the image base address
__asm mov edx, bad_address
__asm mov[eax], edx // change IBA

__asm mov eax, gs:[0x60] // using the GS register to get to the peb
__asm add eax, 0x18 // get the LDR
__asm mov eax, [eax]
__asm add eax, 10h // get the InLoad list
__asm mov eax,[eax]
__asm add eax, 30h // go to where old svchost address was
__asm mov edx, bad_address
__asm mov[eax], edx // overwrite it with the bad address
```

לאחר מכן נריץ שוב את ldrmodules ונקבל את התוצאה הבאה:

```
Pid      Process              Base               InLoad InInit InMem MappedPath
-------- -------------------- ------------------ ------ ------ ----- ----------
    2460 svchost.exe          0x0000000076380000 False  False  False \Windows\SysWOW64\user32.dll
    2460 svchost.exe          0x00000000000f0000 True   False  True  \Windows\SysWOW64\svchost.exe
    2460 svchost.exe          0x0000000076a30000 False  False  False \Windows\SysWOW64\sechost.dll
```

[תמונה 1-5, חיתוך של ldrmodules ב-Volatility ב-KSLSample.vmem על התהליך svchost.exe עם PID 2460]

בתמונה 1-5 ניתן לראות עכשיו שקיימת התאמה בין ה-VAD אליו ממופה ה-svchost.exe החדש לבין ההצבעה אליו ב-LDR Lists. בנוסף התהליך נשאר חבוי מ-malfind, malfofind ו-hollowfind. קיימים DLL-ים שנראים שהם לא ממופים בזיכרון, בנוסף קיימים DLL-ים אחרים שכן ממופים. הסיבה היא שה-payload שלנו משתמש ב-DLL-ים אחרים מ-svchost.exe וכי כשהסרנו את svchost.exe המקורי בפעם

הראשונה, הסרנו גם DLL-ים שהיו ממופים. בעיקרון אנחנו יכולים לשנות את ה-LDR Lists (כיוון שגם הוא נמצא ב-User-mode) כדי שיצא פלט יותר אמין מהשימוש של ldrmodules.

המחקר

ראינו איך אנחנו יכולים לקחת את הטכניקה של Process Hollowing ולחדד אותה מעט על מנת שהיא עדיין תהיה רלוונטית, כלומר תצליח להישאר חבויה משיטות הזיהוי. הסיבה העיקרית שיכולנו לעשות זאת היא שאותם פלאגינים מסתמכים על מידע שנמצא בזיכרון User-mode כמקור השוואה. הבעיה היא שיחסית קל לשנות את מבני הנתונים ב-User-mode כדי להתאים את עצמנו ולהישאר חבויים. האתגר שהתמודדנו איתו הוא אם קיימים מבנים ב-Kernel-mode שנוכל לבצע בעזרתם את ההשוואות ולמצוא Process Hollowing ואולי גם הזרקות נוספות. הרעיון של השוואה בין "מה אמור לרוץ לבין מה רשום באותו קטע זיכרון" עדיין בבסיסו רעיון טוב. אחרי חפירה קצרה ראינו שני ערכים מעניינים שנמצאים במבנה _ETHREAD (מ-*Windows Internals 6th Edition*):

- StartAddress - הינו מצביע (pointer) לפונקציה ntdll.dll!RtlUserThreadStart שהיא פונקציית מעטפת שערכה נקבע ע"י מערכת ההפעלה.
- Win32StartAddress - הינו מצביע (Pointer) לפונקציה שתרוץ ע"י ה-Thread. המצביע לפונקציה עובר בעת שימוש בפונקצית ה-CreateThread API. אם ניקח את תהליך המחשבון מההדגמות הקודמות, ה-Win32StartAddress ב-_ETHREAD של ה-Main Thread יצביע לפונקציית main של הקובץ הנמצאת ב-text section.).

השינוי של Win32StartAddress הוא בלתי נמנע (שלב 7 בתיאור של Process Hollowing) והוא מצביע לקטע קוד שאמור להיות מורץ. ברגע שיש לנו את הערך של Win32StartAddress נוכל למצוא את ה-VAD של אותו איזור זיכרון. נוכל לאחר מכן להריץ מספר השוואות על מנת למצוא סימנים מחשידים.

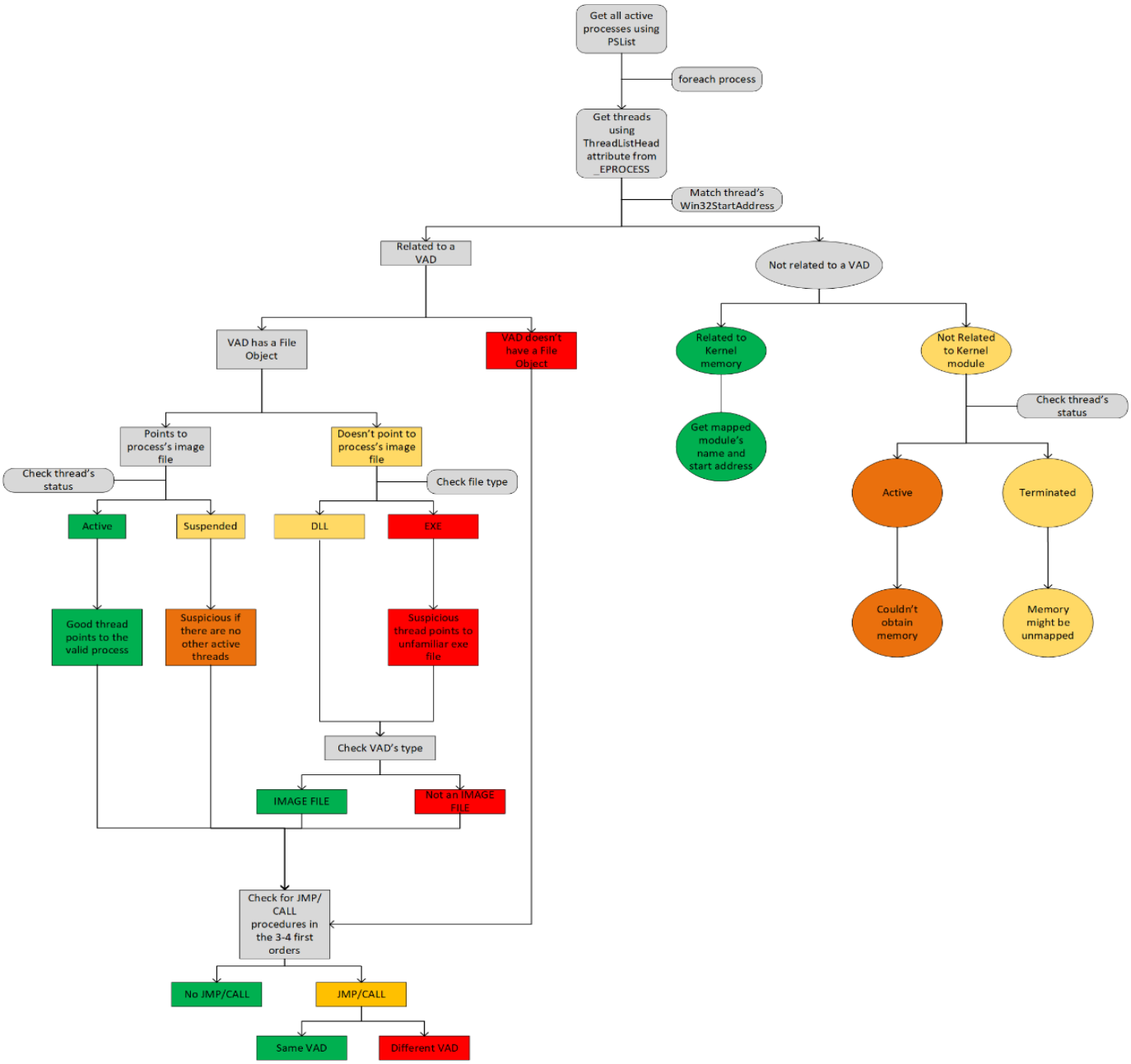
הנחות בסיס

אחרי שמצאנו שני מבנים Kernel-יים שנוכל לבצע בהם השוואות, ביצענו כמה הנחות שגיבשנו לידי חוקים על מנת להחשיד תהליך:

- כל VAD ש-thread מצביע אליו חייב להכיל File Object ולהיות מסוג Image File.
- לכל תהליך חייב להיות לפחות thread אחד שמצביע לאיזור בזיכרון עם קובץ (image file) טעון.
- Thread המצביע ל-VAD שה-File Object שלו הוא exe ששונה מהתהליך המקורי ייחשב כחשוד.
- קריאות JMP/CALL שנמצאו בקטע הזיכרון אליו מצביע ה-Win32StartAddress שמצביעות ל-VAD שונה ייחשבו כחשודות.

אחרי גיבוש החוקים כתבנו plugin שיישם את הבדיקות שביצענו. על מנת להבין את הגרף יש לשים לב לנקודות הבאות:

- ברגע ש-thread נמצא חשוד, כלומר הגיע לקטגוריה אדומה הוא ייחשב חשוד גם אם בדיקות ההמשך יוצאות שליליות (לדוגמה: Thread שה-Win32StartAddress מצביע ל-VAD ללא File Object ייחשב חשוד, גם אם לאחר מכן לא נמצאו שום JMP/CALL חשודים).
- קטגוריה כתומה לא תשתנה בחזרה לירוקה ונחשבת חשודה במידה סבירה.
- קטגוריה צהובה יכולה להיות מזוכה תוך כדי התהליך ולהשתנות לירוקה.
- קטגוריה ירוקה תחשב לתקינה.



ניתוח עם Threadmap

כפי שנאמר מקודם, המטרה של ה-plugin שלנו היא להשתמש בניתוח מבוסס Kernel-mode, במקום מבוסס User-mode על מנת למנוע שיבושים זדוניים.

הגישה בה נקטנו מבוססת על מיפוי של כל Thread ל-VAD התואם שלו, וניסיון להשיג כמה שיותר מידע לגבי התהליך אליו ה-Thread שייך. המיפוי מבוסס על כתובת ההתחלה הנמצאת בכל אובייקט _ETHREAD, המצביע לאזור בזיכרון בו ה-Thread מתחיל את ריצתו. על ידי מציאת ה-VAD, נהיה פשוט יותר להחליט אם תהליך כלשהו זדוני או לא.

המאפיינים הבאים נמצאים ב-VAD ויכולים להעיד על זדוניות:

1. ל-VAD אין File Object
2. JMP או CALL בתחילת הקוד הנמצא תחת ה-VAD נחשב חשוד במיוחד אם הקפיצה היא לאזור אחר בזיכרון.
3. ה-VAD לא מסומן כאובייקט השייך ל-Image טעון
4. ה-File Object מציין קובץ לא מוכר

עם שימוש בתכונות אלו אנחנו יכולים להבטיח תוצאות אמינות, ושיעור התראות שווא נמוך.

```
Thread Map Information:

Process: svchost.exe PID: 2460 PPID: 2596

** No thread is pointing to process's image file
** Found suspicious threads in process

Thread ID: 1228 (ACTIVE)

Reason:
        Thread points to a vad without a file object

Vad Info:
        Thread Entry Point: 0x432104
        Vad Base Address: 0x430000
        Vad End Address: 0x445fff
        Vad Size: 0x15fff
        Vad Tag: VadS
        Vad Protection: PAGE_READWRITE
        Vad Mapped File: ''

        0x00432104  c2 04 00 68 eb 20 43 00 e8 12 02 00 00 a3 20 20   ...h..C........
        0x00432114  44 00 59 83 f8 ff 75 03 32 c0 c3 68 e4 2a 44 00   D.Y...u.2..h.*D.
        0x00432124  50 e8 6d 02 00 00 59 59 85 c0 75 07 e8 05 00 00   P.m...YY..u.....
        0x00432134  00 eb e5 b0 01 c3 a1 20 20 44 00 83 f8 ff 74 0e   .........D....t.

        0x43210400 c20400           RET 0x4
        0x43210700 68eb204300       PUSH DWORD 0x4320eb
        0x43210c00 e812020000       CALL 0x432323
        0x43211100 a3202044005983f8ff MOV [0xfff8835900442020], EAX
        0x43211a00 7503             JNZ 0x43211f
        0x43211c00 32c0             XOR AL, AL
        0x43211e00 c3               RET
        0x43211f00 68e42a4400       PUSH DWORD 0x442ae4
        0x43212400 50               PUSH RAX
        0x43212500 e86d020000       CALL 0x432397
        0x43212a00 59               POP RCX
        0x43212b00 59               POP RCX
        0x43212c00 85c0             TEST EAX, EAX
        0x43212e00 7507             JNZ 0x432137
        0x43213000 e805000000       CALL 0x43213a
        0x43213500 ebe5             JMP 0x43211c
        0x43213700 b001             MOV AL, 0x1
        0x43213900 c3               RET
        0x43213a00 a12020440083f8ff74 MOV EAX, [0x74fff88300442020]
        0x43214300 0e               DB 0xe
-------------------------------------------------------------------------

-------------------------------------------------------------------------
```

Suspicious process name, PID and PPID

Reasons why the process is marked as suspicious

Thread's ID and state

Reason why this thread is marked as suspicious

Thread's entry point VAD's information

Disassembly of thread's entry point

תמונה 1-2, פלט של Threadmap ב-Volatility על KSLSample.vmem של PID: 2460

עשינו סינון לתהליך אחד בלבד כדי להסתיר מידע לא נחוץ. הפלט מחלק תהליכים כאשר כל בלוק מידע מכיל מידע נוסף על ה-Thread-ים של אותו תהליך. תהליך יסומן כחשוד כאשר:

- אין אפילו Thread אחד המצביע ל-Image של תהליך, או שה-Thread היחידי שמצביע לאימג' נמצא בהשהייה.
- Thread חשוד נמצא בתוך התהליך

אם התהליך תואם לקטגוריה הראשונה כל ה-Thread-ים יודפסו בפלט. בקטגוריה השנייה, רק ה-Thread-ים שסומנו כחשודים יודפסו בפלט תחת הכותרת "Reason".

בתמונה 2-1 ניתן לראות את התהליך החשוד svchost.exe שמתאים לשתי הקטגוריות כלומר כל ה-Thread-ים שלו מודפסים (במקרה זה יש רק Thread אחד). Thread זה מסומן כחשוד משום של-VAD אין File Object, והסוג שלו אינו Image File. ההרשאות של ה-VAD הודפסו והן PAGE_READWRITE. VAD זה מכיל את ה-payload הזדוני של התהליך. יש לציין שזה התהליך ששינינו את ההרשאות של ה-VAD על מנת להתחמק מפלאגינים אחרים (ניתן לראות למעלה).

ישנם כמה מקרים בהם ניתן להיתקל בהתראות שווא. דוגמה אחת - תהליך CSRSS שתמיד יופיע כהתראת שווא ולא ניתן לסנן באופן אוטומטי מבלי לסכן את יכולות הזיהוי שלנו. תהליך זה הינו תהליך יחודי מאחר וה-Thread-ים שלו לא מצביעים לאיזשהו image file (אך אין זה אומר שלא ניתן לעשות לו hollowing כך שכדאי תמיד להסתכל על התהליך).

הערה לגבי תמיכה בפלטפורמות שונות

בגרסה זו בחרנו לא לתמוך בפרופילים של XP בגלל העובדה שהפתרון שלנו כרגע לא מותאם טוב לסביבה שכזו. תחת XP, ניתן לכתוב לאזורי זיכרון השייכים ל-Kernel מ-User-mode, ועובדה זו מייתרת הרבה מהנחות המחקר הבסיסיות שלנו. בעיה נוספת קשורה לאיך ש-Kernel, השייך למערכות הפעלה ישנות יותר, מאחסן מידע על Thread-ים ולמרבה הצער חלק מהשוני לא מאפשר ל-plugin לעבוד כהלכה.

Flags אופציונליים

ישנם שני דגלים שניתן להעביר ל-plugin בשביל פלט שונה:

- (--PID) p- - רץ על תהליך ספציפי אחד או קבוצה של תהליכים (מופרד ברווח)
- (--verbose) v- - מדפיס עבור כל תהליך את ה-Thread-ים שלו. הפלט כולל גם תהליכים מזוכים וגם Thread-ים של תהליכים מסומנים. תהליכים מסומנים מודפסים עם הערות ו-Thread-ים מודפסים תחת הכותרת "Reason" (תמונה 2-1)

דגל נוסף נתמך הוא - (--dump-dir) D שמבצע dump ל-VAD שלם הקשור לנקודת התחלה של Thread כלשהו. עובד רק כאשר התהליך מסומן. אם יש JMP/CALL הפונים לטווח זיכרון אחר, שני ה-VADs יודפסו

סיכום

Process Hollowing קיים כבר די הרבה זמן, ובכל זאת יחסית מעט מחקר נעשה מהסוג שאנחנו עשינו, למרות שהמון נוזקות משתמשות בסוג הזרקה כזה. לקחנו את הכלים הטובים ביותר שניתן להשיג והראינו כמה עדיין קל להתחבא מהם. מן ההגיון שלא אמור להיות כל כך קל להתחמק מכלים כאלו במיוחד בהינתן העובדה שהזרקת קוד בשיטה זו היא כל כך נפוצה. על ידי שימוש בשיטה חדשה שמתבססת על מידע ב-Kernel אנחנו נותנים לחוקרים כלי יותר טוב ויותר מדויק להתמודד עם נוזקות בדרכים שלא נוסו בעבר, ובהתאם מקלים על תהליכי הגנה מפני איומים נפוצים.

על הכותבים

אנו קבוצת חוקרים, בה חברים: קייל נס, שחף עטון וליעם שטיין, המתאגדים תחת השם ksl group. שלושתינו מתעסקים בתחום, ויצא לנו לעבוד יחד במסגרת הצבאית. תרגישו חופשי לבקר:

Github: https://github.com/kslgroup

נשמח לכל שאלה / טענה / מענה, בכתובת האימייל:

Ksl.taskforce@gmail.com

קישורים לקריאה נוספת

מידע נוסף על הפרויקט שלנו נמצא ב:

https://github.com/kslgroup/threadmap

את ה-Memory Dump ניתן למצוא:

https://drive.google.com/file/d/0B7v1Owo0v5SYZ016VmVoVFV1elE/view?usp=sharing

או:

https://www.mediafire.com/file/jlmtbbinanuh6jr/KSLSample.rar

Hale Ligh Michael, Case Andres, Levy Jamie, Walter Aaron, The Art of Memory Forensics, Wiley, 2014

Szor Peter, The Art of Computer Virus Research and Defense, Addison Wesley Professional, 2005

Russinovich Mark, Solomon A. David, Ionescu Alex, Windows Internals 6th Edition, Microsoft Press, 2012

מידע על המחקר של hollowfind וה-plugin:

https://cysinfo.com/detecting-deceptive-hollowing-techniques/

מידע על המחקר של malfofind וה-plugin:

https://github.com/volatilityfoundation/community/tree/master/DimaPshoul/

הפרויקט שבו השתמשנו כבסיס ל-PoC נמצא ב:

https://github.com/m0n0ph1/Process-Hollowing