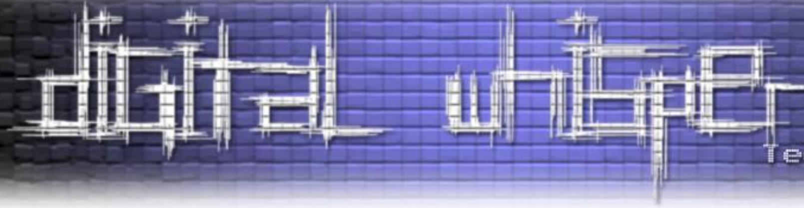


API Set Map & AVRF

מאת שחק שלו

הקדמה

במאמר זה נכיר שני מנגנונים שמצויים במערכת ההפעלה Windows, מהו שימושם האמיתי ואיך אנחנו יכולים לנצל את עצם קיומם. שני המנגנונים, API Sets ו-Application Verifier, קיימים גם ב-Windows 10 והקוד במאמר ייתחס לגרסא העדכנית ביותר של Windows.

API Sets

רעיון ה-API Sets נובע מתהליך גדול יותר שהתרחש ב-Windows הנקרא MinWin. דמיינו גרף של קריאות מערכת המחולק לשכבות לפי סדר התלויות (Dependencies) בין המרכיבים השונים במערכת ההפעלה כך שבאופן טבעי השכבות הנמוכות בגרף מכילות את הפונקציות שיותר קריטיות למערכת ההפעלה. במערכת מתוקנת כל מרכיב בשכבה מסוימת יקרא רק למרכיב בשכבה מתחתיו אך מסתבר שב-Windows לא היה המצב כך פעם ונוצרו מצבים בהם פונקציות "נמוכות" קראו לפונקציות בשכבות גבוהות יותר שלא בהכרח נמצאות כרגע.

MinWin למעשה בא להגדיר ליבה ב-WIndows שיכולה לתפקד בעצמה ללא תלות בגורמים חיצוניים, או במילים אחרות - מאיזו שכבה בגרף נוכל פשוט "לחתוך" מטה ולהישאר עם מערכת מתפקדת עצמאית. בשביל לארגן מחדש את גרף הקריאות ב-Windows כך שקריאות מערכת ייקראו רק לקריאות מערכת אחרות הנמצאות מתחתיהן בגרף, נכנסו לתמונה API Sets.

רעיון ה-API Sets בא ומפריד בין האימפלמנטציה של קריאות מערכת לבין הארכיטקטורה בגרף הפונקציות והוא מבצע זאת בעזרת DLL-ים וירטואליים ("Virtual DLLs"). אם נתקלתם בקבצים עם Imports מוזרים כמו "api-ms-win-core-registry-l1-1-0.dll" או "api-ms-win-core-heap-l1-2-0.dll", אלה הם ה-Virtual DLLs - אשר מחולקים ארכיטקטונית והיררכית לפי סוגי הפונקציות שהם מכילים (למשל לפי פונקציות הקשורות ל-Heap/Registry/Files וכו'). תפקיד ה-API Sets הוא למפות כל Virtual DLL ל-Logical DLL שבו למעשה נמצא המימוש של הקוד - למשל ב-Kernel32.dll.

יחד עם MinWin הגיע גם DLL חדש בשם Kernelbase.dll אשר מייצא את האספקט ה-User Mode-י של רעיון ה-MinWin. פונקציות רבות אשר הוגדרו כ-"MinWin APIs" הועברו מ-DLL-ים אחרים אליו וכעת אותן פונקציות עדיין קיימות באותם DLL-ים אך מכילות רק הפניות ל-KernelBase.dll.

Mapping Virtual DLLs to Logical DLLs

- The mapping of virtual to logical is stored in a schema that's embedded in Apisetschema.dll
  - Kernel reads schema during boot and maps it into every process for quick lookup
  - Loader refers to schema for DLL loads that are pathless to find mapping
- Virtual DLLs images present on system for application compatibility with tools like Dependency Walker
  - Not used by loader

[Windows 7 and Windows Server 2008 R2 Kernel Changes - Dawie Human]

התהליך הופיע לראשונה ב-Windows 7 וב-Windows Server 2008 R2 ומאז הוא משתדרג וגדל בין גרסא לגרסא. ב-Windows 7 דובר על עשרות בודדות של הפניות של ה-DLL-ים (או: Contracts), Windows 8 הגיע כבר עם 365 הפניות, וב-Windows 10 (Build 10586) מדובר כבר על 641 הפניות וכעת התהליך תומך גם בגישת ה-"One Core" של מייקרוסופט.

ApiSetSchema

כל מנגנון ה-API Sets סובב סביב DLL אחד בשם ApiSetSchema.dll שנמצא בתיקיית System32. ApiSetSchema מכיל Section בשם apiset שם מתרחש כל הקסם. ה-Section הוא למעשה טבלת ההמרה בין הספריות הוירטואליות לספריות הלוגיות (סלחו לי על העברות).

Property	Value	Value	Value
Name	.rdata	.apiset	.rsrc
Virtual Size (bytes)	0x00000150 (336)	0x000141F4 (82420)	0x00000400 (1024)
Virtual Address	0x00001000	0x00002000	0x00017000
Raw Size (bytes)	0x00000200 (512)	0x00014200 (82432)	0x00000400 (1024)
Raw Address	0x00000400	0x00000600	0x00014800

ApiSetSchema.dll נטען עם עליית מערכת ההפעלה ע"י **winload!OslpLoadApiSetSchemaImage** וממופה ל-Section בזיכרון אשר בתורו ממופה לכל תהליך חדש במערכת תחת PEB->ApiSetMap. ב-Windows 7 ממופה האזור אמנם כאזור קריאה בלבד אך ניתן לשנות זאת עם שינויי הגנה רגילים. החל מ-Windows 8 כבר נעשה שימוש ב-SEC_NO_CHANGE ובהמשך נראה איך כתוקפים אנחנו נתגבר על זה. חשוב לציין גם כי בעת הטעינה הראשונית בקרנל, נעשית בדיקה של חתימה דיגיטלית של ApiSetSchema.dll.

סיקור מקיף (הרבה) יותר על האספקט הקרנלי של ApiSets ניתן למצוא כאן (נכון ל-Windows 8 32bit).

כתוקף, נוכל לשנות את הטבלה עצמה בזיכרון של התהליך, אך נצטרך להתגבר על כמה מכשולים (לא בהכרח אבטחתיים) אותם אראה כבר בהמשך, והשינוי יהיה הכי אפקטיבי אם נבצע אותו כמה שיותר מוקדם בריצת התהליך (כאן ננצל את מנגנון ה-Application Verifier עליו יורחב בהמשך). כעת לפני שנתחיל לכתוב את הקוד שישנה את הטבלה, נכתוב תוכנית שרק תדפיס לנו את ה-Contracts שקיימים אצלנו בעמדה בשביל להכיר קצת יותר טוב את מבנה ה-API Sets.

הקוד רלוונטי ל-Windows 10 (Build 10586) ולא יתאים לשאר גרסאות Windows, אם כי ניתן להתאים אותו בקלות יחסית לגרסא הרצויה.

ApiSetSchema מורכב מכמה Struct-ים, הראשון שבהם הוא API_SET_NAMESPACE_ARRAY (המבנים לקוחים, עם שינויים קוסמטיים, מהפרויקט המעולה Blackbone):

```
typedef struct _API_SET_NAMESPACE_ARRAY
{
      ULONG Version;
      ULONG Size;
      ULONG Flags;
      ULONG Count;
      ULONG Start;
      ULONG End;
      ULONG Unk[2];
} API_SET_NAMESPACE_ARRAY, *PAPI_SET_NAMESPACE_ARRAY;
```

המשתנים הרלוונטים אלינו:

- Size - הגודל של כל טבלת ה-API Set
- Count - מספר הרשומות בטבלה
- Start - היסט לתחילת הטבלה, למעשה הטבלה מתחילה ישירות אחרי המבנה הנ"ל.

להלן ה-Struct-ים הנוספים בהם נשתמש, לפי סדר היררכי מהעליון לתחתון. ניתן להבין את תפקידם משמם:

```
typedef struct _API_SET_NAMESPACE_ENTRY
{
      ULONG Limit;
      ULONG Size;
} API_SET_NAMESPACE_ENTRY, *PAPI_SET_NAMESPACE_ENTRY;

typedef struct _API_SET_VALUE_ARRAY
{
      ULONG Flags;
      ULONG NameOffset;
      ULONG Unk;
      ULONG NameLength;
      ULONG DataOffset;
      ULONG Count;
} API_SET_VALUE_ARRAY, *PAPI_SET_VALUE_ARRAY;
```

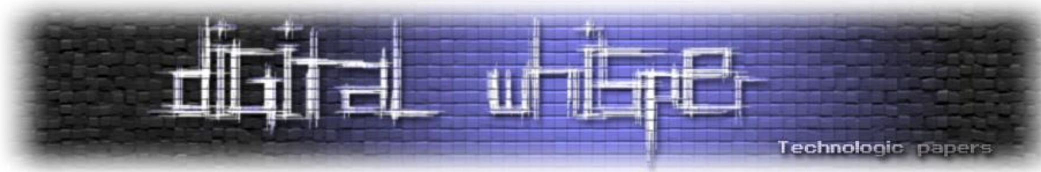
```
typedef struct _API_SET_VALUE_ENTRY
{
       ULONG Flags;
       ULONG NameOffset;
       ULONG NameLength;
       ULONG ValueOffset;
       ULONG ValueLength;
} API_SET_VALUE_ENTRY, *PAPI_SET_VALUE_ENTRY;
```

נתחיל מלהשיג את הכתובת של ה-ApiSet:

```
PEB * peb = NtCurrentTeb()->ProcessEnvironmentBlock;

PAPI_SET_NAMESPACE_ARRAY pApiSetMap = (PAPI_SET_NAMESPACE_ARRAY)peb-
>Reserved9[0]; //ApiSetMap
```

עכשיו כשהמשתנה pApiSetMap מצביע לטבלה, נקרא את מספר הרשומות ונדפיס אותם באיטרציה:

```
for (size_t i = 0; i < pApiSetMap->Count; i++)
  {
    wchar_t apiNameBuf[255] = { 0 };
    wchar_t apiHostNameBuf[255] = { 0 };
    size_t oldValueLen = 0;

   PAPI_SET_NAMESPACE_ENTRY pDescriptor =
(PAPI_SET_NAMESPACE_ENTRY)((PUCHAR)pApiSetMap
+ pApiSetMap->End + i * sizeof(API_SET_NAMESPACE_ENTRY));

   PAPI_SET_VALUE_ARRAY pHostArray = (PAPI_SET_VALUE_ARRAY)((PUCHAR)pApiSetMap +
pApiSetMap->Start + sizeof(API_SET_VALUE_ARRAY) * pDescriptor->Size);

   memcpy(apiNameBuf, pApiSetMap + pHostArray->NameOffset, pHostArray-
>NameLength);

   PAPI_SET_VALUE_ENTRY pHost = (PAPI_SET_VALUE_ENTRY)(pApiSetMap +
pHostArray->DataOffset);

   memcpy(apiHostNameBuf, (PUCHAR)pApiSetMap + pHost->ValueOffset, pHost-
>ValueLength);

   if (pHostArray->Count == 1)
   wprintf(L"[%d] %s -> %s\n", i, apiNameBuf, apiHostNameBuf);
   else
   {
   wchar_t baseApiHostNameBuf[255] = { 0 };

   memcpy(baseApiHostNameBuf, (PUCHAR)pApiSetMap + pHost[1].ValueOffset,
pHost[1].ValueLength);

   wprintf(L"[%d] %s -> %s --> %s\n", i, apiNameBuf, apiHostNameBuf,
baseApiHostNameBuf);
    }

  }


return 0;
```



שימוש לב לשורה:

```
if (pHostArray->Count == 1)
```

לכל Virtual DLL יכולות להיות שתי הפניות משורשרות - זאת אומרת שההפניה הראשונה מפנה לרשומה השניה, שדה Count במבנה API_SET_VALUE_ARRAY יגלה לנו את מספר השרשורים.

אם נריץ נקבל פלט לדוגמא:

```
[597] api-ms-win-core-winrt-errorprivate-l1-1 -> combase.dll
[598] ext-ms-win-net-isoext-l1-1 -> firewallapi.dll
[599] ext-ms-win-shell-browsersettingsync-l1-1 ->
[600] api-ms-win-core-psapi-l1-1 -> kernelbase.dll
[601] ext-ms-win-advapi32-auth-l1-1 -> advapi32.dll
[602] api-ms-win-core-rtlsupport-l1-1 -> ntdll.dll
[603] api-ms-win-core-rtlsupport-l1-2 -> ntdll.dll
[604] ext-ms-win-scesrv-server-l1-1 -> scesrv.dll
[605] ext-ms-win-mf-pal-l1-1 ->
[606] api-ms-win-core-localization-private-l1-1 -> kernelbase.dll
[607] ext-ms-win-appmodel-state-ext-l1-2 -> kernel.appcore.dll
[608] api-ms-win-core-winrt-registration-l1-1 -> combase.dll
[609] api-ms-win-core-path-l1-1 -> kernelbase.dll
[610] api-ms-win-core-processtopology-private-l1-1 -> kernelbase.dll
[611] ext-ms-win-netprovision-netprovfw-l1-1 -> netprovfw.dll
[612] ext-ms-win-ui-viewmanagement-l1-1 ->
[613] ext-ms-win-core-resourcepolicy-l1-1 -> rmclient.dll
[614] api-ms-win-core-stringansi-l1-1 -> kernelbase.dll
[615] api-ms-win-core-file-ansi-l1-1 -> kernel32.dll
[616] api-ms-win-core-file-ansi-l2-1 -> kernel32.dll
[617] ext-ms-win-rtcore-gdi-object-l1-1 -> gdi32.dll
[618] api-ms-win-security-base-l1-1 -> kernelbase.dll
[619] api-ms-win-security-base-l1-2 -> kernelbase.dll
[620] ext-ms-win-session-usermgr-l1-1 -> usermgrcli.dll
[621] ext-ms-win-kernel32-errorhandling-l1-1 --> kernel32.dll --> faultrep.dll
[622] ext-ms-win-wevtapi-eventlog-l1-1 -> wevtapi.dll
```

שמתם לב לרשומות הריקות בלי ההפניה? אלה נמצאות בגלל גישת ה-One Core של מייקרוסופט - תוכלו למצוא כאן Virtual DLL-ים של Windows Phone, XBOX וחבריהם. חלק מהספריות הוירטואליות פשוט יצביעו לספריות לוגיות שונות בין מוצרים שונים וחלק פשוט לא יצביעו לכלום. רשומה מספר 380 במחשב שלי לדוגמא היא ext-ms-win-security-developerunlock-l1-1 אך אין לה שום DLL לוגי בצד השני שמחכה לה, למה? כי developerunlock ככל הנראה מדבר על Developer Unlock למכשירי Windows Phone (תהליך הדומה להשגת root באנדרואיד או Jailbreak ב-iOS).

שינוי ה-API Sets

אז איך נשנה טבלת API Sets?

בתיאוריה, נעבור על כל API Set בטבלה, כאשר נגיע לרשומה אותה נרצה לשנות - נשנה את ה-DLL באותה רשומה וזהו נצא לחגוג. זה בסך הכל נכון אך עם זאת נצטרך להתגבר על כמה מהמורות קטנות בדרך:

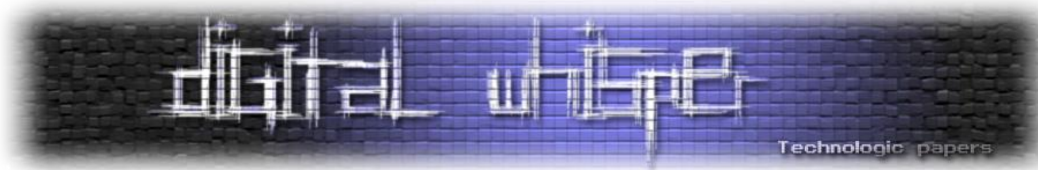
- Windows יחפש את ה-DLL אליו אנחנו מצביעים ברשומה ב-System32. אם נרצה בכל זאת לשנות את הטבלה בלי הרשאות אדמין, נוכל לשים את ה-DLL שלנו בתת תיקייה ב-System32 אליה לא דרושות הרשאות חזקות, ולרשום את הנתיב הזה ברשומה (החל מ-System32).
- מאחר והאיזור עצמו ממופה כ-PAGE_READONLY ו-SEC_NO_CHANGE לא נוכל לשנות את ההגנות, אך נוכל בהחלט לשנות את המצביע מה-PEB ל-API Set Map שאנחנו ניצור.
- ב-Windows 8/8.1 מייקרוסופט תיעדו בשבילנו את ההמרה בין פונקציות API ל-Virtual DLL, תוכלו למצוא את זה כאן, בשאר הגרסאות תוכלו להגיע לפונקציה שתרצו לעשות לה Hook בעזרת ניחוש מושכל.
- שינוי הרשומה לא יכול להתבצע על ידי דריסה פשוטה של המחרוזת שמכילה את שם ה-DLL הלוגי מאחר ואותה מחרוזת משומשת בעוד Contracts. זאת אומרת, שינוי של ה-DLL ב-Contract שאתם רוצים לשנות ישפיע על חוזים אחרים המצביעים לאותו DLL. כדי להתגבר על זה, נוכל להקצות בטבלה החדשה שאנחנו ניצור עוד זיכרון אליו נכתוב את שם ה-DLL שלנו, ונשנה את המצביע ב-Contract אליו במקום לדרוס את המחרוזת המקורית.

שימו לב שיש פונקציות שלא נוכל להשתמש בהן בקוד שלנו כי גם ה-DLL שלנו נתון תחת ההפניות של ה-API Set Map, ז"א שיהיו פונקציות שאם נשתמש בהן ניכנס ללולאה אינסופית מאחר והן יפנו לעצמן. בשביל להתגבר על זה תוכלו לממש את הפונקציות בעצמכם או לצרף לפרויקט שלכם ספריה (.lib) שמפנה לקוד המקורי כך הפונקציות לא יושפעו מההפניה (לא אציג זאת במאמר).

```
#define DLLNAME L"lemon.dll"
#define DLLLEN  9
```

נתחיל עם שני ה-Define-ים פשוטים - שם ה-DLL אותו נזריק ואורך המחרוזת.

```
PEB * peb = NtCurrentTeb()->ProcessEnvironmentBlock;
PVOID ProcessHeap = peb->Reserved4[1];  // ProcessHeap

PAPI_SET_NAMESPACE_ARRAY pApiSetMap = (PAPI_SET_NAMESPACE_ARRAY)peb-
>Reserved9[0]; // ApiSetMap

int realApiSetMapSize = pApiSetMap->Size;

PAPI_SET_NAMESPACE_ARRAY pFakeApiSetMap =
(PAPI_SET_NAMESPACE_ARRAY)RtlAllocateHeap(ProcessHeap, HEAP_ZERO_MEMORY,
realApiSetMapSize + (DLLLEN * 2));
```



```
__memcpy((char *)pFakeApiSetMap, (char *)pApiSetMap, realApiSetMapSize);
__memcpy(((PUCHAR)pFakeApiSetMap + realApiSetMapSize), DLLNAME, (DLLLEN * 2));
```

תחילה, כמו מקודם, נשיג את הכתובת ה-API Set Map, אך כעת היא תשמש אותנו בשביל בשביל העתקתה למקום אחר בזיכרון.

ניצור מצביע בשם pFakeApiSetMap ונאתחל אותו להצביע אל אזור בזיכרון ה-Heap בגודל הטבלה המקורית + גודל מחרוזת שם ה-DLL שלנו כפול 2 (מאחר וכל המחרוזות בטבלה הן Unicode). נעתיק את כל הטבלה אל אזור הזיכרון החדש שיצרנו ולאחר מכן נעתיק מיד לאחר הטבלה בזיכרון את DLLNAME.

```
DbgPrint("\nApiSetMap:%x\n FakeApiSetMap:%x\n\n", peb->Reserved9[0],
pFakeApiSetMap);

for (size_t i = 0; i < pFakeApiSetMap->Count; i++)
{
    wchar_t apiNameBuf[255] = { 0 };
    wchar_t apiHostNameBuf[255] = { 0 };
    size_t oldValueLen = 0;

    PAPI_SET_NAMESPACE_ENTRY pDescriptor =
(PAPI_SET_NAMESPACE_ENTRY)((PUCHAR)pFakeApiSetMap
+ PAPI_SET_VALUE_ARRAY pHostArray =
(PAPI_SET_VALUE_ARRAY)((PUCHAR)pFakeApiSetMap
+ pFakeApiSetMap->Start + sizeof(API_SET_VALUE_ARRAY) * pDescriptor->Size);

    __memcpy((char *)apiNameBuf, (char *)pFakeApiSetMap + pHostArray-
>NameOffset,
  pHostArray->NameLength);

    PAPI_SET_VALUE_ENTRY pHost = (PAPI_SET_VALUE_ENTRY)(pFakeApiSetMap +
pHostArray->DataOffset);

    __memcpy((char *)apiHostNameBuf, (char *)pFakeApiSetMap + pHost-
>ValueOffset,
      pHost->ValueLength);

    if (wcsstr(apiNameBuf, L"api-ms-win-core-file-l2"))
    {
      oldValueLen = pHost->ValueLength;
      pHost->ValueLength = DLLLEN * 2;
      pHost->ValueOffset = realApiSetMapSize;
    }

  }
```

את רוב הקוד כבר ראיתם, השינוי העיקרי הוא רק שינוי המצביעים במקרה ש-apiNameBuf מתאים לרשומה אותה נרצה לשנות. לבסוף נקנח עם:

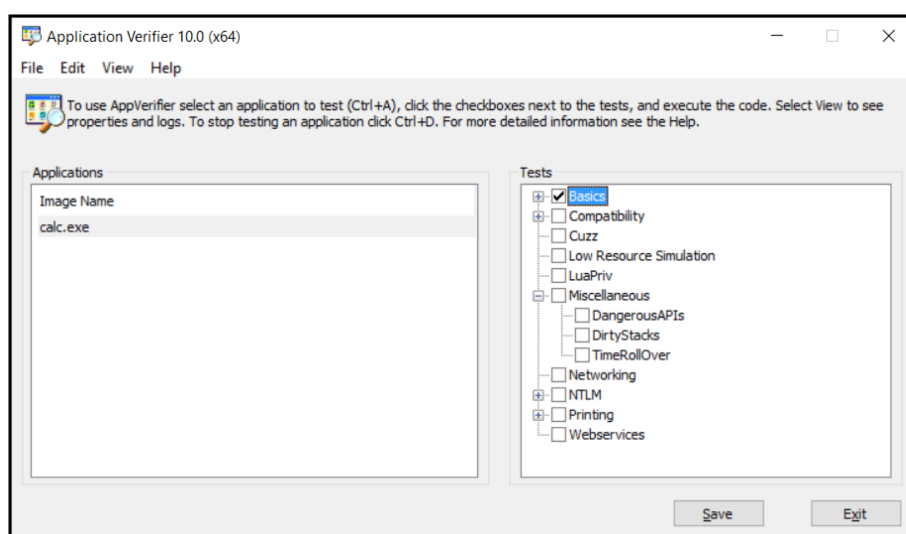
```
peb->Reserved9[0] = pFakeApiSetMap;
return 0;
```

למעשה, פשוט שינינו את הפוינטר ב-PEB אל הטבלה החדשה.

אמרנו שעדיף לשנות את ה-API Set Map כמה שיותר מוקדם בריצת התהליך כדי שהשינוי באמת יהיה אפקטיבי, יש מספר דרכים לשנות מרחב כתובות של תהליך בתחילת הריצה שלו אבל אני חושב שיהיה נחמד אם נשתמש בשיטה לא כל כך מוכרת המנצלת מנגנון אחר ב-Windows - Application Verifier.

Application Verifier

ה-Application Verifier הוא כלי המגיע עם Windows ומאפשר לנו להוציא מידע דיאגנוסטי רב על תוכנות הרצות במחשב כגון הקצאות זיכרון שמתרחשות, Handles שנפתחים/נסגרים, אירועי רשת וכו'.
כל סט כזה של בדיקות (ניתן לראות בחלון הימני בתמונה מטה) נקרא Provider וניתן לכתוב אחד כזה גם משלנו.



מה שקורה מאחורי הקלעים זה שבעת יצירת התהליך Windows בודק את Image File Execution Options (שאולי כבר הכרתם) ומחפש תחת המפתח של התהליך שיצרנו את הערכים הבאים:

- GlobalFlag עם הערך FLG_APPLICATION_VERIFIER - 0x100
- VerifierDlls עם שם Provider כלשהו הנמצא ב-System32
- VerifierDebug עם הערך 0xffffffff (לא חובה, גורם להדפסה של עוד מידע דיאגנוסטי) (כל הערכים הם REG_SZ)

Verifier.dll (הלוא הוא Application Verifier בעצמו) נטען מוקדם מאד בריצת התהליך, ישירות אחר ntdll.dll, וטוען את שאר ה-Provider-ים הנמצאים בערך VerifierDlls.

```
CommandLine: C:\Windows\System32\calc.exe

************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Symbol search path is: srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00007ff7`a20f0000 00007ff7`a20fd000   calc.exe
ModLoad: 00007ff8`2ba50000 00007ff8`2bc11000   ntdll.dll
ModLoad: 00007ff8`0b400000 00007ff8`0b46d000   C:\WINDOWS\system32\verifier.dll
Page heap: pid 0x3DA0: page heap enabled with flags 0x2.
AVRF: calc.exe: pid 0x3DA0: flags 0x48004: application verifier enabled
AVRF: verifier dll `verifier.dll'
AVRF: initialized provider verifier.dll (descriptor @ 00007FF80B430CB0)
AVRF: verifier dll `myvrf.dll'
ModLoad: 00007ff8`21e00000 00007ff8`21e07000   C:\WINDOWS\SYSTEM32\myvrf.dll
AVRF: myvrf.dll @ 00007FF821E00000: entry point @ 00007FF821E014D0 .
AVRF: hooked dll entry point for dll myvrf.dll
AVRF: dll entry @ 00007FF821E014D0 (C:\WINDOWS\SYSTEM32\myvrf.dll, 4)


#Peb Address:364000
ApiSetMap:85550000
FakeApiSetMap:87a1ddf0
```

Provider אמיתי למעשה מספק ל-Verifier.dll מבנה בשם RTL_VERIFIER_PROVIDER_DESCRIPTOR המתאר את הפונקציות להן הוא רוצה לבצע Hook. ה-Application Verifier יבצע IAT Hooking לאותן פונקציות ויחזיר לנו את הכתובת המקורית של אותן פונקציות. תודה רבה Windows :).

לנו אין צורך באמת לכתוב DLL העונה להגדרות של Provider כי אנחנו רוצים רק לנצל את הטעינה המוקדמת של Application Verifier (אבל בכל זאת אצרף בהמשך קוד כזה).

DLL_PROCESS_VERIFIER

איך DLL יודע שהוא נטען כ-Provider? כנראה אתם מכירים את ארבעת המקרים/"סיבות" ש-DLL יכול להיקרא:

- DLL_PROCESS_ATTACH
- DLL_PROCESS_DETACH
- DLL_THREAD_ATTACH
- DLL_THREAD_DETACH

אז מסתבר שיש עוד מקרה - DLL_PROCESS_VERIFIER בעל הערך 4, אז ה-DllMain שלנו ייראה כך:

```
BOOL WINAPI DllMain(
      _In_ HINSTANCE hinstDLL,
      _In_ DWORD fdwReason,
      _In_ LPVOID lpvReserved
      )
{
      UNREFERENCED_PARAMETER(hinstDLL);
PRTL_VERIFIER_PROVIDER_DESCRIPTOR* pVPD =
(PRTL_VERIFIER_PROVIDER_DESCRIPTOR *)lpvReserved;

      switch (fdwReason) {

      case DLL_PROCESS_VERIFIER:
```

```
              // Insert balagan
changeApiSet();
              break;
       }
       return TRUE;
}
```

כאשר changeApiSet הוא [הקוד שרשמנו מקודם](#).

בונוס

איך נוכל לבצע Hooking בעזרת Application Verifier? אלה המבנים בהם נשתמש:

```
typedef struct _RTL_VERIFIER_THUNK_DESCRIPTOR {
       PCHAR ThunkName;
       PVOID ThunkOldAddress;
       PVOID ThunkNewAddress;
} RTL_VERIFIER_THUNK_DESCRIPTOR, *PRTL_VERIFIER_THUNK_DESCRIPTOR;

typedef struct _RTL_VERIFIER_DLL_DESCRIPTOR {
       PWCHAR DllName;
       DWORD DllFlags;
       PVOID DllAddress;
       PRTL_VERIFIER_THUNK_DESCRIPTOR DllThunks;
} RTL_VERIFIER_DLL_DESCRIPTOR, *PRTL_VERIFIER_DLL_DESCRIPTOR;

typedef struct _RTL_VERIFIER_PROVIDER_DESCRIPTOR {
       DWORD Length;
       PRTL_VERIFIER_DLL_DESCRIPTOR ProviderDlls;
       RTL_VERIFIER_DLL_LOAD_CALLBACK ProviderDllLoadCallback;
       RTL_VERIFIER_DLL_UNLOAD_CALLBACK ProviderDllUnloadCallback;
       PWSTR VerifierImage;
       DWORD VerifierFlags;
       DWORD VerifierDebug;
       PVOID RtlpGetStackTraceAddress;
       PVOID RtlpDebugPageHeapCreate;
       PVOID RtlpDebugPageHeapDestroy;
       RTL_VERIFIER_NTDLLHEAPFREE_CALLBACK ProviderNtdllHeapFreeCallback;
} RTL_VERIFIER_PROVIDER_DESCRIPTOR, *PRTL_VERIFIER_PROVIDER_DESCRIPTOR;
```

RTL_VERIFIER_PROVIDER_DESCRIPTOR יהיה לבסוף ה-Struct אותו נחזיר ל-Application Verifier. הוא יכיל מערך של RTL_VERIFIER_DLL_DESCRIPTOR (כאשר החבר האחרון במערך יהיה מאופס) שכל אחד מהם מתאר DLL אשר נרצה לבצע את ה-Hook על פונקציה אחת או יותר ממנו.

RTL_VERIFIER_DLL_DESCRIPTOR בתורו יכיל מערך של THUNK_DESCRIPTOR-ים (שוב, האחרון יהיה מאופס) כאשר כל THUNK_DESCRIPTOR מכיל את שם הפונקציה אותה הוא מבקש "לתפוס", משתנה אליו תיכתב בכתובת המקורית של הפונקציה והכתובת של הפונקציה החדשה שתתבצע במקום המקורית.

נחבר הכל:

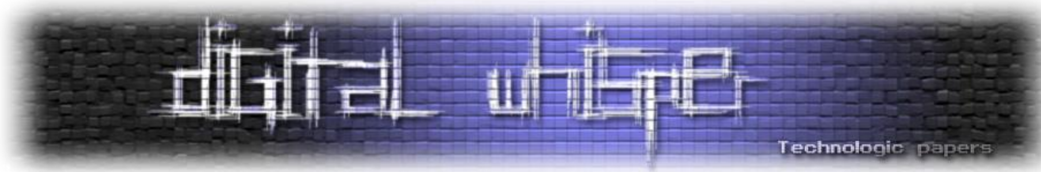
```
int WINAPI MessageBoxWHook(HWND hWnd, LPWSTR lpText, LPWSTR lpCaption,
UINT uType);

static RTL_VERIFIER_THUNK_DESCRIPTOR avrfThunkDesc[] =
{ { "MessageBoxW", NULL, (PVOID)(ULONG_PTR)MessageBoxWHook } };
static RTL_VERIFIER_DLL_DESCRIPTOR avrfDllDesc[] =
{ { L"user32.dll", 0, NULL, avrfThunkDesc } };
static RTL_VERIFIER_PROVIDER_DESCRIPTOR avrfDescriptor =
{ sizeof(RTL_VERIFIER_PROVIDER_DESCRIPTOR), avrfDllDesc };

BOOL WINAPI DllMain(
	_In_ HINSTANCE hinstDLL,
	_In_ DWORD fdwReason,
	_In_ LPVOID lpvReserved
	)
{
	PRTL_VERIFIER_PROVIDER_DESCRIPTOR* pVPD =
(PRTL_VERIFIER_PROVIDER_DESCRIPTOR *)lpvReserved;

	UNREFERENCED_PARAMETER(hinstDLL);

	switch (fdwReason) {

	case DLL_PROCESS_VERIFIER:
		*pVPD = &avrfDescriptor;
		break;
	}
	return TRUE;
}

int WINAPI MessageBoxWHook(HWND hWnd, LPWSTR lpText, LPWSTR lpCaption,
UINT uType)
{
	((OldMessageBoxW)avrfThunkDesc[0].ThunkOldAddress)(hWnd, L"Lemon
Waffle", lpText, uType);
	return 0;
}
```

וזהו, קמפלו את ה-DLL, מלאו את הפרטים תחת Image File Execution Options ותהנו.

שימו לב שבגלל שנטענו ישר לאחר ntdll אין לנו גישה לפונקציות אחרות. אם בכל זאת נרצה להשתמש בפונקציות למשל מ-user32.dll אז נוכל לרשום פונקציית callback תחת `ProviderDllLoadCallback` עם הפרוטוטייפ:

```
typedef VOID(NTAPI * RTL_VERIFIER_DLL_LOAD_CALLBACK) (PWSTR DllName,
PVOID DllBase, SIZE_T DllSize, PVOID Reserved);
```

שתודיע לנו על כל DLL חדש שנטען לתהליך, ברגע שנזהה כי ה-DLL שאנחנו מעוניינים בו נטען נוכל להמשיך ולהשתמש בפונקציות שלו (טעינה בזמן ריצה).



מילים אחרונות

במאמר ניסיתי להכיר לכם בצורה פשוטה ומובנת את שני המנגנונים API Set ו-Application Verifier, איך הם עובדים, איך ניתן להשתמש בזה לרעה, איך נוכל לגרום להם לעבוד יחדיו ולבנות POC של ניצול המנגנונים.

את כלל הקוד ניתן למצוא ב-Git.

לפניות או שאלות ניתן לפנות לכתובת: shahakshalev@gmail.com

תודה רבה ל**איתי כהן** ו**עומר אלימלך** שטרחו ונתנו מזמנם לקרוא את הטיוטה ונתנו הצעות לשיפורים.

קריאה נוספת:

MinWin:

- https://channel9.msdn.com/shows/Going+Deep/Mark-Russinovich-Inside-Windows-7/
- http://windows-now.com/blogs/robert/mark-russinovich-explains-minwin-once-and-for-all.aspx

ApiSet:

- http://blog.quarkslab.com/runtime-dll-name-resolution-apisetschema-part-i.html
- http://blog.quarkslab.com/runtime-dll-name-resolution-apisetschema-part-ii.html
- http://www.alex-ionescu.com/Estoteric%20Hooks.pdf

Application Verifier:

- http://www.kernelmode.info/forum/viewtopic.php?f=15&t=3418
- https://msdn.microsoft.com/en-us/library/windows/hardware/ff538115(v=vs.85).aspx
- http://blogs.msdn.com/b/reiley/archive/2012/08/17/a-debugging-approach-to-application-verifier.aspx