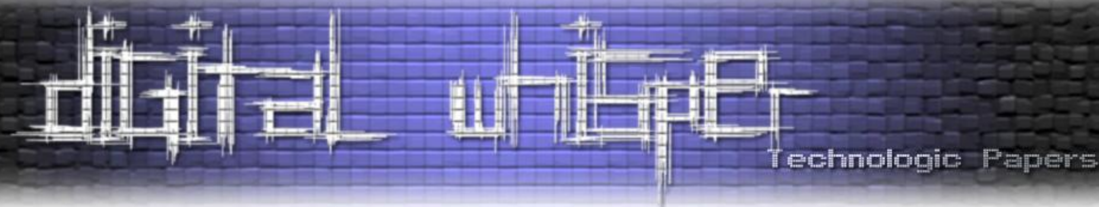


The Art of Exploitation: Windows 7 DEP & ASLR Bypass

מאת אביב ברזילי / sNiGhT

הקדמה

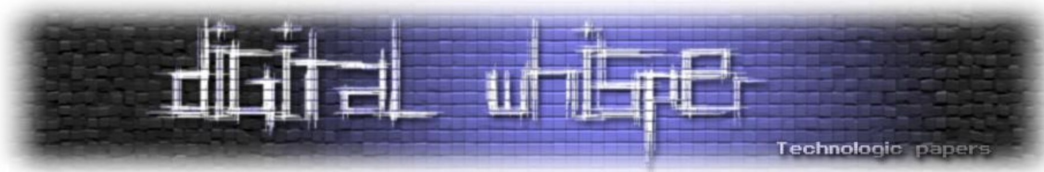
מאז יציאתן של הגרסאות האחרונות של Windows נשאלה לא פעם השאלה: "האם חולשות Buffer Overflow הגיעו לקיצן?"

החולשות ב-User Mode מתחלקות לחולשות ב-Heap ולחולשות במחסנית.

- לגבי ניצול של Heap Overflows השאלה איננה האם הם נגמרו אלא האם ניתן להחיות אותן, וזאת מכיוון שניצול חולשה שכזאת עם כל ההגנות שיש על ה-Heap היא מאוד נדירה.
- לגבי הניצול של Stack Overflows כאן הסיפור שונה לגמרי, למרות שהתווספו לא מעט הגנות אנו רואים מדי-יום שיוצאים אקספלויטים שמנצלים חולשות כאלה- הן במערכות ישנות והן במערכות חדשות ומוגנות. במילים אחרות: **חולשות אלה עדיין חיות ובועטות**.

במאמר זה נסקור חלק מההגנות הקיימות ונציג שיטות לעקיפתן. המאמר יעסוק בעקיפה של ההגנות ASLR ו-DEP. לא נעסוק במאמר זה בעקיפת GS.

למתחילים שבינינו, מומלץ לקרוא קודם את המאמר שכתב שי רוד בגליון 11 של Digital Whisper: Buffer Overflows 101 על-מנת לקבל בסיס בשביל המאמר.



הגנות על המחסנית במערכת Windows 7

במערכת Windows 7 יש 4 הגנות על המחסנית: GS, SafeSEH ,DEP, ASLR שאפשר לחלק אותן לשתי זוגות:

- GS ,SafeSEH
- ASLR ,DEP

כל הגנה שתהיה ללא בת זוגה תהיה כמעט ולא אפקטיבית.

הסבר קצר על ההגנות:

- **GS** - תפקיד ההגנה למנוע מהתוקף להשתמש בכתובת חזרה משוכתבת. הבדיקה מתבצעת באמצעות אתחול ערך Cookie רנדומלי בתוך המחסנית לפני ה-Return Address ובדיקה של תקינות הערך בסיום הפונקציה כך שאם הוא נדרס לא תתבצע פקודת ה-RET ויזרק Exception, ההגנה מכילה עוד כמה אלגוריתמים שתפקידם למנוע דריסה של מצביעים.
- **Safe Structured Exception Handling** - בקיצור SafeSEH, תפקידה למנוע קפיצה ל-Exception Handlers משוכתבים, הבדיקה מתבצעת באמצעות טבלה שמכילה את כתובות ה-Handlers החוקיים, במידה ואחד מהם שונה במהלך הריצה לא תתבצע קפיצה אליו.
- **SEHOP** - הגנה מתקדמת נוספת ל-SEH נקראת SEHOP שמוסיפה גם בדיקה לתקינות הרשימה המקושרת של ה-Handlers באמצעות Cookie.
- **Data Execution Prevention** - בקיצור DEP, תפקידה למנוע הרצה של קוד באזורי נתונים, מטרת ההגנה היא למנוע הרצה של Shellcode.
- **Address space layout randomization** - בקיצור ASLR, תפקידה לבצע רנדומיזציה של הכתובות בקוד לחלק ממבני הנתונים, כדי למנוע מהתוקף לאתר ולקפוץ אל ה-Shellcode שלו.

אנו מתייחסים ל-SafeSEH ו-GS ביחד מכיוון שהדרך לעקוף את ההגנה של ה-GS הייתה באמצעות שיכתוב ה-SEH ומכאן נוצר הרעיון של SafeSEH (וגם ה-Heap overflow תרם את חלקו).

על הקשר בין DEP ו-ASLR נרחיב בהמשך המאמר.

הרחבה על ההגנות

:DEP

הגנה לא ממש חדשה אבל במערכות Windows היא נכנסה לשימוש רק בימי XP SP2. מטרתה לבטל מתן הרשאות ריצה לאזורים בזיכרון שאינם ראויים לכך כמו אזורים נתונים למיניהם.

המימוש שלה נעשה באמצעות תמיכה של המעבד ב-NX bit אך מאפשר גם מימוש ברמת התוכנה במידה ואין תמיכה מצד החומרה. בימינו כל המעבדים תומכים באופציה הזאת.

סיבית ה-NX יושבת ב-Page Table. במידה והסיבית מכובה, היא אומרת "לא ניתן להריץ קוד בדף", כלומר המעבד לא יאפשר הרצה של הפקודות שיש בדף במידה ו-EIP מצביע לשם ובמקרה כזה תזרק:

```
access violation exception.
```

אפשרות זו מונעת הרצה של קטעי קוד על אזורי נתונים כמו המחסנית ו-Heap, לכן ברוב המקרים ה-shellcode שממוקם שם פשוט לא ירוץ וייזרק Exception ברגע שה-EIP יצביע לשם.

קיימים ארבעה מצבים של ההגנה במערכת:

- Optin: ההגנה בברירת המחדל: רק תהליכי מערכת ותהליכים שמבקשים לקבל את ההגנה מקבלים, אחרים לא מקבלים. כמו-כן המנגנון יכול לכבות את עצמו תוך כדי ריצה.
- Optout: כל התהליכים מקבלים הגנה חוץ מכאלה שנכללים ברשימה.
- AlwaysOn: כל התהליכים רצים במצב של DEP ואין אפשרות להוריד את זה תוך כדי ריצה.
- AlwaysOff: כל התהליכים לא רצים ב-DEP ולא ניתן להפעיל אותו תוך כדי ריצה.

את השניים הראשונים ניתן לשנות בממשק בלוח הבקרה ואת השניים האחרונים דרך bcdedit או בכלי אחר של מיקרוסופט הנקרא Emet.

במאמר זה אנו מניחים שהמערכת עובדת ב-Opt-In.

ASLR

הגנה ישנה, כמעט בת 10 שנים, אך עם זאת היא נכנסה לשימוש במערכות מיקרוסופט רק בגרסת Windows Vista. תפקידה לבצע רנדומיזציה של הזיכרון - במילים אחרות טעינה לכתובות בסיס רנדומליות של ה-Image שלנו, ה-DLL-ים שנטענים אליו, המחסנית, ה-Heap ומבנים נוספים לניהול התהליך, בעיקר כאלו שבד"כ התוקף עושה בהם שימוש כשהוא מנצל חולשה.

למה צריך טעינה רנדומלית כזו?

בעבר כשניצלו חולשות היו משתמשים בכתובת זיכרון קבועות. לדוגמא , אם היינו רוצים לקפוץ במקרה של Stack Overflow ל-shellcode שלנו היינו מחפשים פקודה בזיכרון שתקפיץ אותנו אליו (“מקפצה”) כמו Jmp esp, לצורך כך היינו משתמשים בכתובת קבועה של פקודה באחד ה-DLL-ים הטעונים או בתוך ה-image עצמו. ה-ASLR יהפוך חולשה כזאת לסטטיסטית בלבד, כיוון שהכתובת בסיס תשתנה בכל טעינה של המערכת.

להדגמה, טעינה של kernel32.dll ב-3 הפעלות של המערכת ב- Windows 7:

Base	Size	Entry	Name	File version
77940000	000D4000	779910E5	kernel32	6.1.7600.16385
76B60000	000D4000	76BB10E5	kernel32	6.1.7600.16385
76D70000	000D4000	76DC10E5	kernel32	6.1.7600.16385

כמו-כן הרנדומליות בטעינת המחסנית יגרום לשינוי כתובת הבסיס של המחסנית כך שגם הכתובת שבה יושב ה-shellcode היא רנדומלית, מה שימנע ממנו לקפוץ ל-shellcode גם על-ידי שיכתוב של ה-RET עם הכתובת של ה-shellcode.

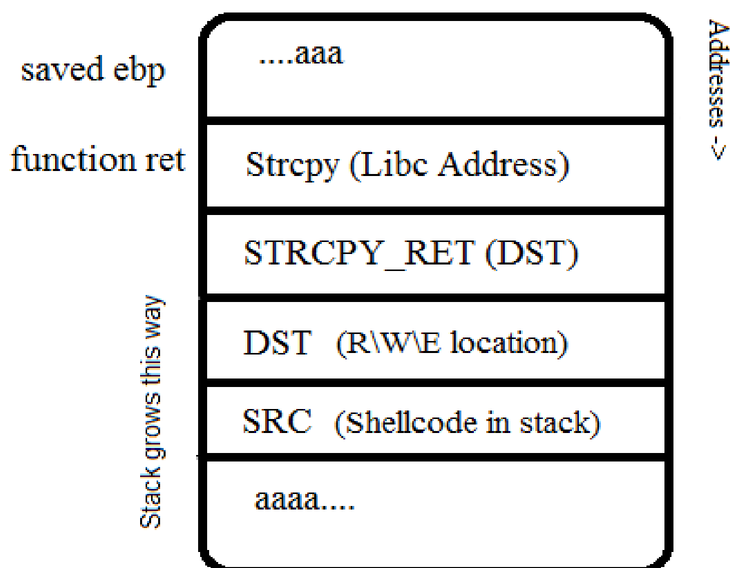
חשיבות שילוב בין שתי ההגנות

עוד בשנת 1998 פורסם ב-BugTraq המאמר המפורסם "Defeating Solar Designer non-executable stack patch" ובו הוצגה שיטה כיצד לעקוף את ההגנה בלינוקס שלא מאפשרת הרצה של קוד מהמחסנית (DEP ב-Win32). השיטה ידועה בשם Ret2Libc, והעיקרון שלה הוא דריסת כתובת החזרה עם כתובת של פונקציית ספרייה כך שבעת סיום הפונקציה תתבצע קריאה לפונקציית הספרייה. לדוגמא: במידה ואנו רוצים לנצל חולשה שהיא לוקאלית נוכל לשכתב את RET עם הכתובת של system(), במידה ואנו מעוניינים לייצר אפשרות של הרצת קוד משלנו נשכתב את RET עם הכתובת של strcpy כדי שנוכל לבצע העתקה של ה-shellcode למקום שכן יש לו הרשאות של כתיבה והרצה ולהריץ אותו משם.

אם ניקח את האופציה השנייה, נציף את המחסנית ונשכתב אותה באופן הבא:

1) נשכתב את ה-RET עם כתובת של strcpy.
2) ב-4 בתים הבאים נכתוב את הכתובת החדשה של ה-shellcode.
3) ב-8 הבתים הבאים נכתוב את שני הכתובות של האורגמנטים ל-strcpy.

המחסנית תראה כך:



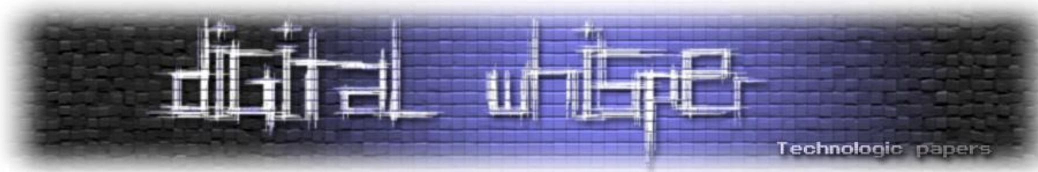
אנו צריכים שהמחסנית תראה כאילו התבצע הקטע הקוד הבא:

```
push SRC
push DST
call strcpy ; ( push STRCPY_RET)
```

לאחר הדריסה שלנו כאשר הפונקציה תסיים את פעולתה ותבצע RET, היא תקפוץ לכתובת של תחילת הפונקציה Strcpy שתבצע את ההעתקה של ה-shellcode שלנו מהמחסנית למקום בדאטה-סיגנמט בעל הרשאות כתיבה והרצה ולאחר מכן כאשר Strcpy תבצע בעצמה RET היא תשלוף את ה-Ret החדש שנתנו לה (STRCPY_RET) שהוא הכתובת של ה-shellcode החדש שלנו (DST) וה-shellcode שלנו ירוץ, כך בעצם התגברנו על ההגנה.

שיטת העקיפה הזאת תנוטרל אם נוסיף את ההגנה של ה-ASLR, כיוון שהמקום שמכיל את ההרשאות של הכתיבה-הרצאה ישתנה בכל הרצה של התוכנית או של המערכת לא נוכל לדעת להיכן להעתיק את ה-shellcode.

נוספת לכך העובדה שדי נדיר למצוא מקום היום בזיכרון בעל הרשאות של כתיבה-הרצה ולכן אנחנו כמעט מנוטרלים.



DEP Bypass

כאמור ה-ASLR היא הגנה חדשה יחסית במערכות ה-Windows אבל עוד הרבה לפניה נאלצו להתמודד עם בעיית ה-DEP שקדם לה.

היה אפשר להתמודד איתה באותה השיטה שהוצגה לעיל- כלומר העתקת ה-shellcode למקום בעל הרשאות כתיבה-הרצה, אך זה לא אפשרי כיוון שבד"כ לא מצויים אזורים בזיכרון שיש להם הרשאות כאלה.

הפתרונות שהוצגו לרוב כדי להתגבר על המגבלה של ה-DEP היו ניסיונות לגשת באמצעות ret2libc למספר מקומות ידועים במודולים הטעונים לזיכרון כדי לבטל את ההגנה של ה-DEP או לחילופין על-מנת להקצות מקום בעל הרשאות של כתיבה-הרצה כדי שנוכל להריץ משם את ה-shellcode.

<u>לדוגמא:</u>

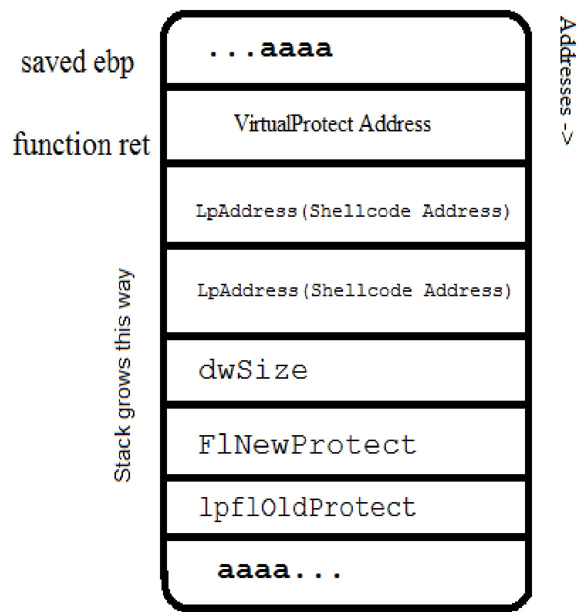
שימוש בפונקציות כמו NtSetInformationProcess כדי לבטל את ההגנה של ה-DEP עבור הפרוסס שלנו או VirtualProtect כדי לתת הרשאות הרצה לאזור שבו יושב ה-shellcode שלנו.

<u>איך זה מתבצע?</u>

ניקח את הדוגמא של שימוש ב-VirtualProtect, פונקציה המאפשרת לשנות את ההרשאות עבור אזורים ממופים בזיכרון. לפי ה-MSDN אלו הערכים שהפונקציה צריכה לקבל:

```
BOOL WINAPI VirtualProtect(
  __in   LPVOID lpAddress,
  __in   SIZE_T dwSize,
  __in   DWORD flNewProtect,
  __out  PDWORD lpflOldProtect
);
```

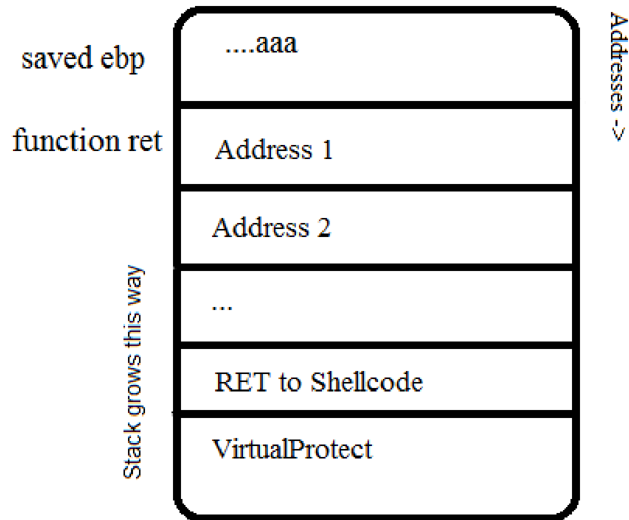
אם היה באפשרותנו לסדר את הערכים בצורה ישירה, היינו את מציפים את המחסנית באופן כזה :



הבעיה היא (וזה עוד לפני שאנחנו מדברים על ASLR) שאנו צריכים להשתמש בערך 0x00 למשל ב-dwSize, כי אם לא נוכל להשתמש ב-0x00 אנו נהיה חייבים לתת לו ערך מאוד גדול, דבר שיגרום לפונקציה להחזיר שגיאה כיוון שרוב מרחב הזיכרון שאנחנו נבקש לשנות את הרשאותיו יהיה בכלל לא ממופה (השטח הכי קטן שנוכל לבקש הוא 0x01010101 בתים...).

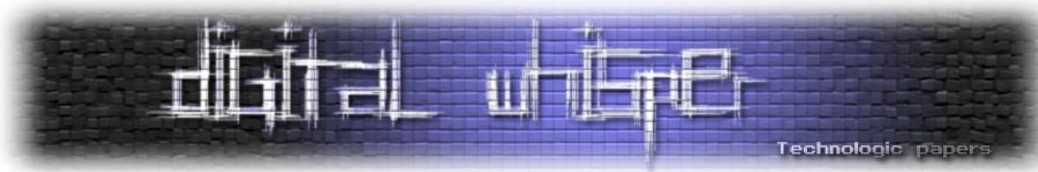
מי שעסק קצת בניצול חולשות כאלו יודע שבדרך כלל אין אפשרות להשתמש בערך 0x00 בתוך ה-shellcode.

לכן צריך להגיע למצב שבו המחסנית תהיה מסודרת עם כל הארגומנטים לפונקציה VirtualProtect, אבל כיוון שאין באפשרותנו להריץ פקודות אלא רק לבצע קריאות לאזורים בזיכרון אנו נבחר לקפוץ לאזורים בזיכרון שיש להם הרשאות הרצה כדי שיבצעו עבורנו מספר פקודות מסוימות, כך שבסופו של דבר אנחנו נסדר את כל הפרמטרים שנצטרך במחסנית ואז נוכל לקפוץ לפונקציה VirtualProtect וכאשר היא תסיים את הפעולה שלה, כלומר תיתן הרשאות כתיבה לאזור בזיכרון של ה-shellcode שלנו, נדאג שה-RET שלה יצביע למקפצה שתקפיץ אותנו ל-shellcode.



נסביר איך זה עובד: לאחר השכתוב כאשר הפונקציה של התוכנית הפגיעה מסיימת את פעולתה היא מבצעת RET וקופצת ל-ADDRESS1 שהכנסנו לה, ואחרי שהיא תסיים את הקטע קוד באזור של ADDRESS1, כלומר תגיע לפקודה RET, היא תחזור לכתובת הבאה שמופיעה במחסנית, כלומר ADDRESS2, ותתחיל לרוץ שם עד שהיא תגיע ל-RET ותשלוף את ADDRESS3 וכך הלאה, עד שהיא תשלוף באמצעות-RET את הכתובת של VirtualProtect שתשלוף בסוף פעולתה בעת ביצוע פקודת ה-RET שלה את הכתובת של המקפצה ל-shellcode שלנו, כל זאת כמובן כל עוד נדאג ש-ESP לא ישתנה.

כמובן שבפועל זה לא כזה "נקי" כמו שזה נראה כאן.



(ROP :בקיצור) Return Oriented Exploitation

השאלה שעולה עכשיו היא לאיזה כתובת אנו מעוניינים לקפוץ, והתשובה היא פשוטה.

אנו נחפש כתובות שמבצעות פעולה "אלמנטרית" אחת ומיד אח"כ RET ללא LEAVE או mov esp,ebp וכו'.

לדוגמא :

```
7C34290A    33C0                XOR EAX,EAX
7C34290C    C3                  RET
```

```
7C36B413    83E0 20             AND EAX,20
7C36B416    C3                  RET
```

```
7C36FB1F    83C0 FE             ADD EAX,-2
7C36FB22    C3                  RET
```

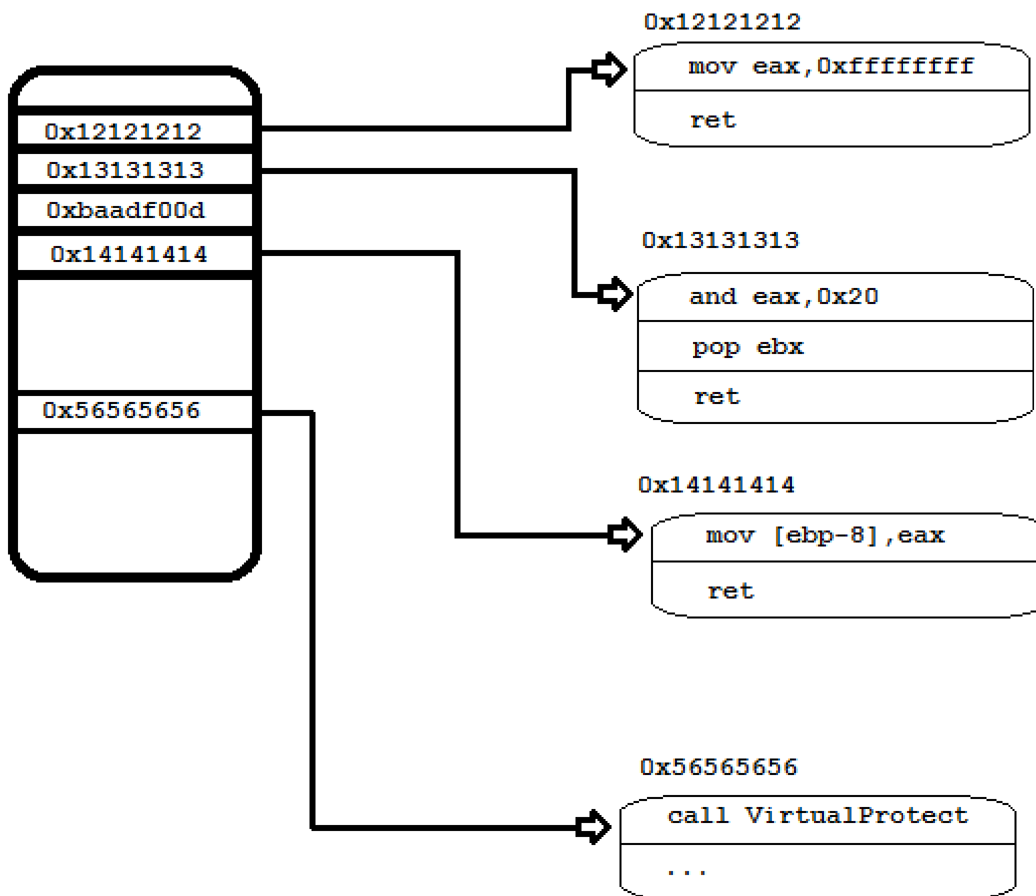
```
7C18BBF6    83C8 FF             OR EAX,FFFFFFFF
7C18BBF9    C3                  RET
```

כל מקום כזה בזיכרון אם נקפוץ אליו יבצע את הפקודה ה"אלמנטרית" שהוא אמור לבצע ומיד יקפוץ לכתובת הבאה (שכמובן אנו אלה שנותנים לו אותה) באמצעות RET.

מכיוון שאין לנו יותר מידי "מציאות" כאלה אנו נאלץ לפעמים להשתמש במשהו בסגנון:

```
7C3419FF    8BC7                MOV EAX,EDI
7C341A01    5F                  POP EDI      (pop junk)
7C341A02    5E                  POP ESI      (pop junk)
7C341A03    C3                  RET          (pop NextRet)
```

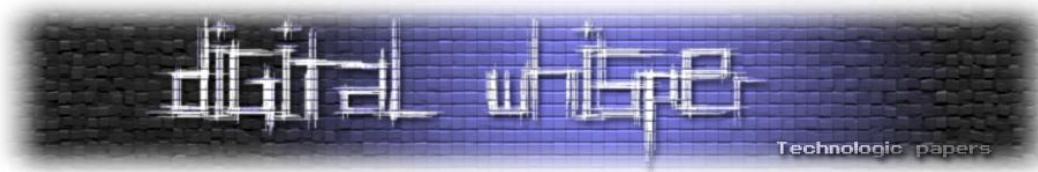
וזה אומר שאנו צריכים להוסיף שני ערכים של "זבל" שישלפו מתוך ה-ROP Shellcode שלנו בזמן שתתבצע קפיצה לאזור כזה.



ישנו סקריפט מעולה ל-Immunity Debugger שעוזר למצוא כתובות יעודיות כאלה ושמו: pvefindaddr.

```
7C345246    30     # ADD ESP,30 # POP EDX # RET                                            MSVCR71.dll
7C34523F    30     # ADD ESP,30 # POP EDX # RET                                            MSVCR71.dll
7C345246    30     # ADD ESP,30 # POP EDX # RET                                            MSVCR71.dll
7C345246    30     # ADD ESP,30 # POP EDX # RET                                            MSVCR71.dll
7C34533C    10     # ADD ESP,10 # FLD QWORD PTR SS:[ESP+4] # RET                           MSVCR71.dll
7C345457    10     # ADD ESP,10 # FLD QWORD PTR SS:[ESP+4] # RET                           MSVCR71.dll
7C345600    1C     # ADD ESP,1C # RET                                                      MSVCR71.dll
7C34572B    EBX    # ADD ESP,EBX # ADD EAX,MSVCR71.7C390144 # RET                          MSVCR71.dll
7C34588F    0A     # ADD ESP,0A # TEST EAX,0 # MOV EAX,DWORD PTR DS:[EDX+4] # RET          MSVCR71.dll
7C345C09    8      # ADD ESP,8 # FADD QWORD PTR SS:[ESP+4] # RET                           MSVCR71.dll
7C345C09    8      # ADD ESP,8 # FADD QWORD PTR SS:[ESP+4] # RET                           MSVCR71.dll
7C345C6E    10     # ADD ESP,10 # FLD QWORD PTR SS:[ESP+4] # RET                           MSVCR71.dll
7C346169    10     # ADD ESP,10 # RET                                                      MSVCR71.dll
7C346249    10     # ADD ESP,10 # RET                                                      MSVCR71.dll
7C346239    1C     # ADD ESP,1C # SUB ESP,10 # MOVLPS QWORD PTR SS:[ESP+4],XMM0 # FLD QWORD MSVCR71.dll
7C3462E3    10     # ADD ESP,10 # RET                                                      MSVCR71.dll
7C34651D    10     # ADD ESP,10 # RET                                                      MSVCR71.dll
7C3465FE    1C     # ADD ESP,1C # RET                                                      MSVCR71.dll
7C3467A1    10     # ADD ESP,10 # RET                                                      MSVCR71.dll
7C346882    1C     # ADD ESP,1C # RET                                                      MSVCR71.dll
7C346C16    10     # ADD ESP,10 # RET                                                      MSVCR71.dll
7C3470B9    10     # ADD ESP,10 # RET                                                      MSVCR71.dll
7C347100    1C     # ADD ESP,1C # RET                                                      MSVCR71.dll
7C3472DF    10     # ADD ESP,10 # RET                                                      MSVCR71.dll
7C347329    10     # ADD ESP,10 # RET                                                      MSVCR71.dll
7C3473C4    10     # ADD ESP,10 # RET                                                      MSVCR71.dll
7C348B05    EAX    # XCHG EAX,ESP # RET                                                    MSVCR71.dll
7C348D11    0C     # ADD ESP,0C # RET                                                      MSVCR71.dll
7C348D22    0C     # ADD ESP,0C # RET                                                      MSVCR71.dll
7C348D11    0C     # ADD ESP,0C # RET                                                      MSVCR71.dll
7C348D31    0C     # ADD ESP,0C # RET                                                      MSVCR71.dll
```

התאוריה נראית מסובכת גם ככה אבל אין ברירה וצריך להוסיף עוד מכשול בדרך וזה ה-ASLR...



ASLR Bypass

למרות שהתהליך כן קומפל עם ASLR לא מובטח לו שכל ה-DLL-ים שהוא טוען אליו יהיו כאלו שקומפלו עם ASLR, ולכן במקרה שהתהליך יעשה שימוש ב-DLL שאינו מקומפל עם ASLR אותו ה-DLL יטען תמיד לאותו המקום בזיכרון.

לצורך ניצול חולשה בתהליך שטוען אליו DLL כזה, נשתמש במרחב הכתובות הקבוע שנוצר כתוצאה מהטעינה.

לצורך הניצול נצטרך לפחות DLL אחד שאינו ASLR שנטען לתהליך, במידה וישנו אחד או יותר נצטרך ממנו שלושה דברים:

- שתהיה בו קריאה ל-VirtualProtect
- שיהיו בו מספיק קטעי זיכרון עם פקודות אלמנטריות ומיד אח"כ RET - שיאפשרו לנו לבנות מחסנית מתאימה כדי לקרוא ל-VirtualProtect.
- במידה ש-1 או 2 לא מתקיים, אם יש ב-DLL קריאה ל-LoadLibrary נוכל לטעון DLLs שאינם נוספים שאין להם ASLR פעיל ובכך להגדיל את הסיכויים שלנו. (האמת נוכל להסתדר גם אם נטען כאלה שכן יש להם ASLR פעיל כיוון שכבר יהיה לנו את כתובת הבסיס של ה-DLL כערך חזרה מ-LoadLibrary ונוכל להשתמש ב-Offset שהוא קבוע) .

בניגוד לניצול של חולשות ללא הגנות על המחסנית כאן נדרשת הרבה יצירתיות כיוון שכל חולשה דורשת ROP-Shellcode שונה. בחולשה שניצלתי לאחרונה (עד לפרסום המאמר עדיין לא יצא ה-Advisory ולכן לא יכולתי להציג אותה) מצאתי 3 DLLs שאינם ASLR, כך שהיה לי מבחר לא קטן של פקודות. עם זאת חרגתי מהנוהל המוכר והשתמשתי גם באזורי זיכרון שאין בסופם RET אלא כאלה שבסופן מתבצע Call לאוגר מסוים שבעוד מועד דאגתי שהוא יחזיק את הכתובת הבאה שאליה אני צריך לקפוץ. רעיונות כאלו מתפתחים בזמן הניצול כאשר מתקדמים שלב אחרי שלב ביחד עם הדיבאגר. במאמר זה יוצגו רק העקרונות.

DLL שאינם ASLR – היכן ניתן למצוא מה התדירות?

DLL-ים כאלה אפשר למצוא בדרך-כלל בתוכנות שהתקמפלו על גרסאות קצת ישנות של Visual Studio מגירסת 2003 ומטה (Platform toolset 71) יטענו לתוכנית DLL-ים כגון: MSVCR71.DLL, MSVCP71.DLL , MFC71.DLL וכו' שאינם מקומפלים עם ASLR פעיל.

כמו-כן מחקר שהתבצע בזמן האחרון גילה שדווקא חברות האנטי-וירוס וחומות האש למיניהם מקמפלות את התוכנות שלהם ללא שום הגנה פעילה נגד Buffer Overflows, לעתים גם ללא SafeSEH ומכיוון שהן דואגות לשים Hook כמעט על כל תהליך במערכת כדי לטעון אליו את ה-DLL שלהם, דווקא הן אלו שהופכות את ההגנות על המחסנית להרבה פחות אפקטיביות. בחלק מהמקרים הן אפילו יוצרות מקום קבוע בזיכרון שלא מושפע מה-ASLR בעל הרשאות של כתיבה-הרצה, דבר המאפשר להעתיק לשם את ה-shellcode ולרוץ משם באותה השיטה שהוצגה קודם במאמר.

מחקר נוסף שבוצע על-ידי חברת Secunia על רמת השימוש של התוכנות המוכרות במנגנוני ההגנה מראה שעדיין אנו רחוקים מהיום שבו כל התוכנות יהיו מקומפלות עם כל ההגנות.

אחד הכלים שבודקים האם המודולים הטעונים לתהליך הם ASLR Enabled הוא pvefindaddr:

```
** [+] Gathering executable / loaded module info, please wait...
** [+] Finished task, 53 modules found
Loaded modules - ASLR protection status :
----------------------------------------------------------
 * 0x7c340000 - 0x7c396000 : MSVCR71.dll  (C:\windows\system32\MSVCR71.dll) : NO ASLR
 * 0x73810000 - 0x73842000 : WINMM.dll  - †BaseFixup† (C:\windows\system32\WINMM.dll) : ASLR ENABLED
 * 0x761e0000 - 0x76315000 : urlmon.dll  - †BaseFixup† (C:\windows\system32\urlmon.dll) : ASLR ENABLED
 * 0x7c140000 - 0x7c243000 : MFC71.dll  (C:\windows\system32\MFC71.dll) : NO ASLR
 * 0x6fbd0000 - 0x6fe29000 : AcXtrnal.DLL  - †BaseFixup† (C:\windows\AppPatch\AcXtrnal.DLL) : ASLR ENABLED
 * 0x706d0000 - 0x70757000 : MSVCP80.dll  - †BaseFixup† (C:\windows\WinSxS\x86_microsoft.vc80.crt_1fc8b3b9a1e18e3b_8.
 * 0x75b40000 - 0x75b4c000 : MSASN1.dll  - †BaseFixup† (C:\windows\system32\MSASN1.dll) : ASLR ENABLED
 * 0x764c0000 - 0x76594000 : kernel32.dll  - †BaseFixup† (C:\windows\system32\kernel32.dll) : ASLR ENABLED
 * 0x74630000 - 0x74670000 : UxTheme.dll  - †BaseFixup† (C:\windows\system32\UxTheme.dll) : ASLR ENABLED
 * 0x75280000 - 0x75498000 : AcGenral.DLL  - †BaseFixup† (C:\windows\AppPatch\AcGenral.DLL) : ASLR ENABLED
 * 0x74320000 - 0x74333000 : dwmapi.dll  - †BaseFixup† (C:\windows\system32\dwmapi.dll) : ASLR ENABLED
 * 0x77980000 - 0x77abc000 : ntdll.dll  - †BaseFixup† (C:\windows\SYSTEM32\ntdll.dll) : ASLR ENABLED
 * 0x75e80000 - 0x75e99000 : sechost.dll  - †BaseFixup† (C:\windows\SYSTEM32\sechost.dll) : ASLR ENABLED
 * 0x750f0000 - 0x75107000 : USERENV.dll  - †BaseFixup† (C:\windows\system32\USERENV.dll) : ASLR ENABLED
 * 0x74d10000 - 0x74d31000 : ntmarta.dll  - †BaseFixup† (C:\windows\system32\ntmarta.dll) : ASLR ENABLED
 * 0x542f0000 - 0x542fd000 : SortServer2003Compat.dll  - †BaseFixup† (C:\windows\system32\SortServer2003Compat.dll) :
 * 0x76830000 - 0x76a29000 : iertutil.dll  - †BaseFixup† (C:\windows\system32\iertutil.dll) : ASLR ENABLED
 * 0x76320000 - 0x763bd000 : USP10.dll  - †BaseFixup† (C:\windows\system32\USP10.dll) : ASLR ENABLED
 * 0x759e0000 - 0x759fa000 : SspiCli.dll  - †BaseFixup† (C:\windows\system32\SspiCli.dll) : ASLR ENABLED
 * 0x75b50000 - 0x75b62000 : DEVOBJ.dll  - †BaseFixup† (C:\windows\system32\DEVOBJ.dll) : ASLR ENABLED
 * 0x77820000 - 0x7797c000 : ole32.dll  - †BaseFixup† (C:\windows\system32\ole32.dll) : ASLR ENABLED
 * 0x76180000 - 0x7619f000 : IMM32.DLL  - †BaseFixup† (C:\windows\system32\IMM32.DLL) : ASLR ENABLED
 * 0x765a0000 - 0x76669000 : USER32.dll  - †BaseFixup† (C:\windows\system32\USER32.dll) : ASLR ENABLED
 * 0x70cf0000 - 0x70d02000 : MPR.dll  - †BaseFixup† (C:\windows\system32\MPR.dll) : ASLR ENABLED
 * 0x20c70000 - 0x20d0c000 : ISWSHEX.dll  (C:\Program Files\CheckPoint\ZAForceField\Plugins\ISWSHEX.dll) : NO ASLR
 * 0x77ac0000 - 0x77aca000 : LPK.dll  - †BaseFixup† (C:\windows\system32\LPK.dll) : ASLR ENABLED
```

Proof Of Concept

כל החומר התיאורטי שעברנו עליו במאמר דורש המחשה ולשם כך צירפתי למאמר קוד שמריץ -ROP בעזרת שני ה-DLL-ים הטעונים אליו שאינם מוגנים באמצעות ASLR כאשר הקוד עצמו כן קומפל עם DEP ו-ASLR פעילים ב-Visual Studio 2010 על Windows 7.

הדרך ללמוד מה-POC היא להריץ דיבאגר ולנסות להבין את התהליך שקורה שם עד שמגיעים להרצת קוד במחסנית.

ה-Breakpoint הראשון שקופץ הוא לאחר הדריסה של ה-RET, כלומר כשה-ROP מתחיל לרוץ והשני קופץ בתוך המחסנית לאחר שה-ROP הוסיף בהצלחה את הרשאות ההרצה לאזור במחסנית שבו נמצא ה-Breakpoint השני וכמובן ביצע קפיצה לשם.

נסתכל על ה-Dump מ-Windbg:

```
0:000> g
(17c4.968): Break instruction exception - code 80000003 (first chance)
eax=001df2a0 ebx=00000000 ecx=00000000 edx=61616161 esi=00000001
edi=00a7337c
eip=7c34d266 esp=001df2a4 ebp=001df75c iopl=0         nv up ei pl nz ac
pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000216

MSVCR71!raise+0x63:
7c34d266 cc              int3
```

ה-Breakpoint הראשון נמצא בתוך MSVCRT71.DLL המקומפל ללא ASLR, זהו תחילת ה-ROP shellcode אם נעשה Step נגיע לפקודת RET שתשלוף מהמחסנית את הכתובת הבאה לקפוץ אליה, את זה אני משאיר לכם , בכל אופן אם נמשיך בהרצת הקוד נקבל את ה-Breakpoint הבא:

```
0:000> g
(17c4.968): Break instruction exception - code 80000003 (first chance)
eax=00000001 ebx=2afa8166 ecx=001df2e3 edx=772564f4 esi=f775ae01
edi=cac16617
eip=001df768 esp=001df768 ebp=ccd5ff56 iopl=0         nv up ei pl nz na
po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000202
001df768 cc              int3
```

ה-Breapoint קפץ בתוך המחסנית, כלומר הכל עבר חלק והמרחב כתובות שבו נמצא ה-Breakpoint בתוך המחסנית קיבל הרשאות ריצה במידה והיינו מנסים לקפוץ ל-Breakpoint הזה מבלי לשנות את ההרשאות היינו מקבלים Access Violation.

ניתן לראות את השינויים שבוצעו לפני ואחרי הרצה של הקוד:

Stack Access בתחילת ריצה:

```
0:000> !vprot  ebp
BaseAddress:       0023f000
AllocationBase:    00140000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000004  PAGE_READWRITE
Type:              00020000  MEM_PRIVATE
```

Breakpoint ראשון:

```
0:000> g
(12d0.14d0): Break instruction exception - code 80000003 (first chance)
eax=002df708 ebx=00000000 ecx=00000000 edx=61616161 esi=00000001
edi=0138337c
eip=7c34d266 esp=002df70c ebp=002dfbc4 iopl=0         nv up ei pl nz ac
pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000216
MSVCR71!raise+0x63:
7c34d266 cc              int3
```

Breakpoint שני:

```
0:000> g
(175c.160c): Break instruction exception - code 80000003 (first chance)
eax=00000001 ebx=2afa8166 ecx=0023f3db edx=775664f4 esi=f775ae01
edi=cac16617
eip=0023f860 esp=0023f860 ebp=ccd5ff56 iopl=0         nv up ei pl nz na
po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00000202
0023f860 cc              int3
```

ולאחר הריצה:

```
Stack Access:
0:000> !vprot  0023f000
BaseAddress:       0023f000
AllocationBase:    00140000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000040  PAGE_EXECUTE_READWRITE      <----
Type:              00020000  MEM_PRIVATE
```

לסיכום

במאמר זה הצגנו שיטות לעקיפה של שני מתוך ארבע הגנות הפעילות באופן דיפולטי במערכת Windows 7, יש עוד שיטה המוכרת בשם JIT Spray שגם היא מאפשרת עקיפה של ההגנות האלו באמצעות Just-In-Time compiler של VM-ים כגון Flash, Java וכו'. כמובן שהיא שימושית רק בתוכנות המשתמשות ב-VM-ים כאלה, המקרה הקלאסי ביותר הוא הדפדפן.

לעומתה השיטה היותר וותיקה שאותה הצגנו במאמר נכונה לגבי כל תוכנית המכילה לפחות DLL אחד שאין לו הגנת ASLR פעילה. במקרים שלא נמצאים כאלה זהו אתגר לא פשוט אבל לא בהכרח בלתי אפשרי.

אני מקווה שנהניתם מהמאמר - אשמח לקבל תגובות, הערות והארות.

חנוכה שמח!

אביב ברזילי.

ביבליוגרפיה וקריאה נוספת

DEP In Depth:

www.insomniasec.com/publications/DEPinDepth.ppt

- Bypassing Windows Hardware-enforced Data Execution Preventio:

  http://www.uninformed.org/?v=2&a=4&t=txt

- Defeating Solar Designer's Non-executable Stack Patch:

  http://www.insecure.org/sploits/non-executable.stack.problems.html

- Bypassing Stack-Cookies SafeSEH HW-DEP & ASLR:

  http://www.corelan.be:8800/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/?comments=true

- Alexander Sotirov & Mark Dowd - Bypassing Browser Memory Protections

  http://taossa.com/archive/bh08sotirovdowd.pdf