

ומה עם רוצים לדד?

- אינסטרומנטציה – היכולת לשנות את האפליקציה לטובת הבנה טובה יותר שלה
 - Print-ים

• למה?

- הבנת מה רץ? (tracing) ומה לא? (coverage)
- כמה פעמים כל קטע קוד רץ?
- איזה קוד קרא לאיזה קוד? (call graph)
- איזה branch-ים נלקחים? וספקולציה
- כמה זמן לוקח לכל קטע קוד לרוץ? (profiling)
- התמודדות עם שגיאות
- זיהוי זליגות זיכרון וrace conditions



אינסטרואנציה – חלופות

סטטית – שינוי הקובץ לפני הרצתו

- פשוט וקל

דינאמית – שינוי הקובץ בזמן ההרצה

- אין צורך בקומפילציה מחדש
- אפשרות להתמודד עם קוד שמיוצר בזמן אמת (JIT) או משתנה בזמן
- אפשרות להתחבר לתהליכים רצים במערכת

שינוי קוד המקור

- פשוט וקל

שינוי הקוד המקומפל

- אינו תלוי בשפת המימוש
- אפשר לראות מה באמת קורה
- לא תמיד קוד המקור קיים



Detours

- Is a library for instrumenting and intercepting function calls in Win32 binaries.
- Replaces the first instructions of a *target* function with jmp to a *detour* function.
- Preserves original function semantics through a *trampoline* function.
- Enables interception and instrumentation of Win32 binary programs.



פרק 3

הוק'נס



מה זה הוק'נד?

"In computer programming, the term **hooking** covers a range of techniques used to alter or augment the behavior of an operating system, of applications, or of other software components by intercepting function calls or messages or events passed between software components.

Code that handles such intercepted function calls, events or messages is called a "**hook**".

(Wikipedia)



שיאושי הוק'נא

• ניטור תוכנות

- מעקב אחרי פונקציות מסוימות (קלט/פלט וזמן ריצה)
- משמש מפתחים לצורך בדיקת פונקציות מסוימות
 - לדוגמא ניטור פונקציות הקצאה ושחרור זיכרון
 - בדומה לכלי valgrind
- משמש לאיתור התנהגות לא תקינה של תוכנות מסוימות ע"י תוכנת אנטי וירוס
- כלי עזר ל-Reverse Engineering
- קוד עיון שמצורף לתכנית
 - למשל למעקב אחרי המשתמש



שימושי הוק'נא

- שינוי תוכנה
 - שינוי קטע קוד/פונקציה מסוימת
 - שימוש לגיטימי – הרחבת האפשרויות בתוכנה מסוימת
 - למשל Babylon מוסיפה אפשרות תרגום לכל התוכנות הרצות
 - וירוסים מסוימים ישנו את התנהגות המערכת על מנת להסתתר מהמשתמש
- יש עוד מספר שימושים



שיאוי הוק'נ'ל

• עבורנו הוקינג חשוב משתי סיבות

- האחת, נרצה להשתמש בהוקינג בזמן ניתוח קוד דינמי
 - להצגת ערכי ביניים וניטור
- השנייה, לזהות הוקינג בקוד שאנחנו בוחנים
 - למשל כאשר נוזקות ביצעו הוקינג לצרכיהן
 - וכמובן גם לבטל הוקינג כזה



טכניקות הוק'נד

• ניתן לחלק לשתי שיטות

- שינוי תוכנה בזמן ריצה – מתייחס לשינוי התנהגות התוכנה כאשר היא רצה. כלומר שינוי חלק מסוים בזיכרון
- שינוי הקובץ עצמו – שינוי הקוד בקובץ עצמו טרם הריצה
- לעיתים נשלב את שתיהן
 - השינוי בקובץ יפעיל שינוי נוסף בזמן ריצה

• דרישות

- לצורך שינוי קובץ נצטרך
 - פונקציות בסיסיות לקריאה/כתיבה לקובץ
- לצורך שינוי בזמן ריצה נצטרך
 - פונקציות לקריאה וכתיבה מתהליך אחר

• נתחיל מהוקינג ע"י שינוי תוכנה בזמן ריצה

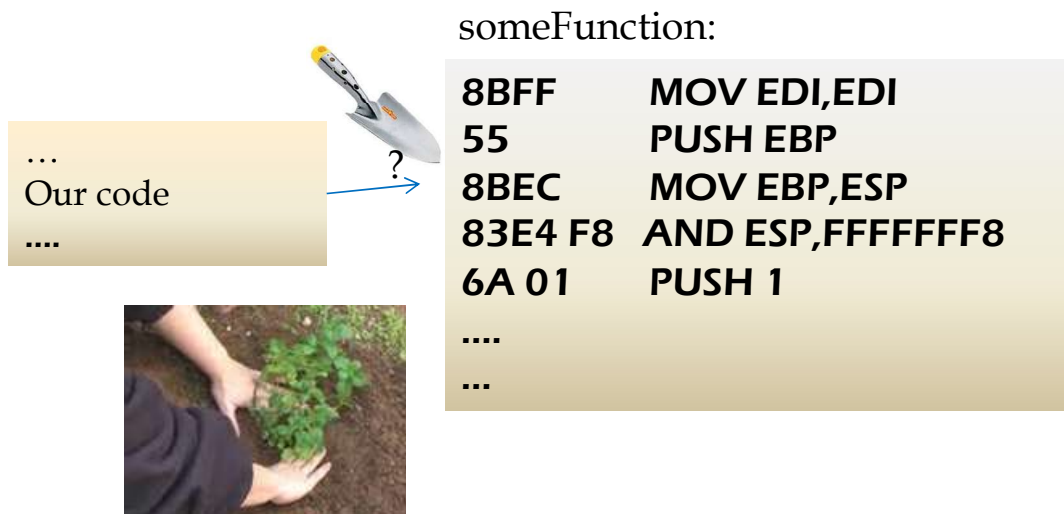
- כשהיא כבר טעונה לזיכרון
- ונניח לרגע שיש ביכולתנו לעשות זאת, בלי לפרט איך



הלכקת קוד

- נניח שיש לנו קטע קוד (פונקציה), באילו דרכים ניתן להשתיל בה קוד?

...
Our code
....



someFunction:

```
8BFF    MOV EDI,EDI
55      PUSH EBP
8BEC    MOV EBP,ESP
83E4 F8  AND ESP,FFFFFFF8
6A 01   PUSH 1
....
...
```

הלרקת קוד

- לצורך הזרקת קוד ניתן להשתמש ב-jmp בתחילת הפונקציה
 - הפקודה לוקחת 5 בתים (קפיצה למקום מרוחק)

someFunction:

8BFF	MOV EDI,EDI
55	PUSH EBP
8BEC	MOV EBP,ESP
83E4 F8	AND ESP,FFFFFFFF8
6A 01	PUSH 1
....	
...	



E9xxxxxxxx	JMP OUR_CODE
83E4 F8	AND ESP,FFFFFFFF8
6A 01	PUSH 1
....	
...	

הלרקה קוד

- לצורך הזרקה נשתמש ב-jmp בתחילת הפונקציה
 - שתקפוץ לקוד שלנו במקום פנוי אחר
 - הפקודה jmp לוקחת 5 בתים
- אם הפקודה לא מתאימה לקוד מבחינת הגודל, נשתמש ב-nop לריפוד (לא חובה)

someFunction2:

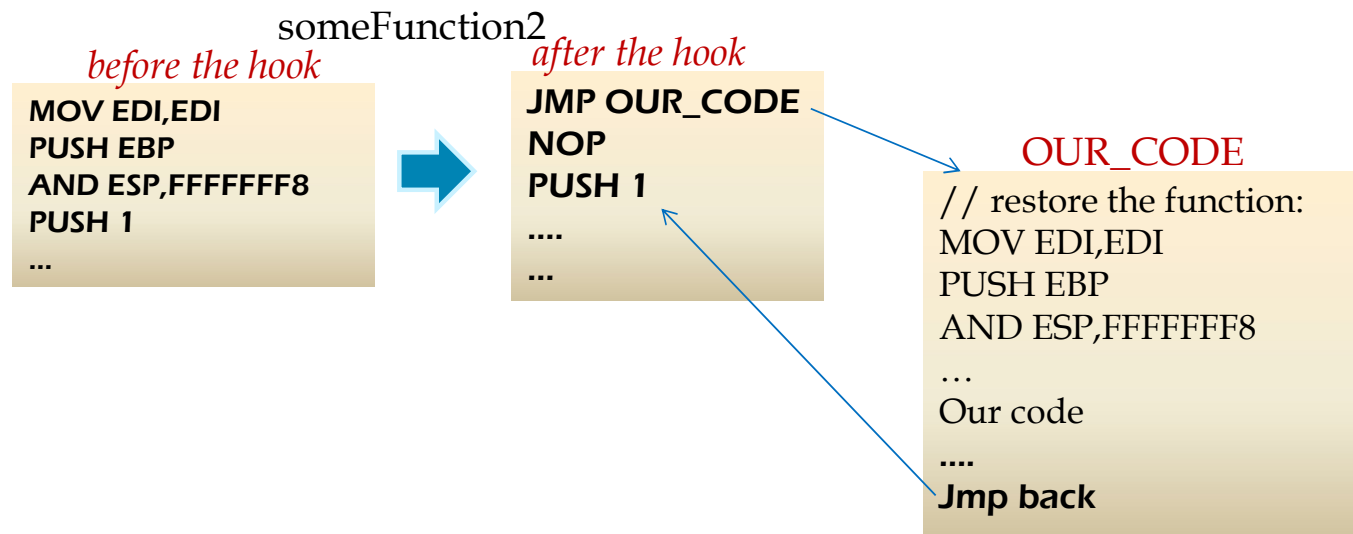
8BFF	MOV EDI,EDI
55	PUSH EBP
83E4 F8	AND ESP,FFFFFFF8
6A 01	PUSH 1
....	
...	



E9xxxxxxxx	JMP OUR_CODE
90	NOP
6A 01	PUSH 1
....	
...	

הלכקת קוד

- יש צורך לזכור את הקוד שנמחק ע"י ה-jmp
 - כדי להחזירם למקומם אם נרצה אי פעם
 - וכדי להריץ את הפקודות שהיו שם, לפני הרצת הקוד שלנו
- וכמובן לקפוץ בחזרה לפקודת המכונה הבאה בסוף הקוד שלנו



הלכקת קוד

• jmp קופצת יחסית לכתובת הפקודה הבאה

- חוסך טיפול ב-relocation בזמן טעינת התכנית...
- לכן XXXXXXXX יהיה

$xxxxxxx = \text{OUR_CODE} - (\text{someFunction2} + 5);$

someFunction2:

E9xxxxxxx	JMP OUR_CODE
90	NOP
6A 01	PUSH 1
...	
...	

OUR_CODE

```
// restore the function:
8BFF    MOV EDI,EDI
55      PUSH EBP
83E4 F8  AND ESP,FFFFFFF8
...
Our code
....
Jmp back
```

שימו לב:

הדרך שהוצגה כאן (jmp) נחשבת אומנם
נפוצה, אך יש עוד המון דרכים לבצע זאת.



הלרקה פאלרת HOT Patching

- Visual Studio תומך בהידור עם הדגל hotpatching

- חלק גדול מה-DLL-ים קומפלו בשיטה זו
- במצב זה הפונקציות "מוכנות" ל-hooking

- Hot patching במצב

- כל פונקציה מתחילה ב-mov edi, edi
 - בעצם nop שניתן לדרוס
 - גודל של jmp short
- לפניהם יהיו בדיוק חמישה nop's
 - גודל של jmp
 - לא מבוצעים בריצה רגילה של הפונקציה



הלרקה עצלרת HOT Patching

- לצורך הזרקת קוד נחליף את שבעת הבתים הללו בקפיצות

Original Code	Patched Code
nop nop nop nop nop	jmp xxxxxxxx to our code
→ mov edi, edi	jmp short 7 bytes backward to the jmp above
Rest of function	Unchanged

- כך בקריאה לפונקציה, נבצע את שני ה-jmp

- שיפנו אותנו לקוד שלנו

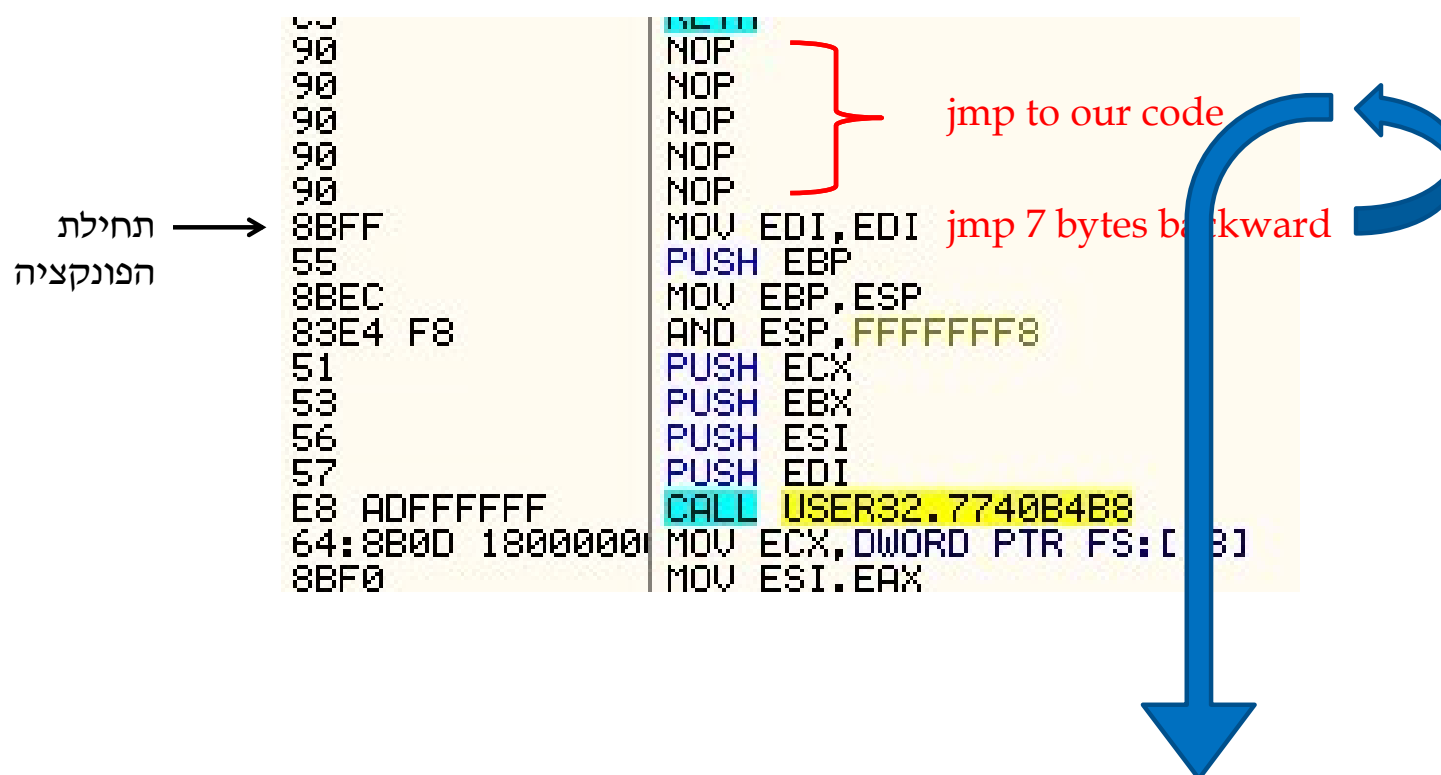
- לא דרסנו פקודות חשובות – אין צורך לשחזר

- למה שני jmp כשאפשר היה אחד?

- כי בריצה רגילה, ללא הוקינג, ביצוע mov edi, edi יעיל יותר מלהפעיל 5 nop-ים בתחילת כל פונקציה



הלרקה עצלרת HOT Patching



הלכה בעצרת שנינו ה-IAT

- בזמן ריצה, ה-IAT מצביע לפונקציות ב-DLL-ים השונים
- כל קריאה לפונקציה של DLL עוברת דרך ה-IAT
- שינוי ה-IAT, כך שיצביע לקוד שלנו במקום לפונקציה ב-DLL, יפנה את הקריאה לפונקציה לקוד שלנו
 - הקוד שלנו יקרא לקוד המקורי, אם יש צורך בכך
 - כאשר הקוד שלנו נטען כ-DLL, אין צורך לזכור את הכתובת המקורית
 - נקרא לפונקציה ישירות דרך ה-IAT של ה-DLL שלנו
 - כך שהשינוי בזמן ריצה יהיה רק שינוי המצביע ב-IAT
- יתרון נוסף: ה-IAT נשמר באיזור זיכרון שאינו מוגן מכתובה



שינוי קובץ ההרצה

- בשקפים הקודמים עסקנו בשינוי הקוד בזמן ריצה
 - כלומר שינוי מקומי בכל פונקציה ופונקציה
 - מאפשר גם לשנות פונקציות מ-DLL-ים
- לעיתים נעדיף לשנות את הקוד ישירות בקובץ ההרצה
 - אפשר לבצע הוקינג באופן דומה למבנה שהצגנו לזמן ריצה
 - אבל לא לשנות קוד בספריות נטענות דינמית
 - טכניקות מורכבות יותר מאפשרות לבצע שינויים גדולים בקוד הנמצא בקובץ ההרצה
 - למשל post-link optimizations



תמיכה בתוכנה

• קיימות תוכנות המאפשרות שינוי ישיר בקובץ ההרצה

- Immunity
- מאפשר לשנות קוד בזמן ניתוח התוכנה
- ואז לשמור את התוצאה לקובץ הרצה חדש

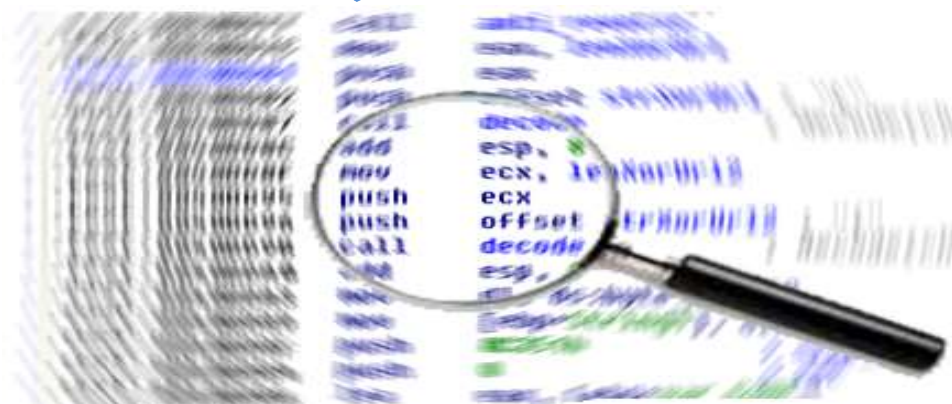


סיכום

- דנו איך לשנות קוד על מנת להכניס פונקציה משלנו
 - כשקוראים לפונקציה אחרת, למשל פונקציית מערכת
 - לא הצגנו כיצד לבצע זאת באופן מעשי, נדגים בהמשך
- יש מגוון דרכים שבהם ניתן לעשות זאת
 - הצגנו דוגמאות פשוטות
 - ישנם אנטי-וירוסים שמזהים שינויי הוקינג
 - לכן וירוסים פיתחו שיטות "יצירתיות", ומנגנונים שטרם נבדקים ע"י אנטי-וירוסים
- הציעו דרך לבצע הוקינג שלא ניתנת לזיהוי פשוט
- הציעו דרך לבצע הוקינג בזמן ריצה
 - בלי לשנות בכלל את קובץ ההרצה



תלכורת ודואא פשיאוס בהאס



מה זה API?

Application Programming Interface

- מתייחס לקבוצת פונקציות מוכנות שמסופקות למתכנת באופן מובנה
 - בדרך כלל ע"י מערכת ההפעלה
 - בחלונות זה נקרא WinApi
 - גם שירותי ענן מספקים API לאפליקציות, למשל פייסבוק
- לדוגמא, פונקציות לטיפול בקבצים
 - מיוצאות ע"י kernel32.dll
 - שבגרסאות חדשות של חלונות מפנה ל-kernelbase.dll
 - CreateFile, ReadFile, WriteFile
 - fopen, fread, fwrite הן עטיפה לפונקציות הגישה לקבצים, ברמת המהדר, המשתמשת בפונקציות לעיל



הודעות בחלונות

- תוכנות בחלונות מקבלות קלט מהמשתמש דרך הודעות מהמערכת
- כאשר המשתמש מבצע פעולה מסוימת בתוכנה
 - הפעולה מתורגמת ע"י מערכת ההפעלה להודעה
 - וזו נשלחת לתוכנה המתאימה
- בכל תוכנה בחלונות יש לולאה שמטפלת בהודעות
- ביצוע הטיפול בהודעה נעשה ע"י פונקציית `callback`
 - שנקראת (בעקיפין) מהלולאה
 - ומוגדרת במבנה נתונים `WNDCLASS`

הודעות =

אירועים של מ"ה שמדווחים לתהליך

ועוד

אירועי עכבר

הקלדות במקלדת

הנדסה לאחור – חורף תשפ"א

© פרופ' אלי ביהם, אביעד כרמל, עמר קדמיאל

11.05.2023



הודעות בחלונות

```
WNDCLASS wc;
...
wc.lpfnWndProc = (WNDPROC) WndProc;
...
RegisterClass(&wc)
...
hwndMain = CreateWindow("MainWndClass", "Sample",
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, CW_USEDEFAULT, (HWND) NULL,
    (HMENU) NULL, hinst, (LPVOID) NULL); ;
...
while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

מטפלת בהודעות מהחלון

LRESULT CALLBACK WndProc(// מטפלת בהודעות מהחלון
HWND hwnd, // handle to window
UINT uMsg, // message identifier
WPARAM wParam, // first message parameter
LPARAM lParam) // second message parameter

Message Loop



דואמאות פסוקי הודעות בחלונות

למשל הודעות על שימוש בעכבר

...
WM_MOUSEFIRST = 0x200
WM_MOUSEMOVE = 0x200
WM_LBUTTONDOWN = 0x201
WM_LBUTTONUP = 0x202
WM_LBUTTONDBLCLK = 0x203
WM_RBUTTONDOWN = 0x204
WM_RBUTTONUP = 0x205
WM_RBUTTONDBLCLK = 0x206
WM_MBUTTONDOWN = 0x207
WM_MBUTTONUP = 0x208
WM_MBUTTONDBLCLK = 0x209
WM_MOUSEWHEEL = 0x20A
WM_MOUSEHWHEEL = 0x20E

...



פונקציית SetWindowsHookEx

קבלת הודעות על תהליכים אחרים

- חלונות מאפשרת לבצע Hook בתהליכים אחרים
 - באמצעות הפונקציה SetWindowsHookEx
- הפונקציה מאפשרת לנטר הודעות בתוכנית
 - כך שההודעה תגיע לקוד שלך בנוסף לקוד הרגיל של התוכנית
 - תומך במגוון סוגי הודעות, חלקן אפשר לפני ההעברה לתהליך או אחרי
- ניתן גם להזריק DLL באמצעות הפונקציה
 - על הזרקות נדון בהרחבה בהמשך
- הפונקציה יכולה לבצע Hook בתוכנית מסוימת או בכל התוכניות
- הפונקציה היא מעטפת לקוד שכותב לזיכרון של תהליך אחר
- כל המידע נמצא ב-MSDN



Key Logger – *למה?*

```
int CALLBACK WinMain(HINSTANCE hInstance,  
                     HINSTANCE hPrevInstance,  
                     LPSTR lpCmdLine, int nCmdShow) {  
    hHook=SetWindowsHookEx(WH_KEYBOARD_LL,  
                           hookProc, hInstance, 0);  
    // error checking...  
    MessageBox(NULL, L"Press OK to exit", L"Note", MB_OK);  
    UnhookWindowsHookEx(hHook);  
    return 0;  
}
```

- הערה : MessageBox מנהלת לולאת טיפול בהודעות פנימית משלה, כך שאין צורך לכתוב אחת כזו במפורש.

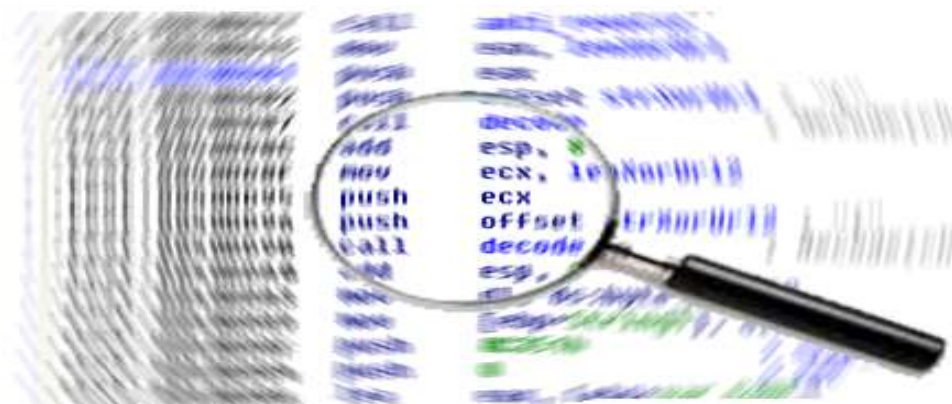


Key Logger – *לכידת מקשים*

```
LRESULT CALLBACK hookProc(int nCode, WPARAM wParam, LPARAM lParam) {
    PKBDLLHOOKSTRUCT ptr = (PKBDLLHOOKSTRUCT) lParam;
    if (wParam == WM_KEYDOWN) {
        switch (ptr->vkCode) {
            case VK_TAB:
                //code...
            case VK_SHIFT:
                //code...
            case VK_CONTROL:
                //code...
            default:
                //code... (ptr->vkCode is the key)
        }
    }
    // Chain to other hooks from other applications
    return CallNextHookEx(0,nCode,wParam,lParam);
}
```



הוק'נ'ג דואנא מפורטת



דואא פהוק'נא

- מתכנת כתב קוד שמבצע מספר קריאות ל-MessageBox:

Program.c:

```
...  
MessageBox(NULL, L"First MsgBox\n",NULL, NULL);  
...  
MessageBox(NULL, L"Second MsgBox\n",NULL, NULL);  
...
```

- נניח שרוצים שהודעות ה-MessageBox יהיו מוצפנות
- ברשותנו הקובץ הבינארי בלבד
 - בלי אפשרות להשיג את קוד מקור
- איך נפתור את הבעיה?



סיקור יישום

- תכנית הדוגמא משלבת את הקבצים הבאים
 - קוד התכנית נמצא נניח ב-program.exe
 - פונקציית MessageBox היא ב-user32.dll
 - והם כמובן גם קוראים בתורם לשירותי kernel32.dll
- האם לשנות את הקוד בקובץ
 - ?program.exe
 - ?user32.dll
 - ?kernel32.dll



אל היכן נשנה?

- kernel32.dll?

- נשמע לא רלוונטי

- user32.dll?

- נשמע המקום הנכון

- אבל ישנה גם בתכניות אחרות!

- המקום הנכון כשרוצים לעקוב אחרי כל התהליכים

- program.exe?

- מסובך. ואין לנו קוד מקור

- אבל זה הקובץ היחיד שניתן לשנות ללא הרשאות מיוחדות

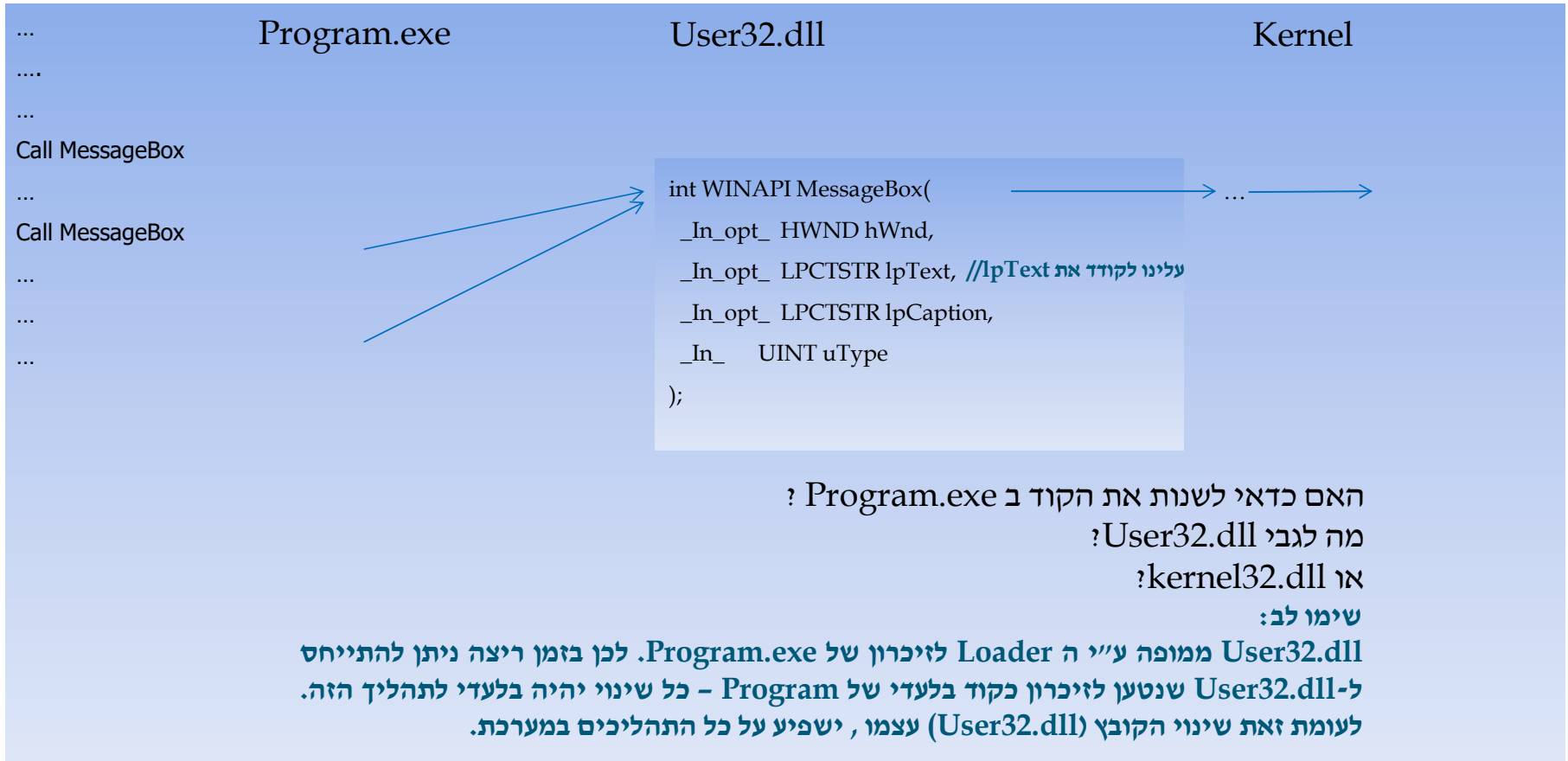
- ודרכו נוכל לשנות את השאר בזמן ריצה!

- ללא צורך בשינוי קבצי ה-DLL

- כלומר ללא השפעה על שאר התהליכים במחשב



אל היכן נשנה?



ספריות דינאיות - DLL

- ספריות סטטיות משולבות בזמן לינק לתוך קובץ ההרצה
 - הקוד שקורא להן יודע את כתובתן היחסית כפי שהוא יודע לכל פונקציה שכתובה ישירות בקוד
- ספריות דינמיות נטענות ע"י ה-loader
 - ורק אז מקושרות הפונקציות שבספריה למקומות בהן קוראים להן
 - ה-loader עשוי לטעון אותן לכתובת אקראית
 - ע"י ASLR
- בלינוקס הן עם סיומת SO
 - למשל
 - /usr/lib64/libc.so
 - /usr/lib64/libcrypto.so
 - /usr/lib64/libX11.so



איך טענות ספריות דינמיות?

"copy-on-write"

- כדי לחסוך בזיכרון סביר שטבלאות הדפים של כל התהליכים מפנות ספריה דינמית לאותו שטח בזיכרון הפיסי
 - כך שנשמר רק עותק אחד של הספריה
- ומה קורה כשתהליך רוצה לכתוב בדף כזה?
 - ???



אל היכן נשנה?

• אפשרות נוספת

- מתהליך אחר שאנחנו מתכנתים בעצמנו
 - שישנה את הפעולה (או תוכן הזיכרון) של התהליך שרצינו
 - תוך שימוש בשירותי מערכת הפעלה שתומכים בכך
- יתרונות
 - לא צריך לשנות שום קובץ!
 - כן צריך הרשאות מתאימות
 - ואי אפשר לזהות דבר מניתוח קבצי התכנית וה-DLL-ים
 - כי השינוי לא נעשה דרכם



האקרה הפשוט: יש קוד *fe* התוכנית

- תחילה נדגים כיצד לשנות את MessageBox כאשר יש לנו גישה לקוד מקור (של Program)
- התהליך מתמקד בשינוי הפונקציה ב-user32.dll



שתי כרסאות fe MessageBox

7513CDD2	.text	Export	MessageBeep
75185F69	.text	Export	MessageBoxA
75185FAF	.text	Export	MessageBoxExA
75185FD3	.text	Export	MessageBoxExW
7518618F	.text	Export	MessageBoxIndirectA
7514148C	.text	Export	MessageBoxIndirectW
7518607C	.text	Export	MessageBoxTimeoutA
75185FF7	.text	Export	MessageBoxTimeoutW
75185F8C	.text	Export	MessageBoxW
75196578	.idata	Import	GDI32.MirrorRgn
7518E4E1	.text	Export	ModifyMenuA
751682EC	.text	Export	ModifyMenuW
75139546	.text	Export	MonitorFromPoint
75136EE4	.text	Export	MonitorFromRect
75136C5D	.text	Export	MonitorFromWindow
75186ED9	.text	Export	mouse_event
7512AD9F	.text	Export	MoveWindow

- נחפש את הפונקציה
ב-Export table של user32.dll :

• מה ההבדל בין MessageBoxA ל-MessageBoxW?

- הראשון נועד לתווי ASCII והשני ל-Unicode
- MessageBoxA ממיר את הקלט ל-Unicode, ואז קורא בתורו ל-MessageBoxW
- כלומר MessageBoxW ייקרא בסוף בכל מקרה
 - מומלץ לעבור על הקוד של הפונקציה ולאתר את ההמרה

הוק'נא fe MessageBoxW

- הפונקציה מוכנה להוקינג עם Hot Patching

75185F87	90	NOP
75185F88	90	NOP
75185F89	90	NOP
75185F8A	90	NOP
75185F8B	90	NOP
75185F8C	8BFF	MOV EDI,EDI
75185F8E	55	PUSH EBP
75185F8F	8BEC	MOV EBP,ESP
75185F91	6A FF	PUSH -1
75185F93	6A 00	PUSH 0
75185F95	FF75 14	PUSH DWORD PTR SS:[EBP+14]
75185F98	FF75 10	PUSH DWORD PTR SS:[EBP+10]
75185F9B	FF75 0C	PUSH DWORD PTR SS:[EBP+0C]
75185F9E	FF75 08	PUSH DWORD PTR SS:[EBP+08]
75185FA1	E8 51000000	CALL USER32.MessageBoxTimeoutW
75185FA6	5D	POP EBP
75185FA7	C2 1000	RETN 10
75185FA0	90	NOP



הוק'נ'ע fe MessageBoxW

- את mov edi, edi נשנה לקפיצה 7 בתים אחורנית
 - כלומר ל- EB F9 (הערך 7- מיוצג ע"י F9)
- משם נקפוץ לפונקציה שלנו ע"י שינוי התוכן של חמשת פקודות ה-nop ל-jmp
 - לא נוכל לדעת את כתובת הקפיצה משם מראש

75185F87	90	NOP	90	NOP	75185F87
75185F88	90	NOP	55	PUSH EBP	75185F88
75185F89	90	NOP	8BEC	MOV EBP,ESP	75185F89
75185F8A	90	NOP	6A FF	PUSH -1	75185F8A
75185F8B	90	NOP	6A 00	PUSH 0	75185F8B
75185F8C	8BFF	MOV EDI,EDI	FF75 14	PUSH DWORD PTR SS:[EBP+14]	75185F8C
75185F8E	55	PUSH EBP	FF75 10	PUSH DWORD PTR SS:[EBP+10]	75185F8E
75185F8F	8BEC	MOV EBP,ESP	FF75 0C	PUSH DWORD PTR SS:[EBP+C]	75185F8F
75185F91	6A FF	PUSH -1	FF75 08	PUSH DWORD PTR SS:[EBP+8]	75185F91
75185F93	6A 00	PUSH 0	E8 51000000	CALL USER32.MessageBoxTimeoutW	75185F93
75185F95	FF75 14	PUSH DWORD PTR SS:[EBP+14]	5D	POP EBP	75185F95
75185F98	FF75 10	PUSH DWORD PTR SS:[EBP+10]	C2 1000	RETN 10	75185F98
75185F9B	FF75 0C	PUSH DWORD PTR SS:[EBP+C]	90	NOP	75185F9B
75185F9E	FF75 08	PUSH DWORD PTR SS:[EBP+8]	90	NOP	75185F9E
75185FA1	E8 51000000	CALL USER32.MessageBoxTimeoutW			
75185FA6	5D	POP EBP			
75185FA7	C2 1000	RETN 10			
75185FAD	90	NOP			



פונקציית ה-Hook

```

__declspec(naked) void msgHookW()
{
    __asm {
        mov eax, [esp+8]
        push a
        push 0x40
        push 0x100
        push eax
        call VirtualProtect
        //add esp, 0x10
        mov eax, [esp+8]

loop:
        inc WORD PTR [eax]
        inc eax
        inc eax
        cmp word ptr [eax], 0
        jne loop
        nop
        ...
    }
}

```

כמה דגשים:

- `__declspec(naked)` מגדיר פונקציה ללא תוספות של הקומפיילר (כמו הקצאת frame בתחילת פונקציה)
- הפרמטר `lpText` נמצא בכתובת `esp+8`
- לא ידוע לנו מהן ההרשאות של ה-`Page` בו נמצא `lpText`, לכן נשתמש ב-`VirtualProtect` לשינוי הרשאות
- `VirtualProtect` דואג לשחזור המחסנית בסיומו, לכן אין צורך לעשות `add esp` לניקוי הפרמטרים
- `lpText` מקודדת ב-`Unicode`, כלומר 2 בתים לכל תו
- בסוף הפונקציה יש צורך לחזור ל-`MessageBoxW` (בתוספת שני בתים)
- אך כרגע אנחנו לא יודעים את הכתובת
- וכך גם במקרה הכללי
- לכן נשאיר מקום לפקודת `jmp`
- אם נדייק, בדוגמא הזו כן ניתן לדעת
- רמז: נחשו איך אנחנו יודעים את הכתובת של `VirtualProtect`



שיווי MessageBoxW

void setHook() { חלק ראשון – חיפוש הכתובת

LPVOID f;

HMODULE h;

CHAR JumpOpcode[6] = "\\xE9\\x90\\x90\\x90\\x90"; //jmp (long)

DWORD lpProtect;

LPVOID CalculatedJump, JumpTo;

h = GetModuleHandle(L"user32.dll");

if (h == NULL) {

return;

}

f = GetProcAddress(h, "MessageBoxW");

if (f == NULL)

return;

JumpTo = (char*)&msgHookW - (char*)f;

VirtualProtect((char*)f-5,0x7,PAGE_EXECUTE_READWRITE,&lpProtect);

memcpy(JumpOpcode+1,&JumpTo,0x4);

memcpy((char*)f-5,&JumpOpcode,0x5);

(char)f = 0xEB;

((char)(f)+1) = 0xF9;

VirtualProtect((char*)f-5,0x7,PAGE_EXECUTE_READ,&lpProtect);

VirtualProtect((char*)msgHookW,0x100,PAGE_EXECUTE_READWRITE,&lpProtect);

JumpTo = (char*)f + 2 - ((char*)&msgHookW+0x2e+0x5);

memcpy(JumpOpcode+1,&JumpTo,0x4);

memcpy((char*)&msgHookW+0x2e,&JumpOpcode,0x5);

VirtualProtect((char*)msgHookW,0x100,PAGE_EXECUTE_READ,&lpProtect);

}

הסבר לחלק הראשון:

- שימוש בפונקציות GetModuleHandle ו-GetProcAddress להשגת הכתובת של MessageBoxW בתהליך שלנו
- ניתן לקרוא עליהן ב-MSDN
- הכנת מחרוזת JumpOpcode (קפיצה חזרה)



שניוי MessageBoxW

חלק שני – עדכון הקפיצות

הסבר לחלק השני:

```
void setHook() {
    LPVOID f;
    HMODULE h;
    CHAR JumpOpcode[6] = "\xE9\x90\x90\x90\x90";
    DWORD lpProtect;
    LPVOID CalculatedJump, JumpTo;
    h = GetModuleHandle(L"user32.dll");
    if (h == NULL) {
        return;
    }
    f = GetProcAddress(h, "MessageBoxW");
    if (f == NULL)
        return;
```

```
JumpTo = (char*)&msgHookW - (char*)f;
```

```
VirtualProtect((char*)f-5,0x7,
    PAGE_EXECUTE_READWRITE,
    &lpProtect);
```

```
memcpy(JumpOpcode+1,&JumpTo,0x4);
```

```
memcpy((char*)f-5,&JumpOpcode,0x5);
```

```
*(char*)f = 0xEB; // jmp (short)
```

```
*((char*)(f)+1) = 0xf9; // -7
```

```
VirtualProtect((char*)f-5,0x7,PAGE_EXECUTE_READ,
    &lpProtect);
```

```
VirtualProtect((char*)msgHookW,0x100,PAGE_EXECUTE_READWRITE,&lpProtect);
```

```
JumpTo = (char*)f + 2 - ((char*)&msgHookW+0x2e+0x5);
```

```
memcpy(JumpOpcode+1,&JumpTo,0x4);
```

```
memcpy((char*)&msgHookW+0x2e,&JumpOpcode,0x5);
```

```
VirtualProtect((char*)msgHookW,0x100,PAGE_EXECUTE_READ,&lpProtect);
```

```
}
```

- ראשית מחשבים את המרחק בין

הפונקציה שלנו ל-MessageBoxW

- יש לשים לב שהקפיצה מחושבת

החל מ-MessageBoxW עצמו

מכיוון שהיא נמצאת 5 בתים
מלפניו

- מאפשרים כתיבה לפונקציה ע"י

VirtualProtect

- מעדכנים את jmpOpcode

- משנים את MessageBoxW



MessageboxW עיון

```
void setHook() {
    LPVOID f;
    HMODULE h;
    CHAR JumpOpcode[6] = "\\xE9\\x90\\x90\\x90\\x90";
    DWORD lpProtect;
    LPVOID CalculatedJump, JumpTo;
    h = GetModuleHandle(L"user32.dll");
    if (h == NULL) {
        return;
    }
    f = GetProcAddress(h, "MessageBoxW");
    if (f == NULL)
        return;

```

E9 74B0B8F4	JMP dllmsg.msgHookW
EB F9	JMP SHORT USER32.75185F87
55	PUSH EBP
8BEC	MOV EBP, ESP
6A FF	PUSH -1
6A 00	PUSH 0
FF75 14	PUSH DWORD PTR SS:[EBP+14]
FF75 10	PUSH DWORD PTR SS:[EBP+10]
FF75 0C	PUSH DWORD PTR SS:[EBP+C]
FF75 08	PUSH DWORD PTR SS:[EBP+8]
E8 51000000	CALL USER32.MessageBoxTimeoutW
5D	POP EBP
C2 1000	RETN 10
CC	INT3

```
JumpTo = (char*)&msgHookW - (char*)f;
VirtualProtect((char*)f-5,0x7,
    PAGE_EXECUTE_READWRITE,
    &lpProtect);
memcpy(JumpOpcode+1,&JumpTo,0x4);
memcpy((char*)f-5,&JumpOpcode,0x5);
*(char*)f = 0xEB;
*((char*)(f)+1) = 0xf9;
VirtualProtect((char*)f-5,0x7,PAGE_EXECUTE_READ,
    &lpProtect);

```

```
VirtualProtect((char*)msgHookW,0x100,PAGE_EXECUTE_READWRITE,&lpProtect);
JumpTo = (char*)f + 2 - ((char*)&msgHookW+0x2e+0x5);
memcpy(JumpOpcode+1,&JumpTo,0x4);
memcpy((char*)&msgHookW+0x2e,&JumpOpcode,0x5);
VirtualProtect((char*)msgHookW,0x100,PAGE_EXECUTE_READ,&lpProtect);
}

```



עיון MessageboxW

חלק שלישי – הפחת פונקציית Hook-

הסבר לחלק השלישי:

```
void setHook() {
    LPVOID f;
    HMODULE h;
    CHAR JumpOpcode[6] = "\xE9\x90\x90\x90\x90";
    DWORD lpProtect;
    LPVOID CalculatedJump, JumpTo;
    h = GetModuleHandle(L"user32.dll");
    if (h == NULL) {
        return;
    }
    f = GetProcAddress(h, "MessageBoxW");
    if (f == NULL)
        return;
    JumpTo = (char*)&msgHookW - (char*)f;
    VirtualProtect((char*)f-5,0x7,
        PAGE_EXECUTE_READWRITE,
        &lpProtect);
    memcpy(JumpOpcode+1,&JumpTo,0x4);
    memcpy((char*)f-5,&JumpOpcode,0x5);
    *(char*)f = 0xEB;
    *((char*)(f)+1) = 0xf9;
    VirtualProtect((char*)f-5,0x7,PAGE_EXECUTE_READ,
        &lpProtect);
    VirtualProtect((char*)msgHookW,0x100,
        PAGE_EXECUTE_READWRITE,&lpProtect);
    JumpTo = (char*)f + 2 - ((char*)&msgHookW+0x2e+0x5);
    memcpy(JumpOpcode+1,&JumpTo,0x4);
    memcpy((char*)&msgHookW+0x2e,&JumpOpcode,0x5);
    VirtualProtect((char*)msgHookW,0x100,
        PAGE_EXECUTE_READ,&lpProtect);
}
```

• VirtualProtect לפונקציה שלנו

• חישוב "כמה" לקפוץ:

נרצה לקפוץ ל-`msgboxW+2` כאשר הקפיצה תמוקם בפונקציה שלנו החל מהבית `0x2e` (זה מספר שרירותי, מה שחשוב זה למצוא מקום עם `nop's` אחרי הקוד שלנו)

מכיוון שהמרחק מחושב מהפקודה הבאה – נוסיף 5

```
VirtualProtect((char*)msgHookW,0x100,
    PAGE_EXECUTE_READWRITE,&lpProtect);
JumpTo = (char*)f + 2 - ((char*)&msgHookW+0x2e+0x5);
memcpy(JumpOpcode+1,&JumpTo,0x4);
memcpy((char*)&msgHookW+0x2e,&JumpOpcode,0x5);
VirtualProtect((char*)msgHookW,0x100,
    PAGE_EXECUTE_READ,&lpProtect);
}
```

הנדסה לאחור – חורף תשפ"א

© פרופ' אלי ביהם, אביעד כרמל, עמר קדמיאל

11.05.2023



הפחת פונקציית ה-Hook

חלק feי - הפחת פונקציית Hook-

```
__declspec(naked) void msgHookW()
{
```

```
__asm {
```

```
mov eax, [esp+8]
```

push a

push 0x40

push 0x100

push eax

```
call VirtualProtect
```

```
//add esp , 0x10
```

```
mov eax, [esp+8]
```

loop:

```
inc WORD PTR [eax]
```

```
inc eax
```

```
inc eax
```

```
cmp word ptr [eax], 0
```

```

jne loop

```

nop

...

$$\}$$

}

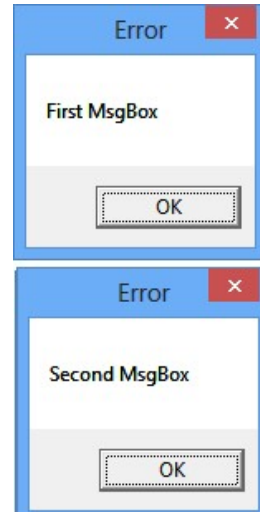
```
MOV EAX,DWORD PTR SS:[ESP+8]
PUSH DWORD PTR DS:[a]
PUSH 40
PUSH 100
PUSH EAX
CALL DWORD PTR DS:[&KERNEL32.VirtualProtect]
MOV EAX,DWORD PTR SS:[ESP+8]
INC WORD PTR DS:[EAX]
INC EAX
INC EAX
CMP WORD PTR DS:[EAX],0
JNZ SHORT dllmsg.69D1101C
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
JMP USER32.75185F8E
```

jmp back to MessageBoxW

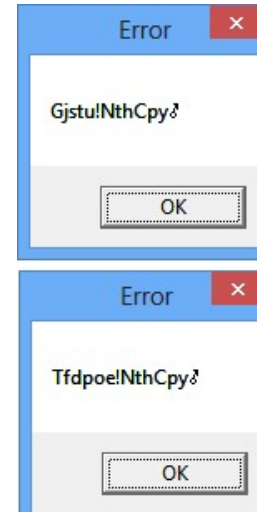
Hook-ה ת'ת ne

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)  
{  
    setHook();  
    MessageBox(NULL, L"First MsgBox\n",NULL, NULL);  
    MessageBox(NULL, L"Second MsgBox\n",NULL, NULL);  
  
    return 0;  
}
```

without Hook:



With Hook:



עתידת ה-Hook

- ללא קוד מקור, נוכל לשתול את הקוד שלנו בקובץ ההרצה ע"י שינויו כך שיכלול
 - את הפונקציות msgHookW ו-setHook
 - למשל ב-section חדש, או בהמשך ל-section קיים
 - ולשנות את תחילת WinMain כך שתקרא ל-setHook
 - ע"י הוקינג
- אבל אפשר גם באופן "פחות פולשני"...



עתידת ה-Hook ב-DLL

- נרצה שה Hook שלנו יהיה גנרי, כך שיהיה ניתן להשתמש במודול שלנו בקלות מכל קובץ
- לכן נכתוב את setHook ב-DLL
- מתכנת שרוצה להשתמש בקוד שלנו – יבצע LoadLibrary

```
INT WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,  
    LPSTR lpCmdLine, int nCmdShow)  
{  
    LoadLibrary("setHook.dll");  
    MessageBox(NULL, L"First MsgBox\n",NULL, NULL);  
    MessageBox(NULL, L"Second MsgBox\n",NULL, NULL);  
  
    return 0;  
}
```



עתידת ה-Hook ב-DLL

```
__declspec(naked) void msgHookW()  
{  
    ...  
    ...  
}
```

```
void sethook() {  
    ...  
    ...  
}
```

• ניתן לקרוא ב-MSDN על DLLs

• חשוב להכיר – ברגע טעינת DLL המערכת קוראת ל-DllMain

• DLL הוא בפורמט PE

• בקוד של המתכנת במקום ל-hook() קוראים ל-LoadLibrary

```
BOOL APIENTRY DllMain( HMODULE hModule,  
    DWORD ul_reason_for_call,  
    LPVOID lpReserved  
    )  
{  
    switch (ul_reason_for_call)  
    {  
        case DLL_PROCESS_ATTACH:  
            sethook();  
            break;  
        case DLL_THREAD_ATTACH:  
        case DLL_THREAD_DETACH:  
        case DLL_PROCESS_DETACH:  
            break;  
    }  
    return TRUE;  
}
```



הלרקת DLL

- נרצה לבצע את ה-Hook גם כאשר אין ברשותנו את קוד המקור
 - לכן נחפש דרך לטעון את ה-DLL לתוכנית בצורה אחרת
- זה נקרא הזרקת DLL
 - DLL Injection
- ישנן מספר שיטות לבצע זאת
 - למשל יש משתנה סביבה שמאפשר לטעון DLL בהרצת EXE
 - אנו נתמקד בשיטה הנפוצה המתבססת על WriteProcessMemory
 - CreateRemoteThread
- בשקפים הבאים נראה מימוש דוגמא ל-DLL Injector

עם



הלדקת DLL

```
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,  
                  LPSTR lpCmdLine,int nCmdShow)  
{  
    PROCESS_INFORMATION pi;  
    STARTUPINFOA Startup;  
    ZeroMemory(&Startup, sizeof(Startup));  
    ZeroMemory(&pi, sizeof(pi));  
    CreateProcessA("msgbox.exe", NULL, NULL, NULL, NULL,  
                  CREATE_SUSPENDED, NULL, NULL, &Startup, &pi);  
  
    if(!(dllInjector("c:\\dllmsg.dll", pi.dwProcessId)))  
        return 1;  
    Sleep(1);  
    ResumeThread(pi.hThread);  
    return 0;  
}
```

מידע נוסף?

.MSDN



הלדקת DLL

```
BOOL dllInjector(char * dllpath , DWORD pID)
{
    HANDLE pHandle , threadHandle;
    LPVOID remoteString;
    LPVOID remoteLoadLib;
    pHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pID);
    if(!pHandle)
        return false;
    remoteLoadLib = (LPVOID)GetProcAddress(GetModuleHandle(L"kernel32.dll"),
                                           "LoadLibraryA");

    // error checking
    remoteString = (LPVOID)VirtualAllocEx(pHandle, NULL, strlen(dllpath)+1,
                                           MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    // error checking
    WriteProcessMemory(pHandle, remoteString, dllpath, strlen(dllpath)+1, NULL);
    threadHandle = CreateRemoteThread(pHandle, NULL, NULL,
                                      (LPTHREAD_START_ROUTINE)remoteLoadLib, remoteString, NULL, 0);
    // error checking
    return true;
}
```



הלדקת DLL

חשוב להקיס את התהליך עם ההרשאות הנכונות.

```
BOOL dllInjector(char * dllpath , DWORD pID)
{
    HANDLE pHandle , threadHandle;
    LPVOID remoteString;
    LPVOID remoteLoadLib;
    pHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pID);
    if(!pHandle)
        return false;
    remoteLoadLib = (LPVOID)GetProcAddress(GetModuleHandle(L"kernel32.dll"),
        "LoadLibraryA");

    // error checking
    remoteString = (LPVOID)VirtualAllocEx(pHandle, NULL, strlen(dllpath)+1,
        MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    // error checking
    WriteProcessMemory(pHandle, remoteString, dllpath, strlen(dllpath)+1, NULL);
    threadHandle = CreateRemoteThread(pHandle, NULL, NULL,
        (LPTHREAD_START_ROUTINE)remoteLoadLib, remoteString, NULL, 0);
    // error checking
    return true;
}
```

openProcess יכול גם להיכשל במידה שאין לנו גישה להרשאות שביקשנו.

לעוד מידע בנושא הרשאות ניתן לעיין ב-MSDN בנושא Access Control Model.



הלדקת DLL

השתמשנו בפונקציות הנ"ל בשקפים קודמים.
יש לשים לב שמבקשים את LoadLibraryA – כלומר פונקציה שמקבלת ASCII כקלט.

```
BOOL dllInjector(char * dllpath , DWORD pID)
{
    HANDLE pHandle , threadHandle;
    LPVOID remoteString;
    LPVOID remoteLoadLib;
    pHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pID);
    if(!pHandle)
        return false;

    remoteLoadLib = (LPVOID)GetProcAddress(GetModuleHandle(L"kernel32.dll"),
        "LoadLibraryA");

    // error checking
    remoteString = (LPVOID)VirtualAllocEx(pHandle, NULL, strlen(dllpath)+1,
        MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    // error checking
    WriteProcessMemory(pHandle, remoteString, dllpath, strlen(dllpath)+1, NULL);
    threadHandle = CreateRemoteThread(pHandle, NULL, NULL,
        (LPTHREAD_START_ROUTINE)remoteLoadLib, remoteString, NULL, 0);
    // error checking
    return true;
}
```

remoteLoadLib יחזיק את הכתובת של LoadLibrary בתהליך שלנו.
הכתובת הזאת זהה גם לתהליכים אחרים במחשב
כי kernel32.dll תמיד נטען לאותו המקום
שנקבע מראש בזמן טעינת מ"ה).



הלרקת DLL

```
BOOL dllInjector(char * dllpath , DWORD pID)
{
    HANDLE pHandle , threadHandle;
    LPVOID remoteString;
    LPVOID remoteLoadLib;
    pHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pID);
    if(!pHandle)
        return false;
    remoteLoadLib = (LPVOID)GetProcAddress(GetModuleHandle(L"kernel32.dll"),
                                           "LoadLibraryA");

    // error checking
    remoteString = (LPVOID)VirtualAllocEx(pHandle, NULL, strlen(dllpath)+1,
                                           MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    // error checking
    WriteProcessMemory(pHandle, remoteString, dllpath, strlen(dllpath)+1, NULL);
    threadHandle = CreateRemoteThread(pHandle, NULL, NULL,
    (LPTHREAD_START_ROUTINE)remoteLoadLib, remoteString, NULL, 0);
    // error checking
    return true;
}
```

- CreateRemoteThread

קריאה לפונקציה כ-thread חדש בתוכנית אחרת.

הפונקציה מקבלת כתובת לפונקציה וכתובת למשתנה - יש לשים לב שאלו כתובות בתוכנית אחרת - לא בזיכרון שלנו.

לכן צריך לדאוג שגם הכתובת של הפונקציה נכונה (בדקנו) וגם לדאוג שהכתובת של הפרמטר תקינה.



הלדקת DLL

```
BOOL dllInjector(char * dllpath , DWORD pID)
{
    HANDLE pHandle , threadHandle;
    LPVOID remoteString;
    LPVOID remoteLoadLib;
    pHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pID);
    if(!pHandle)
        return false;
    remoteLoadLib = (LPVOID)GetProcAddress(GetModuleHandle(L"kernel32.dll"),
        "LoadLibraryA");

    // error checking
    remoteString = (LPVOID)VirtualAllocEx(pHandle, NULL, strlen(dllpath)+1,
        MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    // error checking
    WriteProcessMemory(pHandle, remoteString, dllpath, strlen(dllpath)+1, NULL);
    threadHandle = CreateRemoteThread(pHandle, NULL, NULL,
(LPTHREAD_START_ROUTINE)remoteLoadLib, "file.dll", NULL, 0));
    // error checking
    return true;
}
```

מה היה קורה אילו במקום remoteString היינו כותבים "file.dll" ?



הלרקת DLL

```
BOOL dllInjector(char * dllpath , DWORD pID)
{
    HANDLE pHandle , threadHandle;
    LPVOID remoteString;
    LPVOID remoteLoadLib;
    pHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pID);
    if(!pHandle)
        return false;
    remoteLoadLib = (LPVOID)GetProcAddress(GetModuleHandle(L"kernel32.dll"),
                                           "LoadLibraryA");

    // error checking
    remoteString = (LPVOID)VirtualAllocEx(pHandle, NULL, strlen(dllpath)+1,
    MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    // error checking
    WriteProcessMemory(pHandle, remoteString, dllpath, strlen(dllpath)+1, NULL);
    threadHandle = CreateRemoteThread(pHandle, NULL, NULL,
    (LPTHREAD_START_ROUTINE)remoteLoadLib, remoteString, NULL, 0);
    // error checking
    return true;
}
```

שימוש ב-VirtualAllocEx (הקצאת זיכרון בתהליך מרוחק) ו-WriteProcessMemory (כתיבת זיכרון בתהליך מרוחק).
כך נבטיח שהמחרוזת "c:\\dllmsg.dll" נמצאת ב-Process המרוחק.



הלרקת DLL

- הצגנו דוגמא לשינוי קוד ע"י הזרקת DLL
 - יש דרכים נוספות לבצע זאת

- הצגנו את הדוגמא על תוכנית ייעודית, אך ניתן כמובן להזריק את ה-DLL לכל תהליך

- חפשו ב-MSDN כיצד לקבל רשימה של תהליכים רצים
- רמז: CreateToolhelp32Snapshot



הוק בסוף פונקציה

- כל מה שעשינו עד עתה מבצע הוק בתחילת פונקציה
 - כלומר מאפשר לנו להריץ קוד לפני שהפונקציה רצה
 - כולל אפשרות לשנות את הפרמטרים שהיא מקבלת כקלט
- לעיתים נרצה להחליף את הפונקציה באחרת בלי לקרא למקורית
 - איך נעשה זאת?
- במקרים אחרים נרצה להריץ קוד בסוף הפונקציה
 - למשל קוד שמשנה את ערך החזרה שלה
 - חישבו על לפחות שלוש דרכים לעשות זאת...
 - חשוב לשים לב לשני דברים:
 - איך לקבל את הבקרה בסוף חישוב הפונקציה?
 - איך נוכל לקרא לה למרות ששתלנו בה הוק?
- במקרים מורכבים יותר נרצה
 - לקרא לפונקציה המקורית מההוק יותר מפעם אחת



הוק בסוף פונקציה ב hot-patching

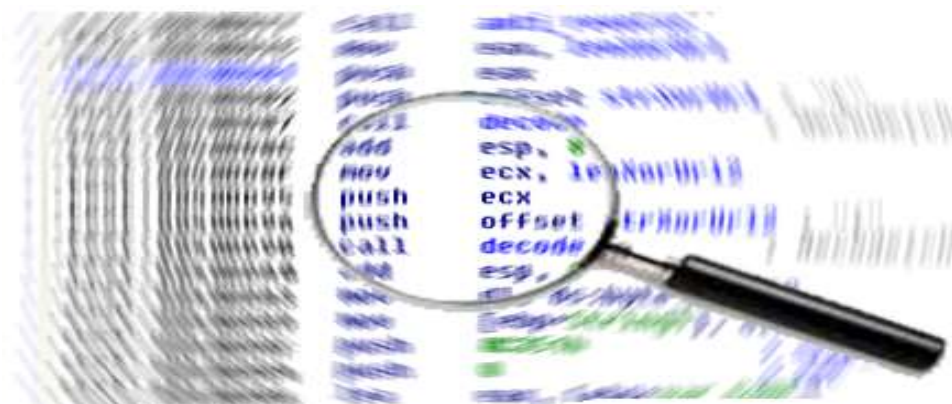
- במקרה שאין תמיכה ב hot-patching הפתרון נעשה מסובך יותר
- מה הבעיות הנוספות שנוצרות?
- איך ניתן לפתור אותן?

• רמז 1 : קריאה לפונקציה

• רמז 2 : ביצוע פקודות המכונה שמוחלפות ע"י jmp אחרי פקודת ה-call לפונקציה



הוק'נס בעולם האמיתי



אז מה היה לנו?

Dll Injection => SetHook() => HookProc()

- אפשר לשנות בקוד המקור
- אפשר לשנות בקובץ הבינארי
- יש פונקציות API המאפשרות hooking במקרים ספציפיים.
- קיימות עוד שיטות ל on-the-fly hooking



Instrumentation

ההרה: הכלים שידון הסעיף
לה אינם מותרים לשימוש
בתרעילי הבית, הסדנא או
במבחן



Pin Tutorial
(Robert Cohn, Intel)

Detours: Binary Interception
of Win32 Functions
(Galen Hunt and Doug Brubacher,
MS Research)



What is Instrumentation?

- A technique that inserts extra code into a program to collect runtime information

```
counter++;  
$0xff, %edx sub  
counter++;  
%esi, %edx cmp  
counter++;  
<L1> jle  
counter++;  
$0x1, %edi mov  
counter++;  
$0x10, %eax add
```



Instrumentation Approaches

- Source instrumentation:
 - Instrument source programs
- **Binary instrumentation:**
 - Instrument executables directly

Advantages for binary instrumentation

- ✓ Language independent
- ✓ Machine-level view
- ✓ Instrument legacy/proprietary software



Instrumentation Approaches

When to instrument:

- Instrument statically – before runtime
- Instrument dynamically – at runtime

Advantages for dynamic instrumentation

- ✓ No need to recompile or relink
- ✓ Discover code at runtime
- ✓ Handle dynamically-generated code
- ✓ Attach to running processes



How is Instrumentation used in Computer Architecture Research?

- **Trace Generation**
- **Branch Predictor and Cache Modeling**
- **Fault Tolerance Studies**
- **Emulating Speculation**
- **Emulating New Instructions**



How is Instrumentation used in Program Analysis?

- **Code coverage**
- **Call-graph generation**
- **Memory-leak detection**
- **Instruction profiling**
- **Data dependence profiling**
- **Thread analysis**
 - Thread profiling
 - Race detection



Detours

- Is a library for instrumenting and intercepting function calls in Win32 binaries.
- Replaces the first instructions of a *target* function with jmp to a *detour* function.
- Preserves original function semantics through a *trampoline* function.
- Enables interception and instrumentation of Win32 binary programs.



Detouring a Function:

Before:

```
;; Target Function
Sleep:
    push    ebp          [1 byte]
    mov     ebp,esp      [2 bytes]
    push    ebx          [1 bytes]
    push    esi          [1 byte]
    push    edi
    ....
;; Trampoline Function
UntimedSleep:
    jmp     Sleep
;; Detour Function
TimedSleep:
    ....
```

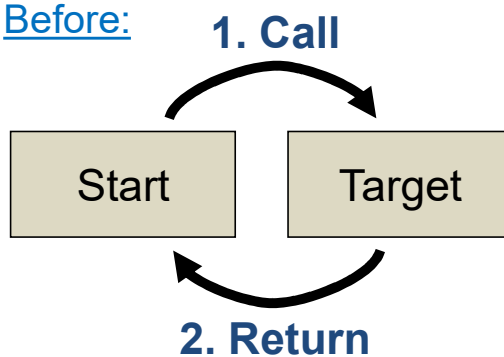
After:

```
;; Target Function
Sleep:
    jmp     TimedSleep [5 bytes]
    push    edi
    ....
;; Trampoline Function
UntimedSleep:
    push    ebp
    mov     ebp,esp
    push    ebx
    push    esi
    jmp     Sleep+5
;; Detour Function
TimedSleep:
    ....
```

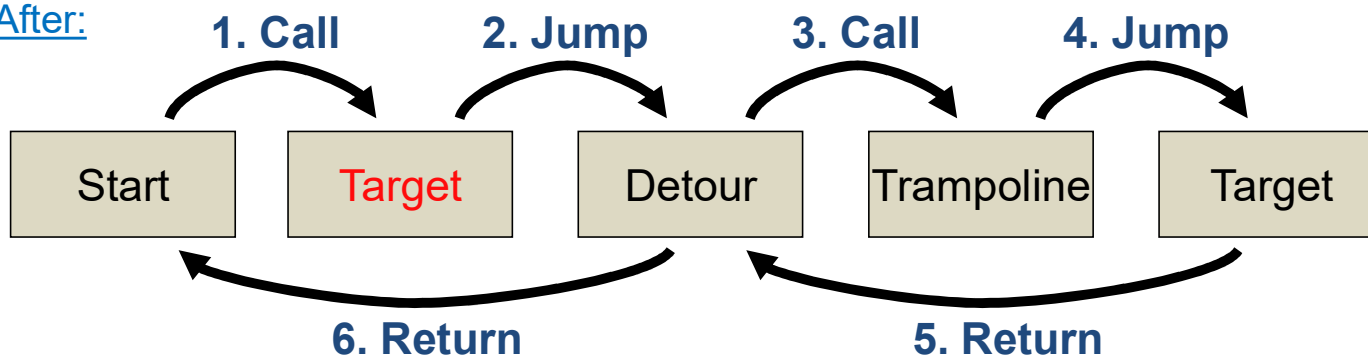


Invoking Your Code:

Before:



After:



Pin Instrumentation APIs

- Basic APIs are architecture independent:
 - Provide common functionalities like determining:
 - Control-flow changes
 - Memory accesses
- Architecture-specific APIs
 - e.g., Info about opcodes and operands
- Call-based APIs:
 - Instrumentation routines
 - Analysis routines



Instrumentation vs. Analysis

- **Instrumentation routines** define where instrumentation is inserted
 - e.g., before instruction
 - ☞ **Occurs *first time* an instruction is executed**
 - ☞ Usually defined in the tool "main"
 - ☞ Heavy lifting
- **Analysis routines** define what to do when instrumentation is activated
 - e.g., increment counter
 - ☞ **Occurs *every time* an instruction is executed**
 - ☞ Usually defined in instrumentation routine
 - ☞ As light as possible



ManualExamples/inscount0.cpp

```
#include <iostream>
#include "pin.h"
```

```
UINT64 icount = 0;
```

```
void docount() { icount++; }
```

analysis routine

```
void Instruction(INS ins, void *v)
```

instrumentation routine

```
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}
```

```
void Fini(INT32 code, void *v)
```

```
{ std::cerr << "Count " << icount << endl; }
```

```
int main(int argc, char * argv[])
```

```
{
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```



Instrumentation Approaches

- JIT Mode

- Pin creates a modified copy of the application on-the-fly
- Original code never executes

➤ More flexible, more common approach

- Probe Mode

- Pin modifies the original application instructions
- Inserts jumps to instrumentation code (trampolines)

➤ Lower overhead (less flexible) approach

