

Digital Whisper

גליון 52, יולי 2014

מערכת המגזין:

מייסדים:

אפיק קסטיאל, ניר אדר

מוביל הפרויקט:

אפיק קסטיאל

עורכים:

שילה ספרה מלר, ניר אדר, אפיק קסטיאל

כתבים:

עומר דיין, דוד א., ראג'דיפ קאנאבארה ודניאל ליבר

יש לראות בכל האמור במגזין Digital Whisper מידע כללי בלבד. כל פעולה שנעשית על פי המידע והפרטים האמורים במגזין Digital Whisper הינה על אחריות הקורא בלבד. בשום מקרה בעלי Digital Whisper /או הכותבים השונים אינם אחראים בשום צורה ואופן לתוצאות השימוש במידע המובא במגזין. עשיית שימוש במידע המובא במגזין הינה על אחריותו של הקורא בלבד.

פניות, תגובות, כתבות וכל הערה אחרת - נא לשלוח אל editor@digitalwhisper.co.il

דבר העורכים

ברוכים הבאים לגיליון ה-52 - גיליון יולי.

חודש יולי הגיע, חודש יולי של שנת 2014, וזה אומר שעברו בדיוק 21 שנים מאז שג'ף מוס ("DarkTangent") וחבריו נפגשו לראשונה בלאס-וגאס והתחילו את המורשת של כנס ה-DEF-CON וכנס ה-Blackhat. וגם השנה, בעוד קצת פחות מחודש, יתקיים בוגאס הכנס [DEF-CON 22](#) ומיד לאחריו, [Blackhat USA 2014](#).

קשה לי להאמין שיש מישהו שקורא את השורות האלה ולא שמע על צמדי המילים הללו, אך בכל זאת, למי שלא יודע, מדובר בדרך כלל בשבוע שלאחריו יש לא מעט רעש. הרבה מאוד חוקרי אבטחת מידע מעדיפים לפרסם את ממצאיהם מעל במות האירוע, [החלטות אשר לא פעם אף גרמו למעצריהם](#).

מה שנחמד לראות בכנסים האלה (מלבד כמובן את ההרצאות האיכותיות ואת הסדנאות המעולות), [זה שכמעט בכל כנס, יש, באופן יחסי, לא מעט ישראלים שמרצים](#). אם תספרו, תגיעו למספר די מרשים של מרצים ישראלים ביחס לגודל המדינה שלנו וביחס לכמות המרצים שיש בכנסים האלה. אני לא נכנס לסיבות, האם זה בגלל המצב הבטחוני, או האם זה בגלל יחידות כמו 8200 וכדומה, וזה לא מעניין אותי גם. זה נכון שלא כולם גדלו פה בקהילה, אבל הנתון הזה עדיין מחמם את הלב.

זה נחמד לראות שלמרות שבהרבה מקומות בעולם, ונראה לעיתים שבמיוחד פה בארץ, יש את ההזנחה הזאת של נושא אבטחת המידע ותרבות ה-"לי זה לא יקרה", עדיין יש לנו פה לא מעט מוחות טובים ששמים את נושא אבטחת המידע בראש מעייניהם ותורמים בעזרת הידע שלהם לשאר העולם.

וכמובן, בנוהל - לפני שנגש לחלק העיקרי של הגיליון, נגיד תודה רבה לכל מי שבזכותם הגיליון פורסם החודש, כל אותם האנשים שהשקיעו מזמנם וממרצם לטובת הקהילה וכתבו מאמרים: תודה רבה לעומר דיין, תודה רבה לדוד א., תודה רבה לראג'דיף קאנאבארה, תודה רבה לדניאל ליבר וכמובן, תודה רבה לשילה ספרה מלר, על היותה העורכת מספר אחת שלנו.

קריאה מהנה!

נר אדר ואפיק קסטיאל.



תוכן עניינים

2	דבר העורכים
3	תוכן עניינים
4	About Memory And Its Weakness
28	Reverse Engineering - לגלות את הסודות החבויים
41	אבטחת קוד פתוח ע"י ניתוח קוד סטטי
60	אז למה לי פוליטיקה עכשיו? / על רגולציה ואבטחת מידע
68	דברי סיכום

About Memory And Its Weakness

מאת עומר דיין

הקדמה

כאשר מדברים על מחשבים, הזיכרון הוא כלי מאוד חשוב, בין אם מדברים על סתם לכתוב עבודה בוורד, או שמדברים על בעיות אבטחה. במאמר הזה אסביר על מספר טכנולוגיות בעזרתן המחשב מנהל את הזיכרון, אסביר על איך השיטות הללו עוזרות לנו להבין את המחשב, ואיך אנו יכולים לנצל אותן לטובתנו בעת ניתוח חולשות וכתובת אקספלויטים לניצולן. להבין את הזיכרון זה אחד הדברים החשובים עבור הבנה של המחשב.

זיכרון - מהו?

זיכרון הוא אוסף תאים המכילים 0 או 1. בעזרת ערכים אלה הזיכרון יכול לייצג כל מה שאתם מכירים. בין אם זה בעזרת מתח חשמלי נמוך שמסמן 0 ומתח חשמלי גבוה שמסמן 1 (כך פועל הזיכרון RAM) או בין אם זה בעזרת מטען מגנטי כאשר כל אחד מהמטענים המגנטיים (שלילי או חיובי) מסמן 0 או 1 (כך פועל זיכרון הדיסק הקשיח). במדעי המחשב כל תא כזה נקרא סיב (ביט, Bit). כל שמונה סיביות נקראים בית (byte). שני בתים נקראים מילה (Word). 1024 בתים זה כבר קילו-בית. 1024 קילו-בית זה כבר מגה-בית. 1024 מגה-בית זה ג'יגה-בית. וכך זה ממשיך.

במאמר הזה, אנו נדבר על איך המחשב מנהל את הזיכרון שלו בזמן ריצת תוכנית. בין אם זה תוכנית



[במקור: <http://www.soyouwanna.com>]

שאנו פיתחנו ובין אם זה תוכנית שפותחה על ידי גורם אחר.

הזיכרון של המחשב כמו שהזכרנו קודם הוא ה-RAM והזיכרון של הדיסק הקשיח, ה-HardDisk-HD. הזיכרון של הדיסק הקשיח הוא עבור שמירת נתונים בצורה פרסיסטנטית (קבועה) ואילו זיכרון ה-RAM הוא עבור תפעולה השוטף של התוכנית.

אנו נתמקד בזיכרון RAM, בזמן ריצת התוכנית שלנו, כאשר המחשב מנהל את המשתנים שאנו רוצים לשמור או תאים שאנו רוצים להקצות עבור שימוש עתידי.

Stack&Heap - המחסנית והזיכרון המבוזר

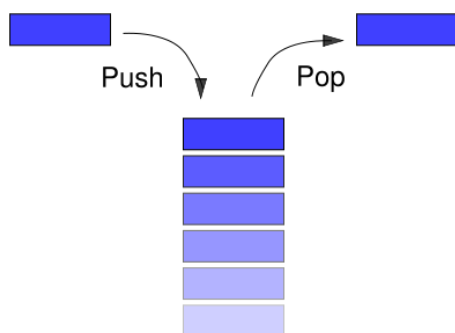
כל אחד מהם בעל שיטת פעולה שונה ועבור מטרה שונה. כאשר אנו מפעילים את התוכנית מערכת ההפעלה דואגת להקצות לנו שטח זיכרון אשר התוכנית שלנו תצטרך להתנהל בתוכו בלבד. זאת אומרת שהתוכנית שלנו לא תוכל לפעול בשטחי זיכרון שלא שייכים לה, יותר מזה, סביר להניח ששטחי הזיכרון האלה שייכים לתוכניות אחרות כך שאסור לה בשום פנים ואופן לפעול על שטחים אלה. הפלא ופלא גם שטח הזיכרון אשר מערכת ההפעלה הקצתה לתוכנית שלנו מתחלקת לכמה סגמנטים עיקריים (לכמה תת-שטחים עיקריים). לכל אחד מהסגמנטים תפקיד מיוחד, אם זה לשמירת נתונים סטטיים או לשמירת נתונים באופן מבוזר או לשמירת הנתונים לשימוש באופן שוטף ואפילו לשמירת מערך הפקודות של התוכנית. העיקריים מבניהם שבאמת מעניינים אותנו במאמר הזה זה המחסנית והזיכרון המבוזר (The stack and the heap).

המחסנית תפקידה לשמירת נתונים לתפעולה השוטף של התוכנית שלנו. כאשר אנו מגדירים משתנה שהוא Value Type חדש הוא ישמר על המחסנית לדוגמא:

```
Int x = 3;
```

המשתנה x יישמר במחסנית. משתנים שהם value type יישמרו גם הם במחסנית. משתני value type הם שהערך שלהם נשמר במחסנית. וזאת בניגוד ל-referense type אשר לא נשמרים במחסנית אלא על הזיכרון המבוזר ואילו מצביע עליהם נשמר במחסנית.

לפני שנתקדם אל הזיכרון המבוזר נסביר איך המחסנית פועלת. שיטת LIFO (Last in first out) בתרגום חופשי - האחרון שנכנס הראשון שייצא. בדיוק כמו מחסנית של רובה. אתה מכניס כדורים והם יורדים



למטה ככל שאתה מכניס יותר. כך שכאשר תוציא אחד מהם אתה תוציא את העליון, את האחרון שהכנסת. אז במחסנית של התוכנית אנו מכניסים משתנים וכאשר נרצה להוציא משתנה אנו נוציא תחילה את האחרון שהכנסנו ולאחר מכן את זה שלפניו וכך עד שנוציא את הראשון שהכנסנו. באופן דיפולטי גודל המחסנית בתוכנית שאנו נפתח יהיה כמגה בית אחד.

[במקור: sir.unl.edu]



הערה חשובה: כאשר אנו ניגשים למשתנה השמור על המחשנית אנחנו לא (!) מוציאים את כל המשתנים שהכנסנו לפניו. אנו פשוט קוראים מהכתובת בזיכרון. כאשר אנו נרצה למחוק משתנה אנו נצטרך להוציא את אלו שמעליו.

הזיכרון המבוזר.

מרחב זיכרון אשר בו אנו נאגור נתונים. כאשר מרחב זיכרון זה לא פועל בשיטת LIFO, למעשה אין בדיוק שיטה בה הוא פועל. זאת אומרת שאם נרצה לגשת אל משתנה שכתוב בתוכו פשוט ניגש אל הכתובת של המשתנה הזה. חד וחלק. על הזיכרון המבוזר, ההיפ נשמרים גם נתונים שהם Refrence type זאת אומרת משתנים שבנוסף לכך שהם נשמרים בזיכרון המבוזר הכתובת שלהם נשמרת במחשנית. בשפת ++C יש אפשרות לשמור נתונים על הזיכרון המבוזר ע"י המילה השמורה new. לדוגמא:

```
Dog myDog = new Dog ();
```

בשורה זאת אנו מגדירים "כלב חדש" אשר הוא עצמו ישמר על ה-HEAP ומצביע אליו ישמר על המחשנית.

Cdecl - שיטת העברת פרמטרים לפונקציה

בהנתן המצב הבא: אנחנו המחשב. עכשיו אנו מפעילים תוכנית מסוימת. בפקודה מסוימת אנו צריכים לקפוץ אל פונקציה חיצונית ולשלוח לה שלוש נתונים, פרמטרים. כאשר הפונקציה תגמר נחזור אל התוכנית המקורית. איך בעצם נעשה את זה?!

עומדות בפנינו מספר בעיות:

- הפונקציה החיצונית לא מכירה את המשתנים של התוכנית שלנו, זאת אומרת שהיא לא יכולה להשתמש בהם.
- אנחנו צריכים לקפוץ אל הפונקציה החיצונית ולאחר מכן כאשר תגמר לקפוץ אל הפונקציה הראשית (התוכנית המקורית).

על מנת לפתור בעיות אלו נוצר הפרוטוקול הזה cdecl, החוקים והכללים הללו של איך כל הפרוצדורה הזאת תתנהל.

איך זה פועל? אז כאשר הבנו מה הבעיות שעומדות בפנינו הגענו למסקנה שהדרך הטובה ביותר היא לשמור את המשתנים בזיכרון ואז לגשת אליהם דרך הפונקציה. ואיך נעשה את זה? פשוט מאוד. את המשתנים שאנו רוצים לשלוח לפונקציה נדחוף למחשנית. בתוך הפונקציה נשלוף אותם. אבל קיימת אצלנו עוד בעיה. אנו רוצים לקפוץ לפונקציה ולחזור ממנה. החלק של הקפיצה אל הפונקציה הוא פשוט. אם יש לנו את הפונקציה אנו יודעים גם מה הכתובת שלה כאשר התוכנית מתקמפלת. אבל מה לגבי הקטע של

לחזור לתוכנית המקורית בדיוק אל אותה פקודה שבה הפסיקה התוכנית את פעולתה? גם את זה נעשה בעזרת המחסנית.

טכנית זה נראה ככה: כאשר פונקציה קוראת אל פונקציה חיצונית קוראים הדברים הבאים:

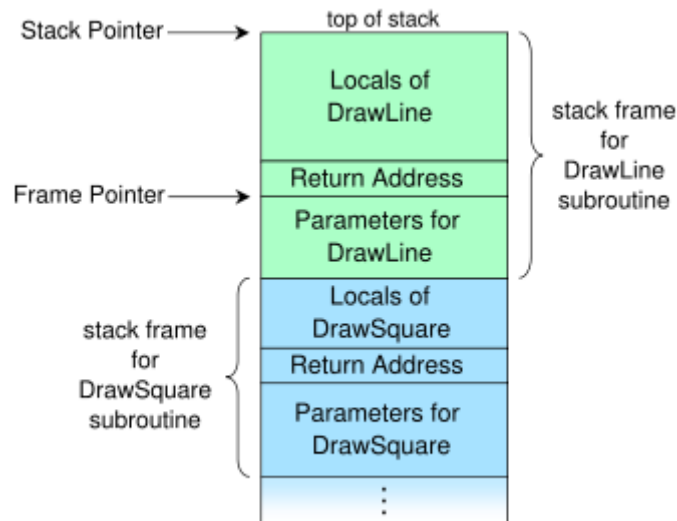
1. דחוף את הכתובת של הפעולה הבאה (בשביל שהתוכנית תדע לאיזה פעולה הגענו).
2. דחוף את המשתנים שאתה רוצה להעביר לפונקציה.
3. קפוץ אל הפונקציה.
4. (בתוך הפונקציה) שלוף את המשתנים.
5. (בתוך הפונקציה)... המימוש של הפונקציה כל מיני שקר ועוד שקר שווה שקר.
6. (בתוך הפונקציה) שלוף את הכתובת שנרצה לחזור אליה.
7. (בתוך הפונקציה) קפוץ אל הכתובת ששלפת.
8. תמשיך את הפונקציה המקורית.

<pre> 1 foo: 2 POP EAX 3 MOV [0x812356], EAX 4 POP EAX 5 MOV [0x812360], EAX 6 ... 7 POP EAX 8 GOTO EAX 9 10 main: 11 ... 12 MOV EAX, EIP 13 PUSH EAX 14 MOV EAX, [A] 15 PUSH EAX 16 MOV EAX, [B] 17 PUSH EAX 18 GOTO foo 19 ... </pre>	<pre> 1 void foo(int a, int b) 2 { 3 ... 4 } 5 6 int main() 7 { 8 ... 9 foo(A, B); 10 ... 11 } </pre>
---	---

כך נראית תוכנית שמממשת את החוקים של cdecl. כמובן שכאשר נקמפל תוכנית בשפה עילית המהדר ידע לממש את אופן זה בעצמו.

כך, עבור כל פונקציה שתקרא ייפתח לנו במחסנית המבנה הזה: כתובת חזרה אליה נקפוץ כאשר הפונקציה תגמר. הפרמטרים של הפונקציה. ולאחר מכן משתנים מקומיים שימחקו בסוף הפונקציה. כל מבנה כזה נקרה StackFrame. אם תנסו לחשוב על זה לרגע, לכל פונקציה שנפתחת כאילו יש מחסנית משלה שממנה היא מתחילה לכתוב ולקרוא. כאשר היא שולפת מהמחסנית איבר שהוא כתובת החזרה אל הפונקציה המקורית, הפונקציה יודעת שפה נגמרת המחסנית שלה.

תמונה שממחישה את ה-StakFrame:



[במקור: http://en.wikipedia.org/wiki/Call_stack]

Structred Exception Handling - SEH

Microsoft פיתחה שיטה לניהול התנהגות עם "חריגות" - התנהגות של התוכנית שלנו החורגת מגבולות ההתנהגות התאימה. זאת אומרת, כאשר התוכנית שלנו צריכה להתמודד עם משהו אשר לא תוכננה להתמודד אתו אין זה יהיה דבר מפתיע אם ההתנהגות שלה תהיה מעט חריגה מהרגיל. לדוגמא: כאשר יש לנו תוכנית שמקבלת מספר ומחזירה את ההופכי שלו, אז לא תהיה כל בעיה לשלוח לתוכנית שלנו מספר ולהיות בטוחים שהיא תחשב את ההופכי שלו. אבל מה אם נשלח לתוכנית שלנו מילה במקום מספר? אם במקום לרשום 3 בשורת הקלט נרשום "מכבי"? מה יקרה אז? סביר להניח שהתוכנית שלנו תפעל בצורה חריגה. המנגנון שמיקרוסופט פיתחו הוא מנגנון להתמודד עם בעיות אלו. במנגנון זה לתוכנית שלנו יש אפשרות להתריע על התנהגות חריגה שלה כך שהמפתח יוכל לעצור את התוכנית באותו מקרה (אם הוא מחליט כך כמובן). נראה דוגמא:

```

1 public int Div(int firstNumber, int secondNumber)
2 {
3     if(secondNumber == 0)
4     {
5         throw new Exception("Can't divide by Zero");
6     }
7     else
8     return firstNumber / secondNumber;
9 }

```


מה שאנו רואים כאן זה מתודה שאמורה לחלק שני מספרים שהיא מקבלת מבחוץ. מה שהיא עושה בפועל היא בודקת אם המספר השני הוא 0 ואם כן היא זורקת שגיאה שהיא לא יכולה לחלק באפס ואם המספר השני הוא לא אפס אז היא מחזירה את תוצאת החילוק. כמובן שמיותר לציין שאם המספר השני הוא אפס אז הפעולה זורקת את החריגה (Exception) ומפסיקה את פעולתה.

בפיתוח תוכנה כלי החריגות הוא כלי בעל כח רב. מעבר לכך שאפשר לזרוק חריגה בכל פעם שקורת בעיה חריגה בתוכנית על המפתח גם לדעת לתפוס את החריגה שנזרקה ולהתמודד אתה. נראה דוגמה לכך:

```
1 public void main(A LOT OF SHEKERS)
2 {
3     try{
4         Div(9, 0);
5     }
6     catch(Exception e)
7     {
8         Console.WriteLine(e.ToString());
9     }
10 }
```

מה שאנו רואים כאן זה את התוכנית הראשית אשר מנסה (!) לבצע חילוק של 9 ב-0. הבלוק try אומר לתוכנית לנסות לעשות את מה שכתוב בתוך הבלוק. במקרה שהצלחת תמשיך את התוכנית, במקרה שלא הצלחת תתפוס את החריגה שנזרקה ותדפיס אותה (catch זה הבלוק שתופס את חריגה). בתוך הבלוק של ה-catch מוגדרות רצף פעולות שנרצה להפעיל במקרה שבאמת קורת התנהגות חריגה ונזרק Exception.

פונקציה יכולה להחליט שהיא לא רוצה לטפל ב-Exception בעצמה אלא רוצה לזרוק את ה-Exception לדרג גבוה יותר, אל הפונקציה שקראה לה. מה שיקרה זה שהפונקציה main תראה ככה:

```
1 public void main(A LOT OF SHEKERS)
2 {
3     try{
4         Div(9, 0);
5     }
6     catch(Exception e)
7     {
8         throw e;
9     }
10 }
```

מה שהשתנה עכשיו זה שהתוכנית לא מדפיסה את החריגה אלא זורקת אותה אל הפונקציה שקראה לה. יכולות להיות הרבה סיבות לעשות דבר כזה, לדוגמה אם אנחנו משתמשים בממשק גרפי אז לפונקציה main אין את המחלקות ומשתנים המתאימים להדפיס לממשק הגרפי הודעת שגיאה אז היא תזרוק את

החריגה לדרג גבוה יותר שאם יש לו את הכלים המתאימים לעשות את זה אז הוא ידפיס ואם לא אז הוא יזרוק את זה גם כן לדרג שמעליו ככה עד שנגיע למתודה שכן תוכל להדפיס ותדפיס את הודעת השגיאה.

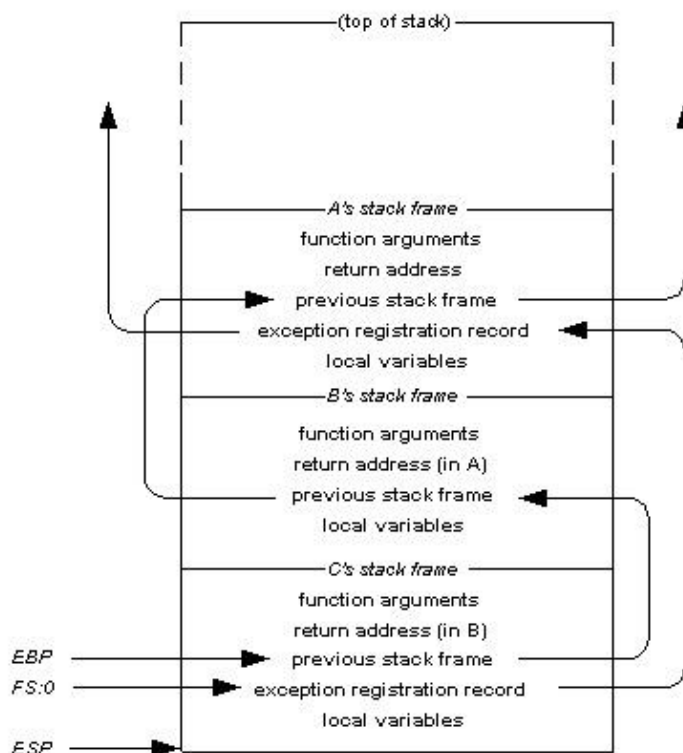
- נקודה חשובה: אם בכל המתודות שלנו נזרוק את השגיאה לדרג גבוה יותר ולעולם לא נטפל בה נגיע למצב בו הדרג הבא הוא מערכת ההפעלה והחריגה תזרק אל מערכת ההפעלה אשר תטפל בה (בהרבה מקרים הטיפול יהיה סגירת התוכנית שלנו מחשש שהחריגה בעיתית מאוד).

מה קורה בפועל ואיך זה קשור לזיכרון?

אז מה שקורה בפועל בכל פעם שאנו עושים try ו-catch בתוך ה-StackFrame שבתוך המחסנית שלנו נשמרים שתי דברים:

1. כתובת הטיפול בחריגה של הדרג שמעליו, זאת אומרת הפונקציה שקראה לפונקציה הנוכחית.
2. כתובת של הטיפול בחריגה בדרג שלנו (במקרה ולא נרצה להעביר את החריגה לדרג מעל).

תמונה קטנה שמתארת את ה-StackFrame עם ה-Exception handlers:



[במקור: <http://www.howzatt.demon.co.uk/articles/oct04.html>]

ניתוב התוכנית

לרוב ניתוב התוכנית יהיה אל מקטע קוד זדוני שהזרקנו ונרצה להפעיל, אך לא תמיד, בהרבה מקרים אנו נרצה פשוט לנתב את התוכנית אל מקטע אחר שקיים כבר בתוכנית.

לנו המשתמשים בתוכנית כלשהי לא תמיד יהיה לנו דרך לנתב את התוכנית לכל מקום שנרצה. יכול להיות שנוכל לנתב את התוכנית אל מספר מסלולים שהמפתח קבע מראש שאפשרי לנתב את התוכנית אליהם. אבל אנחנו, בתור האקרים בפוטנציה, רוצים יותר מזה - לנתב את התוכנית אל כל מקום שחושק ליבנו. תוכנית לא אמורה לאפשר לנו לעשות דבר כזה. ואנחנו בתור האקרים לא אמורים אפשר לתוכנית לנצח אותנו. אנחנו רוצים כן לנתב את התוכנית איך שאנו רוצים. ופה מגיע הקטע של פרצות אבטחה.

אנחנו צריכים איכשהו למצוא דרך לנתב את התוכנית אל מקום כלשהו שקבענו לעצמנו. אותה דרך תהיה פרצת האבטחה באותה תוכנה. קיימות הרבה אפשרויות לפרצות אבטחה בתוכנות וכל מפתח יודע להיזהר מהם. כמובן שסביר להניח שהוא לא מצליח להתגונן בפני כל מה שקיים. כל מה שאנחנו צריכים לעשות זה לחפש איזה פרצת אבטחה בתוכנה שאנו רוצים לפרוץ ולנצל את זה לטובתנו.

קצת לפני שאנו מדברים פרצות אבטחה קיימות, ננסה להכיר כמה דרכים שקשורות לנושא שאנו מדברים עליו בחלק זה של המאמר, ניתוב התוכנית.

בואו נחזור קצת אחורה, אל כל הנושאים שדיברנו עליהם עד עכשיו. כמה פעמים אמרנו שעבור פעולה או חוקים אנו, ואני מצטט, "שמים במחסנית את הכתובת, בשביל שהתוכנית תדע לאן להמשיך" (טוב זה לא בדיוק הציטוט המדויק של מה שנאמר במאמר, אבל זה מעביר את הרעיון). במילים אחרות אנחנו מכניסים לזיכרון את הכתובת אליה התוכנית תנוטב. בגדול, אם נצליח לשלוט על ערך הכתובת שנכנסת לתוך המחסנית אז אנו נוכל לנתב את התוכנית לכל מקום שנבחר. כי התוכנה פשוט תשלוף את הערך הזה שעבורה הוא ערך לגיטימי לכל דבר ותקפוץ אליו, וכל זה בלי לדעת שאת הכתובת אנחנו החדרנו מבחוץ ובאותה כתובת מחכה לה קוד זדוני.

- דיברנו על זה שכל StackFrame מכיל בתוכו ערך החזרה של הפונקציה שקראה אליו - Cdecl
- דיברנו על זה שמנהל החריגות שלנו שומר את הכתובת של המנהל חריגות של הדרג הגבוה יותר - SEH.

שליטה על אחד מהערכים האלה תביא לאותו האקד שליטה על ניתוב תוכנית. באותו רגע הוא ניצח. הוא מנתב את התוכנית לכל מקום שהוא רוצה במקרים מסוימים אפילו אל קוד זדוני שהוא עצמו הזריק.

לפני שנציג פרצת אבטחה שיכולה לעזור לנו לפרוץ תוכנית נסביר במילה על האוגר EIP שמאוד קשור אל כל הקטע של ניתוב התוכנית.

האוגר EIP הוא אוגר המכיל בתוכו את כתובת הזיכרון של הפקודה הנוכחית. עם כל הסתיימות של פקודה האוגר מתקדם אל הפקודה הבאה. דהיינו הכתובת משתנה אל כתובת הפקודה הבאה. כמו שאתם בטח מבינים שליטה על אוגר זה גורמת לשליטה על ניתוב התוכנית. שכן אם יכולנו לבחור מה יהיה כתוב שם אז היינו בוחרים איזה פקודות יפעלו.

בואו נראה פרצת אבטחה מאוד מפורסמת שבעזרתה אנו יכולים לבצע מניפולציות רבות על הזיכרון.

Buffer Overflow - גלישת מחסנית

הסבר הכי פשוט שאפשר לדבר הזה זה גלישת מידע מחוץ לגבולות המוגדרים של אותו מידע. ואם נגיד יותר מזה אז סביר להניח שלאחר אותם גבולות של אותו buffer יש מידע ששייך לחלק אחר של התוכנית.

תדמיינו לכם את הזיכרון בתור מערך בתים בשורה נגיד שהגודל הכולל שלו הוא 10 בתים, פשוט מאוד, שורה של 10 תאי מערך שכל אחד מהם בגודל של בית, שמונה סיביות.



עכשיו נגיד שאני מגדיר שה-4 בתים הראשונים מצד שמאל שייכים לספיידרמן, וה-6 בתים האחרים שייכים לבאטמן. בכל אחד מהתאים שמור "כח על" שאותו גיבור יכול לעשות. בהתחלה אני כותב לספיידרמן 2 כוחות על. ולבאטמן גם כן 2 כוחות על. איך הזיכרון שלנו יראה? יראה ככה:



אם אכותב לכל אחד מהם עוד שתי כוחות על. המערך יראה כך:



יש עכשיו לספיידרמן 4 כוחות על לאחריהם במערך מגיעים עוד 4 כוחות על של באטמן ולבסוף שתי בתים ריקים ששייכים לבאטמן.



בואו נגיד שאני ממשיך לכתוב לכל אחד מהם 2 כוחות על. המערך יראה ככה:

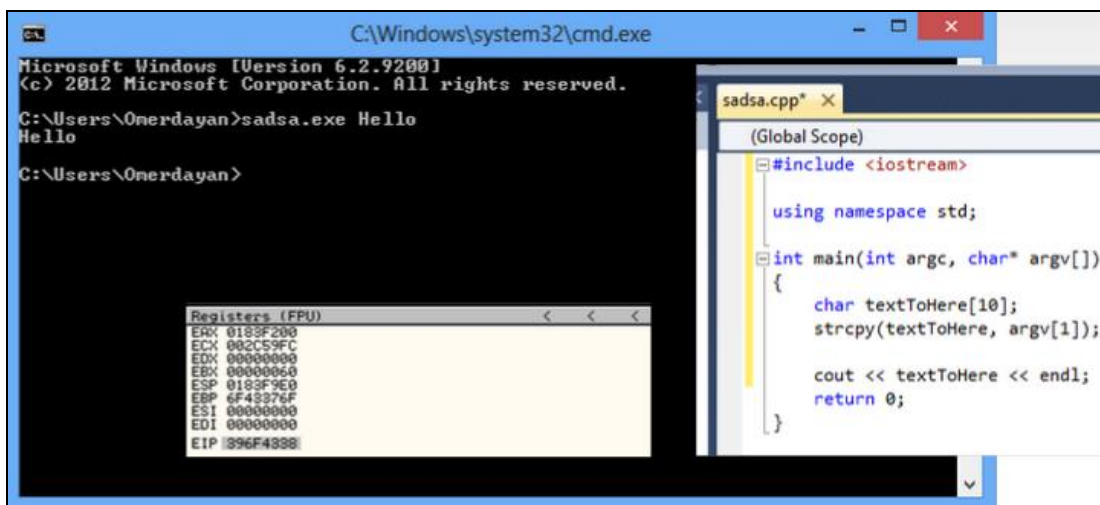


שימו לב מה קרה. לספיידרמן לא נשאר מקום לכתוב בו כוחות על ולכן כאשר בכל זאת כתבתי לו כוחות על הם נכתבו על ידי כך שהם דרסו, מחקו, את כוחות העל הראשונות של באטמן. ובאטמן מבחינתו כתב שתי כוחות על בתאים החופשיים שנשארו לו.

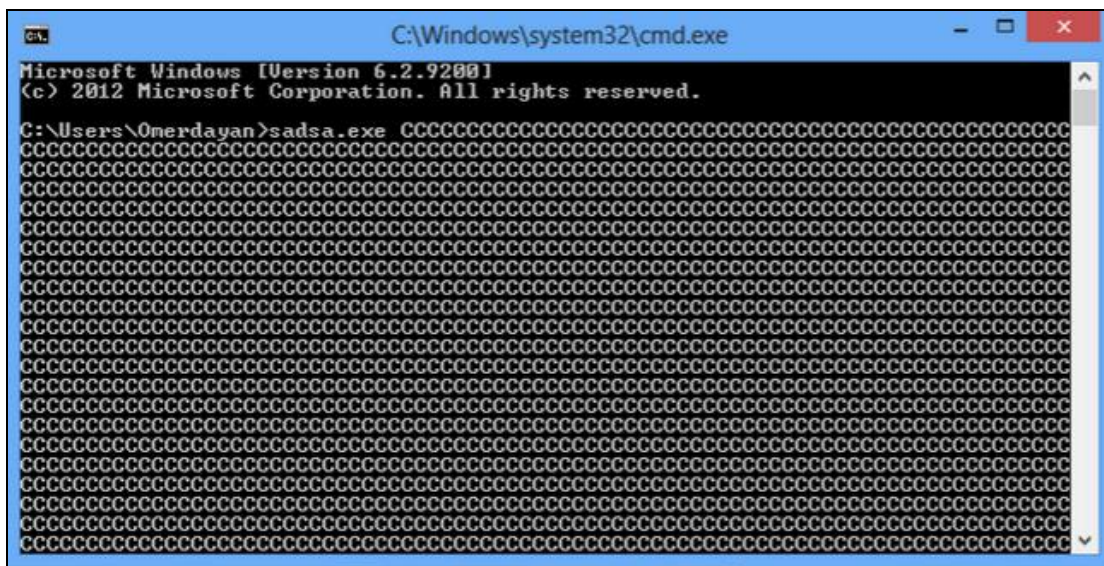
אם זה היה קורה באמת, מה היה קורה לבאטמן? הוא היה מנסה לבצע את כח העל הראשון שלו (לזמן עטלפים או שקר כזה או אחר) אבל מכיוון שכח זה נמחק ובמקומו נכתב כח על אחר (כח ששייך לספיידרמן) אז בטעות יצא לו כורי עכביש.

מה שקרה בעצם זה שחרגנו מגבולות שהוגדרו לנו מראש, ודרסנו מידע שמשמש במקום אחר בתוכנית. כל זה טוב ויפה אבל איך זה עוזר לנו? תחשבו על זה ככה: אמרנו שהמשתנים שלנו כתובים על המחסנית. וכתובות ניתוב של התוכנית גם כן כתובות על המחסנית. אפשר לדרוס אותם.

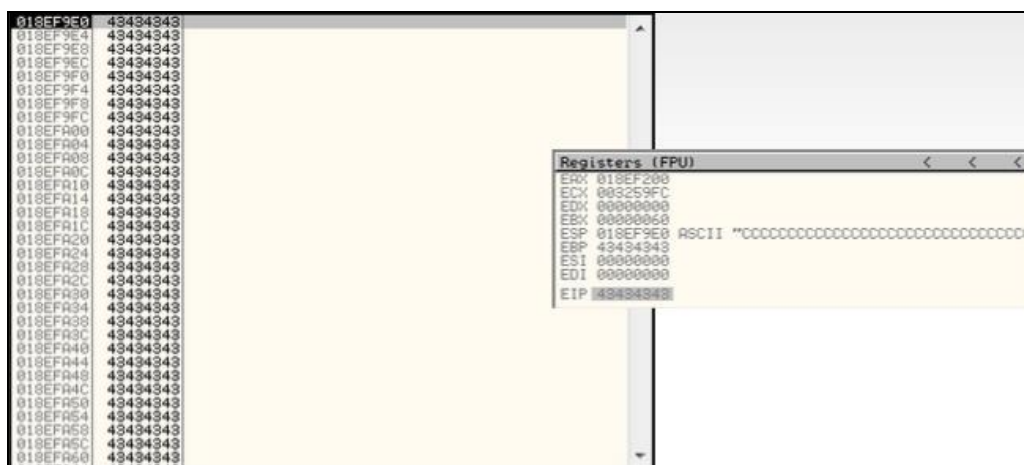
אנחנו מקבלים ניתוב מלא של המשך התוכנית. במקרה שהזרקנו קוד זדוני אנו נוכל לנתב את התוכנית שלנו לשם. נראה דוגמא לאיך זה פועל.



שימו לב לתוכנית הכתובה בצד ימין. זאת תוכנית פגיעה. היא מגדירה מערך תווים בגודל 10 תווים ומעתיקה לשם את התווים שהיא קיבלה כפרמטרים. וכל זאת בלי לבדוק האם זה אפשרי ולא חורג מגבולות המערך. אז הפעלה של התוכנית עם המחרוזת Hello תריץ את התוכנית כראוי כפי אתם רואים (במיוחד שימו לב למצב האוגרים כפי שניתן לראות בתמונה, אין שום דבר יוצא דופן).



ננצל את פרצה זו על מנת לדרוס כל מידע לאחר המערך על ידי כך שנשלח קלט ארוך ממה שהתוכנית מצפה לקבל.



כל הזיכרון נדרס ובכל מקום נכנס הערך "C" (בהמרת התו C לערך הקסהדצימלי יוצא לנו 43). יותר מזה, שימו לב למצב האוגרים בצד ימין, ובמיוחד למצב האוגר EIP אשר אחראי להמשך התוכנית, גם הוא נדרס. הקלט שהכנסנו הצליח לשנות את האוגר הזה. מה שאומר שאנחנו יכולים לשלוט עליו.

הצעד הבא יהיה לבדוק איזה מבין כל תווים שהכנסנו דרסו את האוגר הזה. הרי הכנסנו מאות של תווים ורק ארבעה מהם באמת משנים את כתובת האוגר. כל מנת שנוכל לשנות אותו לכל מה שאנו רוצים, נרצה לדעת איזה מהתווים דורסים את האוגר. הרעיון הכללי יהיה ליצור מחרזות ארוכה שלא חוזרת על עצמה בשום מקום, ולהכניס אותה לתוך התוכנית. נקח את הרצף שדרס את האוגר (אותו כמובן נראה בדיבאגר (אותה תוכנית שאיתה אני מצלם לכם את מצב האוגרים)) ונבדוק את המיקום שלו במחרזות. מכיוון שהמחרזות לא חוזרות על עצמה בשום מקום נקבל רק רצף אחד כזה. כאשר נדע לאחר כמה תווים, שלא

About Memory And Its Weakness

www.DigitalWhisper.co.il

ממש משנה לנו מה יהיה כתוב בהם, מגיעים התווים הדורסים נוכל לדרוס את האוגר כפי שנרצה (על ידי שינוי התווים האלה, כמוכן). על מנת לבצע את הרעיון הזה נשתמש בכלי בשם mona שממומש בפיתון.

```

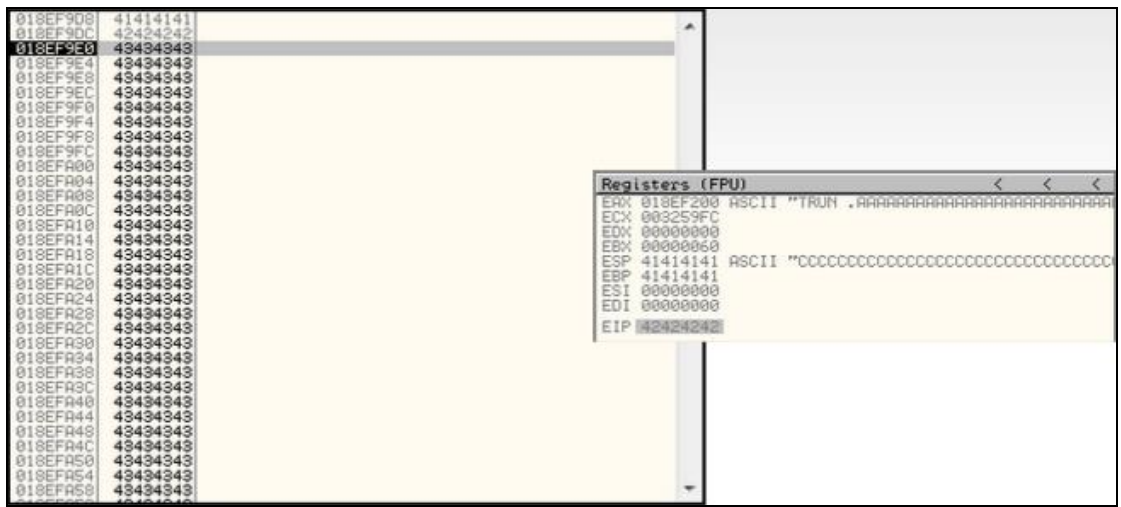
-----
output generated by mona.py v1.0
Corelan Team - http://www.corelan.be
-----
OS : xp, release 5.1.2600
Process being debugged : KMPPlayer (pid 3432)
-----
Pattern of 5000 bytes :
-----
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0
Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1
Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2
Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3
Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4
Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5
Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6
Bc7Bc8Bc9BdBd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7
Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8
Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9
Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0
Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9BxBx1
Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2
Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3
Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4
Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5
Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6
Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7
Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8
Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9
De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0
-----

```

את המחרוזת נכניס לתוך התוכנית ונמצא כמה תווים צריך לחרבש בשביל להגיע עד לתווים שידרו לנו את האוגר. (אם לא בא לכם להיכנס ל-cmd ולרשום PROGRAM.EXE ואז להתחיל לכתוב את כל המחרוזת הזאת, תו, ואני מניח שלא בא לכם) תכתבו קובץ BATCH שיעשה את זה (כי לקובץ הזה בניגוד לחלון CMD כן תוכלו לבצע הדבקה למחרוזת של mona), לא יותר מדי מסובך.

במקרה שלי יצא As2A שנמצא 348 תווים לאחר תחילת המחרוזת דורסים את האוגר.

על מנת לוודא זאת עכשיו נכניס לתוכנית 348 תווי A. לאחר מכן 4 תווי B ואז עוד רצף של תווי C. אם הכל ילך כשורה אנו אמורים לראות שהאוגר EIP נדרס וכתוב בו ארבעה תווי B (42 בבסיס הקסהדצימלי). כל חריגה של תו לפה או לשם אמורה להיראות באוגר.





אנו רואים שהצלחנו לדרוס את האוגר. זאת אומרת שאנו מצליחים לנתב את התוכנית לאיזה כתובת שאנו רוצים, על ידי שינוי אותם ארבעת תווי B לכל ערך, כתובת, שנרצה). בואו נקח את זה צעד אחד קדימה. איך נריץ קוד שאנו מזריקים בפנים?

אז כמו שלמדנו בנושא של ארכיטקטורת פון ניומן אנו צריכים להזריק את הקוד בשפת מכונה ולנתב את התוכנית שלנו אל אותו קוד. הקוד הזה נקרא ה-Payload.

קיימות מספר בעיות. ניתוב התוכנית אל הכתובת של ה-Payload. ואיפה נזריק את ה-Payload.

על הבעיה השנייה כבר התגברנו. אנחנו מצליחים לנתב את התוכנית למקום שאנו רוצים. והאמת היא שגם על הבעיה הראשונה התגברנו. במקום רצף של 348 תווי A שלא עושים כלום אנו נזריק את הקוד שלנו. וכמובן אחרי זה ננתב את התוכנית אל אותו קוד שלנו.

אז איפה הבעיה?

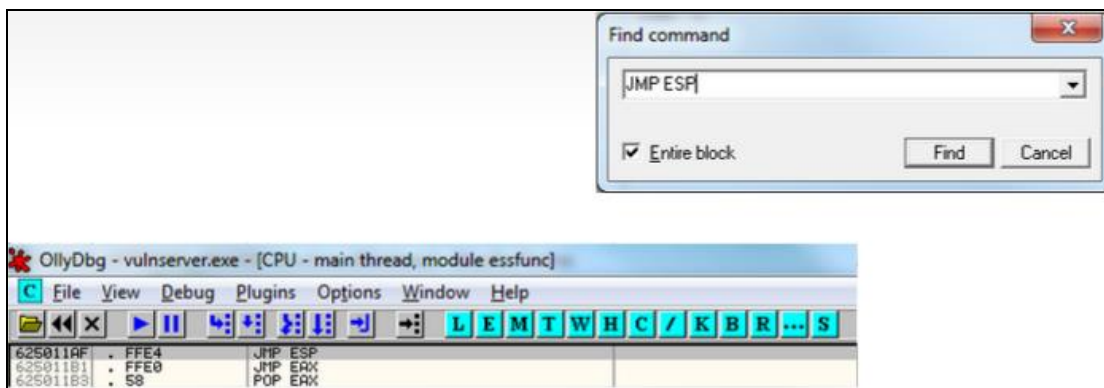
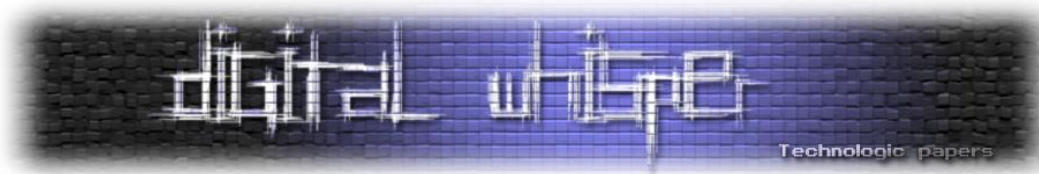
הבעיה היא שאנו לא יודעים לאיזה תא בזיכרון הקוד שלנו יוזרק. אם היינו יודעים היינו מכוונים את התוכנית שלנו אל הכתוב הזו. אבל אנו לא יודעים.

איך מתגברים על הבעיה?

שימו לב למצב האוגרים בתמונה האחרונה, חוץ מהאוגר EIP שנדרס קיים עוד אוגר שנדרס. אוגר ESP. מה אם נמצא בעזרת mona לאחר כמה תווים סתמיים אנו דורסים אותו, כפי שעשינו לאוגר EIP ונדרוס אותו עם רצף של פקודות מעבד? אם נצליח, אנו נשאר רק עם בעיה קטנה, לנתב את התוכנית אל אוגר ESP. וזה האמת, פחות בעייתי. נוכל למצוא כתובת זיכרון שאומרת למעבד לקפוץ אל ESP. ותאמינו לי, לא חסרים כתובות כאלה. אנחנו צריכים מקום כלשהו, שבו כתובה השורה הבאה:

```
JMP ESP // ESP - קפוץ אל
```

כאשר ננתב את התוכנית אל הכתובת של שורה זו, התוכנית תבצע את הפקודה שכתובה שם, הפקודה אומרת לקפוץ ל-ESP, אז התוכנית תקפוץ לשם. כשהיא תגיע לשם תחכה לה המתנה שהכנו לה, קוד שהזרקנו שיופעל על גבי המחשב.



חיפוש בדיבאגר מביא אותנו אל שורה שאומרת למעבד JMP ESP זאת אומרת קפוץ אל ESP. אז כל מה שנשאר לנו זה לנתב את התוכנית אל הכתובת של הפקודה הזאת, ואת זה, כבר הצלחנו לעשות.

דוגמא נוספת - SEH

הדוגמא הקודמת הייתה דוגמא ל-Direct EIP. ה-Overflow קרה במחסנית ודרסנו היישר את כתובת האוגר EIP.

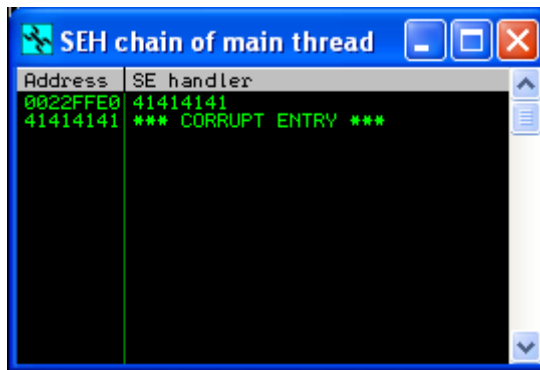
כעת, נראה דוגמא נוספת, הפעם של Structed Exception Endler - SEH Overflow (מנהל החרیגות שלנו). הפגיעות תהיה שוב ב-Overflow שיקרה במחסנית. נזכר רגע איך זה פועל. למחסנית נכנסים שני ערכים, שניהם כתובות. הראשונה היא למנהל החריגה, השנייה היא לחריגה הבאה (כך שנוצרת לנו רשימה מקושרת של כל החריגות). כאשר נזרקת חריגה או בודקים אם מנהל החריגה הראשון שלנו יכול לטפל בחריגה, תפעיל את פונקציית הטיפול, אם לא, תלך אל המנהל חריגות הבא ברשימה ותבדוק את זה גם עליו וכן הלאה. החולשה היא בכך שהתוכנית דוחפת למחסנית את כתובת המנהל חריגה ואת כתובת החריגה הבאה. אז כמובן שהשתלטות על כתובות אלו תיתן לנו השתלטות על ניתוב התוכנית.

נראה את זה קורה: הכנתי תוכנה פשוטה ופגיעה שלא עושה משהו מיוחד:

```
1 #include <cstdlib>
2 #include <iostream>
3
4 using namespace std;
5
6 int DoSomethingElse ()
7 {
8     cout << "Enter number else" << endl;
9     char s[10];
10    cin >> s;
11    cout << s << endl;
12    cin >> s;
13    return 2;
14    throw 1;
15 }
16
17 int DoSomething()
18 {
19     cout << "Enter number" << endl;
20     char s2[10];
21     cin >> s2;
22     cout << s2 << endl;
23     try{
24         throw 1;
25         DoSomethingElse ();
26     }
27     catch(...)
28     {
29         cout << "Catch > DoSomething" << endl;
30     }
31     return 2;
32     throw 1;
33     return 2;
34 }
35
36 int main(int argc, char *argv[])
37 {
38     try
39     {
40         DoSomething();
41     }
42     catch(...)
43     {
44         cout << "Exception!" << endl;
45     };
46     system("PAUSE");
47     return EXIT_SUCCESS;
48 }
```

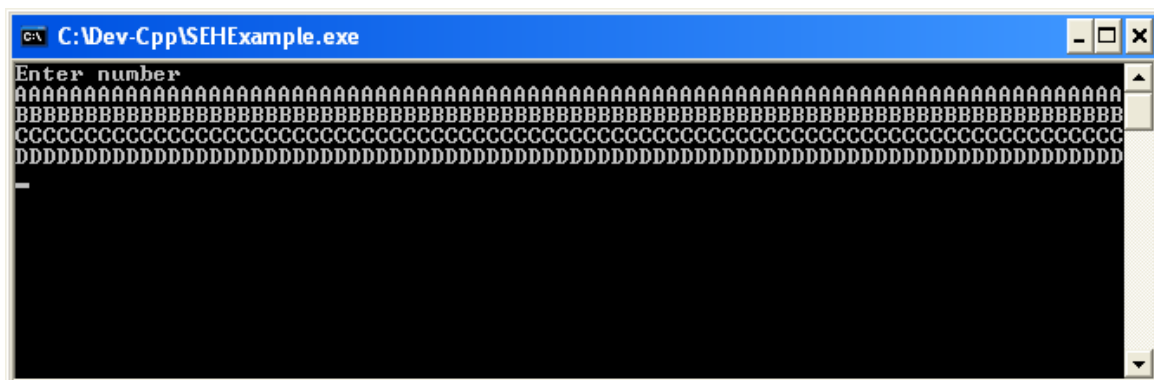
התוכנה פשוט מקבלת מחרוזת ומעתיקה אותה בתוך בלוק try באופן לא בטוח. (שימו לב שמטעמי נוחות של דיבוג התוכנית יכולה לקבל את המחרוזת כפרמטר או כקלט בתוכנית) (הנקודה של הדוגמה הזאת היא להראות רק איך החולשה מנוצלת)

בואו נראה את התוצאה ב-SEHChain:

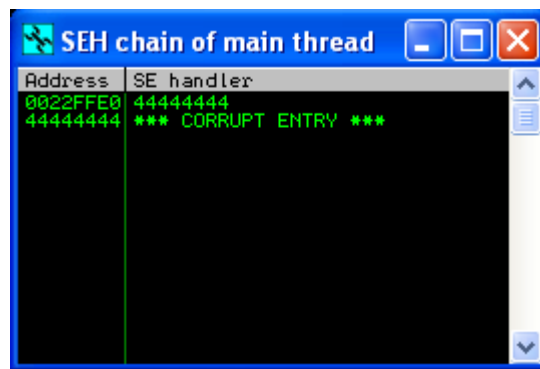


אנו רואים שהצלחנו לדרוס את הכתובת של החרیגה, ואת כתובת החריגה הבאה (בגלל זה הוא לא מוצא עוד חריגות), ואנו רואים שדרסנו אותם עם התו '0x41' שזה התו 'A'.

אם אנו יודעים שהתווים הדורסים הם ברצף ה-'A' אז אנו יודעים בערך איפה המיקום שלהם - ב-4 השורות הראשונות של הקלט בקונסול. אז עכשיו נכניס בכל שורה תו אחר - התו שידרוס יראה לנו באיזה שורה התווים הדורסים:



בואו נסתכל על מצב ה-SEHChain:



התו '0x44' הוא התו הדורס, זהו התו 'D' ולכן התווים הדורסים נמצאים איפשהו בשורה הרביעית.

בשביל למצוא איפה בדיוק אנו נכניס שלושה שורות של זבל (תווים שאנו יודעים שהם לא הדורסים אז לא משנה מה הם יהיו) ובשורה הרביעית נכניס רצפים של 4 תווים זהים (רצפים שונים, אפילו לפי ה-ABC):

```

C:\Dev-Cpp\SEHExample.exe
Enter number
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAABBBBBCCCCDDDEEEFFFGGGHHHH I I I JJJJJKKKKLLLLMMMMNNNNOOOOPPPQQQRRRRSSSTTTT
    
```

לפי התווים שנמצא שהם דורסים נוכל לדעת בדיוק איפה נמצאים התווים שאנו מחפשים:

Address	SE handler
0022FFE0	4A4A4A4A
49494949	*** CORRUPT ENTRY ***

טוב אז אנו רואים שהמצביע למנהל חריגה הבא נדרס עם התו '0x49' שזה התו 'I'. והתו שדרס את המצביע לפונקציית הטיפול בחריגה הנוכחית הוא '0x4A' שזה התו 'J'.

מה שאומר לנו שבקלט של האקספלויט שלנו אנו נכניס 3*80 תווי A (שזה בעצם שלוש שורות של תווים שלא חשובים לנו), לאחר מכן עוד 4*8 תווי A (שזה שמונה רצפים של רביעיות תווים שלא חשובים לנו), ולאחר מכן את התווים שידרסו את ה-SEHChain.

אבל במה נדרוס אותנו? טוב לכאורה זה אמור להיות ברור - נדרוס אותו בכתובת שבה נחדיר את ה-shellcode שלנו, ככה כאשר התוכנית תקפוץ אל מנהל החריגה היא תגיע אל הקוד המוזרק שלנו.

אבל קיימת בעיה אחת, אנחנו לא יודעים לאן יגיע הקוד המוזרק שלנו בזיכרון (מה שאומר שאנחנו לא יודעים באיזה כתובת זה יהיה). לכן צריך לחשוב על משהו יותר חכם. דבר אחד בטוח, הקוד המוזרק צריך להיות איפשהו בקלט, או לפני הכתובות שאנחנו הולכים לדרוס או אחריהם.

אז קיימת שיטה להתמודד עם הבעיה הזאת, מה שקורה במחסנית כאשר התוכנית קופצת אל המנהל חריגה הנוכחית הוא שהמחשב דוחף לשם (למחסנית) שתי כתובות. הראשונה מבניהם (זאת שתהיה

מתחת לשנייה) היא כתובת המצביע לחריגה הבאה. מבחינת התוכנית, אם אי אפשר לטפל בחריגה הזאת בעזרת המנהל חריגה הנוכחית היא תעביר אותה אל מנהל החריגה הבא, ובשביל זה היא צריכה את הכתובת שלה (שהיא דחפה למחסנית). הרעיון של השיטה הזאת אומר שניקח את הכתובת הזאת ונקפוץ אליה, ושם יחכה לנו הקוד המוזרק. ואז אנחנו שואלים את עצמנו שתי שאלות - איך נצליח לקפוץ אל הכתובת הזאת דרך המנהל חריגה הנוכחי. ושאלה יותר קשה, האם זה אומר שהקוד המוזרק יצטרך להיות רק 4 תווים?

למה 4 תווים? כי אם אנחנו אומרים שהמקום הזה שקופצים אליו הוא המצביע למנהל החריגה הבא מה שאומר שאחרי ה-4 תווים האלה מגיע מנהל החריגה הנוכחי (ואותו כמובן אנחנו דורסים בכתובת כלשהי, זאת אומרת שחשוב לנו ששם לא יהיה ה-shellcode).

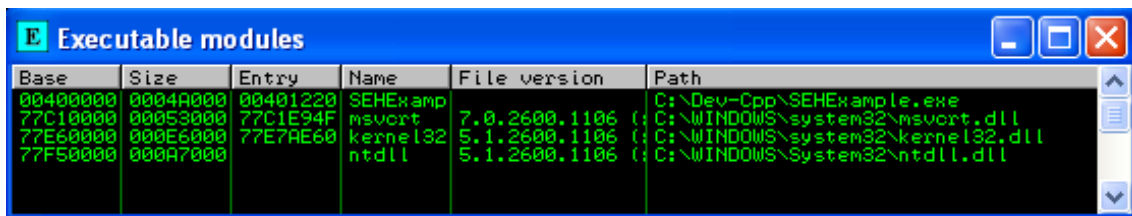
לשאלה הראשונה יש תשובה כזאת - נקח את כל המודולים שהתוכנית טענה בשביל לרוץ (מודולים של מערכת ההפעלה אנחנו נעדיף) ונחפש בהם פיסת קוד שעושה את הדבר הבא:

תוציא כתובת מהמחסנית, תוציא עוד כתובת מהמחסנית, תחזרי אליה.

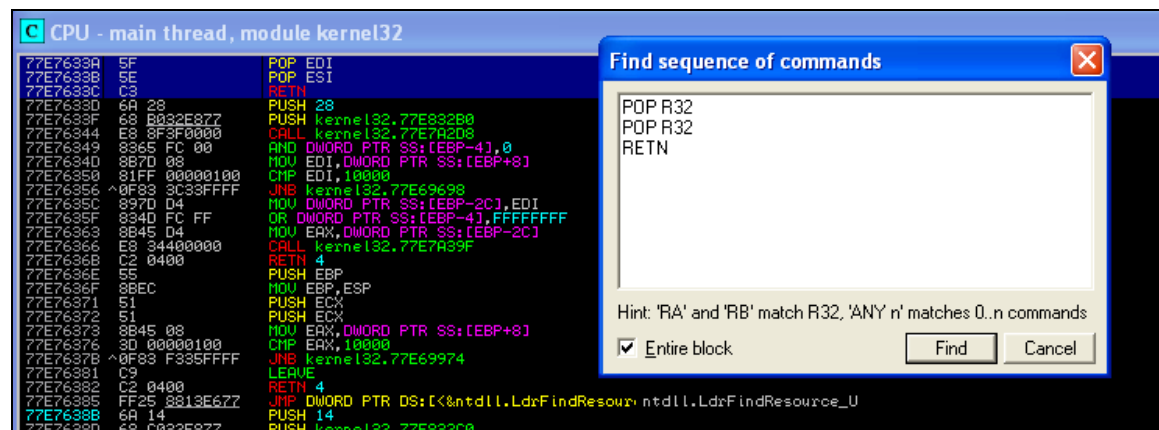
למה? אמרנו שלמחסנית נדחפות שתי כתובות, שהשנייה מהם תיקח אותנו לקוד המוזרק. אז בהוצאה הראשונה לא חשוב לנו הערך שנשלף, בהוצאה השנייה כבר נרצה לקפוץ אל אותו ערך. לכן נחפש רצף פקודות כאלה:

```
POP/POP/RET
```

בואו נראה איך עושים את זה - רשימת המודולים שהתוכנית טענה:



בחרתי את kernel32 ובו אני אחפש את רצף הפקודות:



About Memory And Its Weakness

www.DigitalWhisper.co.il



השורות שמסומנות בכחול הם מה שמצאתי. לכן אני אקח את הכתובת של הפקודה הראשונה שיגיד לתוכנית שלי לקפוץ אליה. (זאת אומרת שאני אדרוס את המצביע למנהל החריגה הנוכחית עם הכתובת הזאת). מה שיקרה זה שהתוכנית תקפוץ לפה, ורצף הפקודות האלה יגרמו לה לקפוץ אל אותו מצביע למנהל החריגה הבא, שם נשים את ה-shellcode שלנו (למרות שיש לנו שם רק 4 תווים פנויים).

איך פותרים את הבעיה של ארבעת התווים? פשוט מאוד, אנחנו צריכים לא לשנות את ארבעת התווים שמגיעים לאחר התווים האלה, נכון? (כי זה הכתובת שתפנה אל POP/POP/RET) אז נרשום את ה-shellcode אחרי הכתובת הזאת, ובארבע התווים שיש לנו נעשה קפיצה אל ה-shellcode.

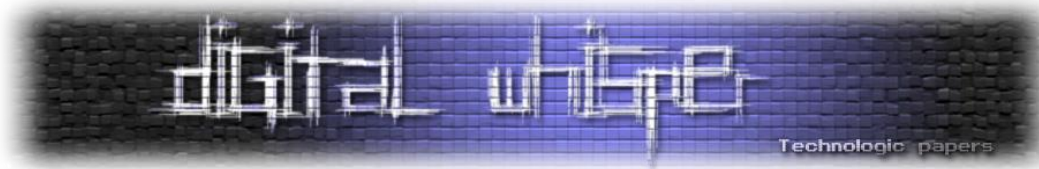
קפיצה אל ה-shellcode נראית ככה: '\0xEB\0x06\0x90\0x90' בגדול ארבעת התווים האלה אומרים דלג מעל הארבע תווים הבאים ומשם תמשיך לבצע את התוכנית.

אז האקספלויט שלנו יראה ככה:

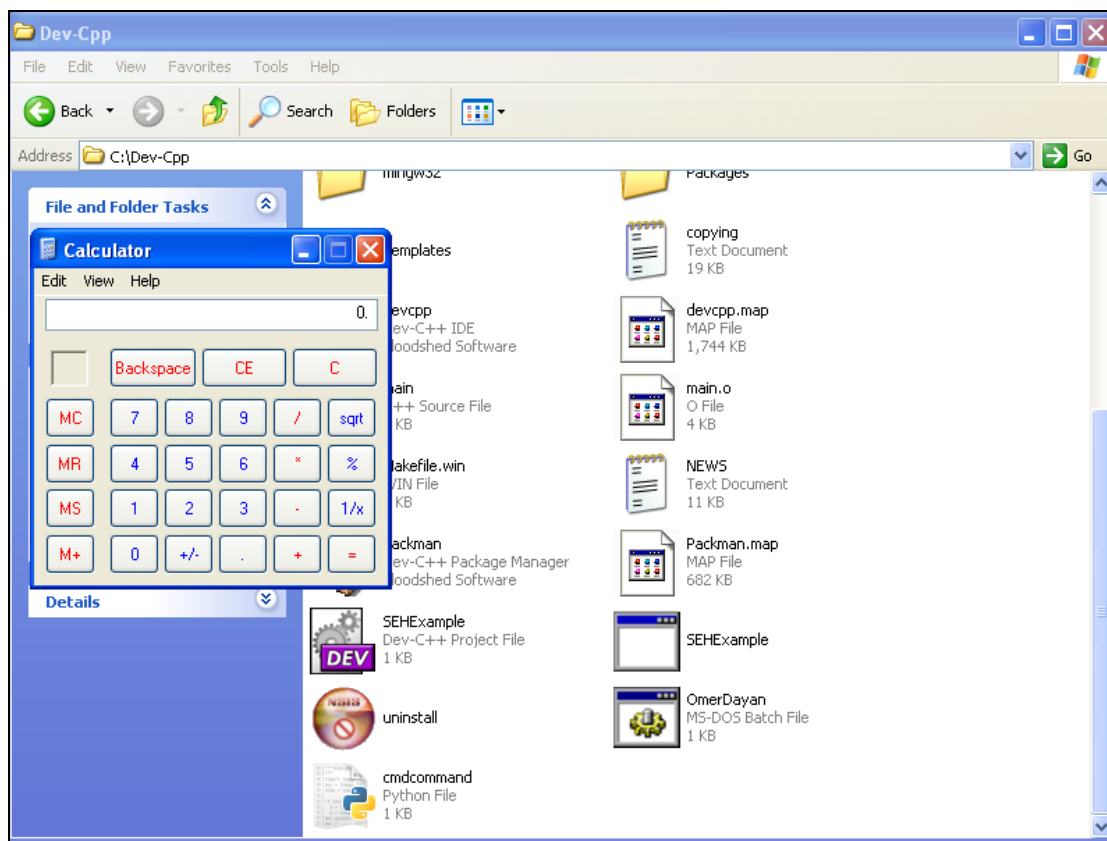
- הרבה תווי A (חישבנו את המספר המדויק מקודם).
- 4 תווים שהם פעולת דילוג על ארבעת התווים הבאים.
- כתובת אליה התוכנית תקפוץ כאשר תיזרק החריגה (שם יחכה לה POP/POP/RET).
- Shellcode.

ה-shellcode שנשתמש בו הוא פשוט פותח מחשבון. לא עושה משהו מיוחד. רק בשביל להראות שאפשר להריץ כל קוד על מחשב הפגיע.

בעמוד הבא מצורף הקוד המלא של האקספלויט.

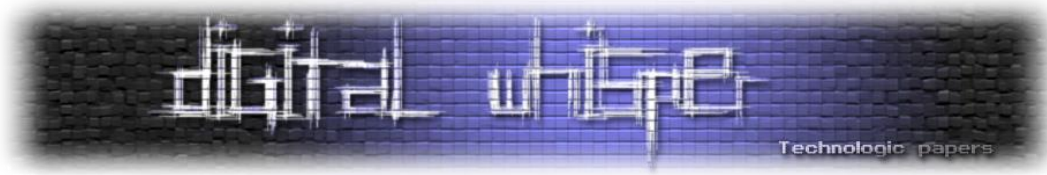


כעת נפעיל אותו ונראה מה קורה:



הצלחנו!

כמובן שאת ה-shellcode שלנו היינו משנים לקוד יותר זדוני במקרה של פעילות אמיתית.



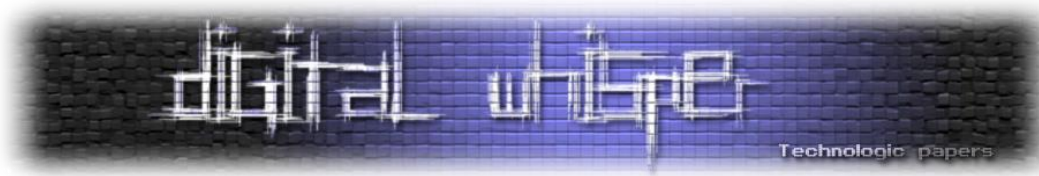
סיכום

במאמר זה דיברנו על נושאים רבים, בהקשר זיכרון של תוכנית ואיך המחשב שלנו מנהל אותה. כמובן מכל נושא אפשר ללמוד בעוד מקומות רבים, יש הרבה איפה להרחיב ולתאר. על כל נושא תמצאו ספרים רבים באינטרנט או בכל מקום אחר.

לכל הדוגמאות היו כמובן הסברים ותיאורים לפני. אך למרות זאת כמובן גם בדוגמאות לא תיארנו את זה באופן מלא. לפי דעתי רק ניסיון אמיתי והתנסות יכולה ללמד את הדברים האלה באמת.

על המחבר

שמי עומר דיין, ואני בן 19 מקרית גת, משרת ביחידה מובחרת בחיל המודיעין. עוסק בפיתוח ואבטחת מידע בעיקר. מתעניין מאוד בטכנולוגיה. DigitalWhisper היה עבורי מקור מידע איכותי ונגיש. למדתי ממנו המון. והחלטתי לתרום לו בחזרה. מקווה שיזדמן לי לכתוב פה עוד מאמרים. עבור כל בקשת עזרה או שאלה או דבר כזה אתם מוזמנים ליצור איתי קשר: omerxdayan@walla.com



Reverse Engineering - לגלות את הסודות החבויים

מאת דוד א.

הקדמה

Reverse Engineering (או הנדסה הפוכה) הינו תחום שהפך למרכזי וחשוב מאוד בתחום המחשבים, מגוון של קבוצות וענפי מחשבים עוסקות בו, מקבוצות של האקרים לאנשי מחשב בתחום ההייטק והצבא. כולם יכולו להעיד על החשיבות הרבה של תחום זה. במאמר הזה אסביר מהו Reverse Engineering, מהם השימושים בו היום ואף אציג בפועל הדגמה של שימוש ב-Reverse Engineering.

אז מה זה Reverse Engineering?

Reverse Engineering הוא תהליך שבו מגלים עקרונות טכנולוגיים של מוצר דרך אופן פעולתו ומבנהו, בתהליך חוקרים באופן קפדני את המוצר ומנסים להשיג כמה שיותר מידע עליו, לעיתים משתמשים בתהליך על מנת להרכיב מוצר חדש שדומה באופן פעולתו למוצר הראשוני שנחקר ולעיתים משתמשים בתהליך על מנת למצוא ולנצל חולשות במוצר הקיים. התהליך הזה אינו חדש לעולם ואינו מוגבל לתחום המחשבים, דוגמא לשימוש בו הוא בזמן מלחמת העולם השנייה כאשר הגרמנים הצליחו להעתיק את מבנה תותח ה-120 מ"מ של הכוחות הרוסים על ידי שימוש ב-Reverse Engineering. אך מאמר זה אינו שיעור היסטוריה ולכן בואו נתעדכן בשימוש של Reverse Engineering בעולם המודרני ובעיקר בתחום המחשבים.

שימושים ב-Reverse Engineering

בתחום המחשבים כיום, נעשית הנדסה הפוכה של תוכנות על ידי שימוש בכלי Debugging - כלים שעוזרים לחקור את פעולות התוכנה בזמן הריצה שלה, הכלים האלה מאפשרים למשתמשים לראות את קוד ה-Assembly של התוכנות שהם רוצים לחקור ולנסות להבין כיצד הן עובדות מאחורי הקלעים.

כפי שצינתי בהתחלה ל-Reverse Engineering יש שימוש במגוון של קבוצות.

Reverse Engineering לגלות את הסודות החבויים -

www.DigitalWhisper.co.il

בתחום הייטק ניתן לראות את השימוש ב-Reverse Engineering בעיקר בתחום ההגנה, חברה שרוצת לשמור על הבטיחות של המוצרים שלה מעסיקה אנשי אבטחה שמשתמשים בכלי Reverse Engineering על מנת לחקור ולגלות חולשות בתוכנה של החברה, לאחר שהם מגלים את החולשות הם מדווחים אותם לחברה ועוזרים לה למנוע מגורמים זרים לנצל את החולשות האלו. בנוסף לכך, בתחום הייטק משתמשים לעיתים בתהליך על מנת להעתיק מאפיינים ואלגוריתמים של תוכנות של חברות מתחרות.

בתחום הצבא ובתחום ההאקינג נעזרים בהנדסה הפוכה על מנת לחקור ולמצוא באגים וחולשות אפשריות בקוד של תוכנה, לאחר שהם נמצאים ניתן להשתמש בהם על מנת לגרום לקריסת התוכנה או במקרים יותר קיצוניים להרצת קוד זדוני על המחשב של בעל התוכנה (ראה [exploits](#)).

לתחום ההאקינג יש עוד שימוש חשוב ב-Reverse Engineering, האקרים שחוקרים תוכנות מצליחים באמצעות Reverse Engineering לשנות את אופן הפעולה של תוכנות על מנת שיתאימו לרצונם. התחום הזה נקרא Software Cracking והוא כולל שינויים לתוכנה כגון: שינוי התוכנה כך שלא תכיל פרסומות או מודעות, שינוי תוכנה מסחרית שדורשת תשלום בכסף, כך שיהיה ניתן להשתמש בכל מאפייני התוכנה בחינם ושינוי תוכנות משחקי מחשב כדי להקנות יתרונות בזמן משחק נגד שחקנים אחרים.

Software Cracking נעשה על ידי שינוי, הוספה או הוצאה של קוד Assembly לתוכנה הרצויה, לאחר השינוי האקרים שומרים קובץ חדש שהוא העתק של הקובץ המקורי עם השינויים הרצויים, קובץ זה נקרא Crack או קראק ובאינטרנט ניתן למצוא מספר רב של קראקים ממגוון סוגים לכל מיני תוכנות.

בחלק הבא של המאמר אני אתמקד בתחום ה-Software Engineering, אני אדגים על תוכנה מסחרית כיצד אני יכול לגרום לכך שהתוכנה תירשם על שמי ותיתן לי להשתמש בכל התכונות שלה מבלי לשלם את התשלום שהיא מחייבת אותי.



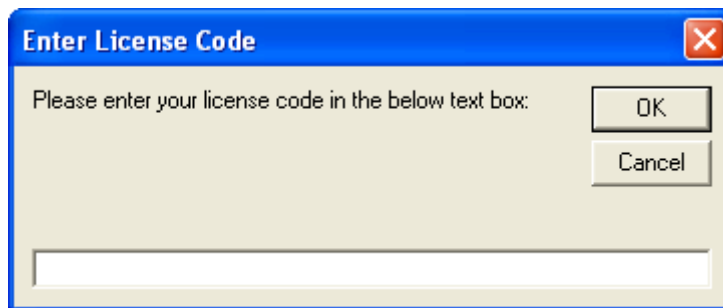
התוכנה היא M4a to Mp3 Converter version 1.20 והכלי debugging שאני אשתמש כדי לחקור את התוכנה ולאפשר את כל התכונות שלה הוא: OllyDbg version 1.1.

שימו לב לבקשת התשלום

נלחץ על ה-About וננסה להירשם על ידי כניסה ל-Enter Code:



כפי שניתן לראות התוכנה מבקשת שנכניס קוד על מנת להירשם.

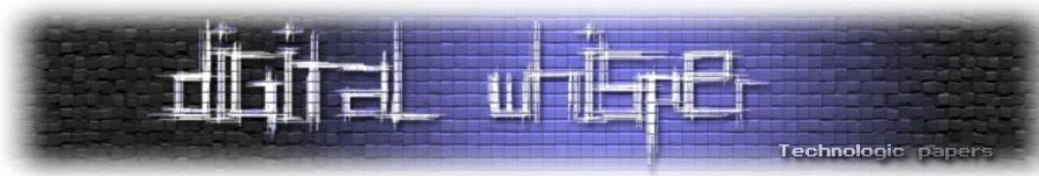


נכניס סתם קוד ונקבל את ההודעה הבאה:



הקוד שהכנסנו לא התקבל בתוכנה.

אז איפשהו בתוכנה מתרחש בדיקה של הקוד שהכנסנו באיזשהו שיטת בדיקה של התוכנה כדי לבדוק אם הוא קוד לגיטימי, אנחנו צריכים לחקור את התוכנה ולגרום לכך שנוכל לעבור את שיטת הבדיקה הזאת כדי שהתוכנה תקבל את הקוד שלנו.



נסתכל על התוכנה ב-OllyDbg ונראה איך היא עובדת. אסביר בקצרה איך OllyDbg מציגה מידע על התוכנה: אנחנו יורדים בקוד בלחיצה על המקש F8 ונכנסים לקריאות פונקציה במקש F7.

הוראות האסמבלר של התוכנה

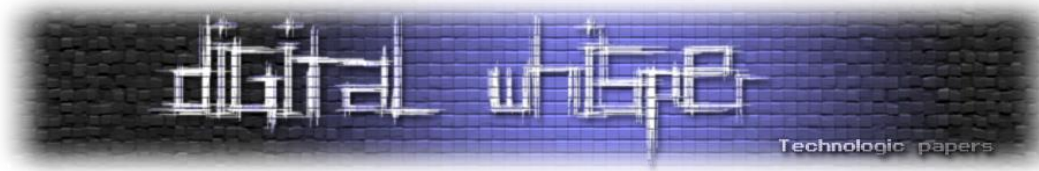
הצגה הערכים של האוגרים של התוכנה

The screenshot shows the OllyDbg interface with the following components:

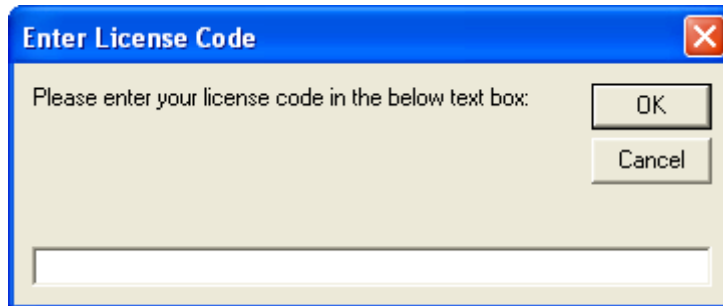
- Assembly View:** Shows assembly instructions such as `JMP DWORD PTR DS:[<&MSUBUM60.#311>]` and `PUSH m4atomp3.0040AC9C`. The instruction at address `00401B94` is highlighted.
- Registers (FPU):** A list of registers including EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, and EIP. EIP is set to `00401B94`.
- Hex Dump:** A table showing memory addresses and their corresponding hex values. The address `0041C000` is highlighted.
- Command Line:** Shows the program entry point.

ערכים שבכתובות זיכרון

ערכים שבתוך המחשנית

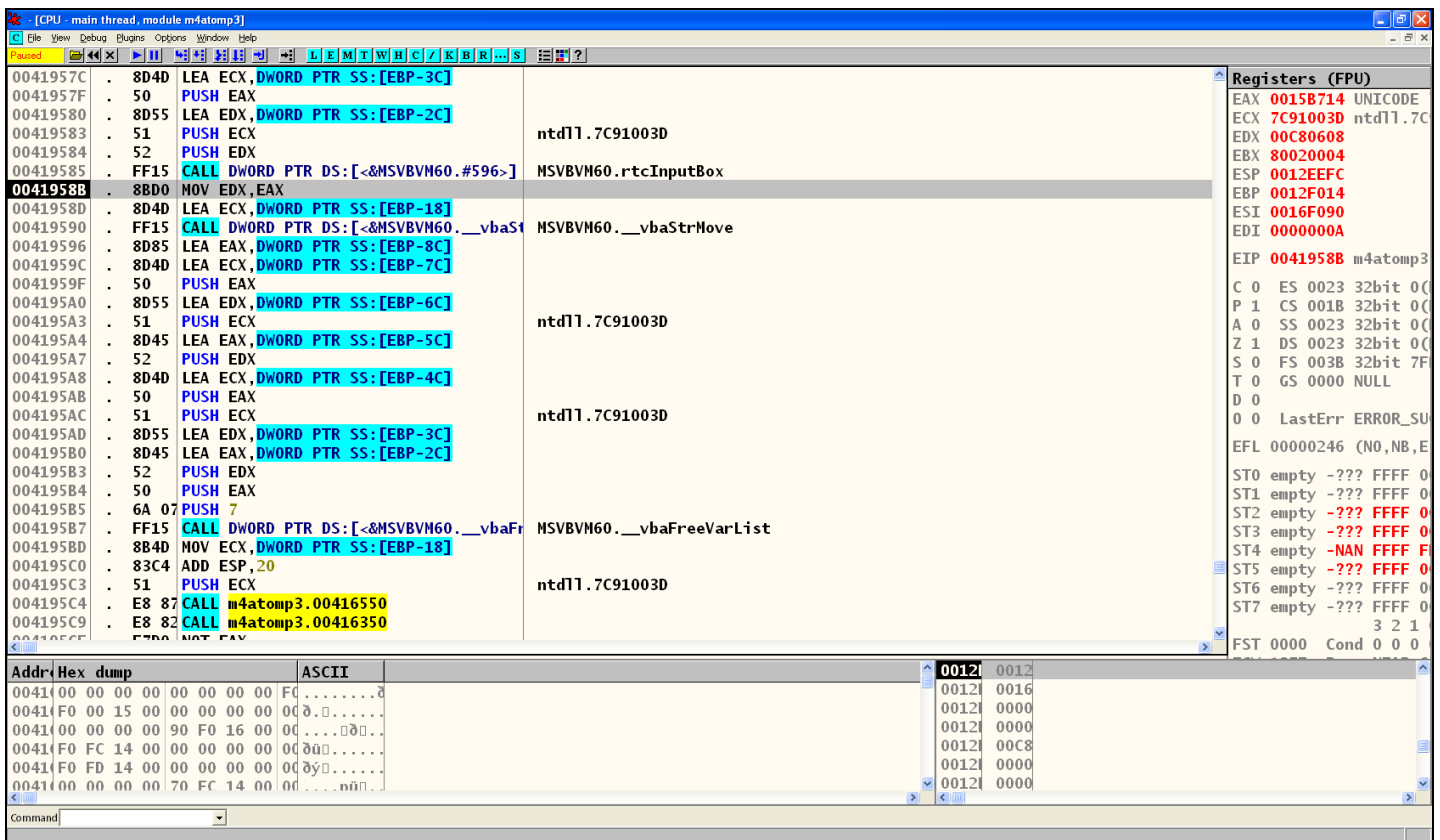


כעת, אחרי שאתם מבינים באופן בסיסי את המבנה של OllyDbg אני אמשיך ואראה מה השלב הבא. ננסה שוב להכניס קוד על מנת לעקוב אחרי התוכנה מקליטת הקוד עד לנקודה בה היא בודקת את תקינות הקוד ומחליטה אם יש לאשר לנו את התוכנה. נפתח את החלון שנית:



ונלחץ על F12 או כפתור העצירה שיעצור את התוכנית שלנו ואז על ALT+F9, על ידי כך שהקשנו ALT+F9 עשינו פעולה ב-OllyDbg שנקראת "Execute till user return" שבעצם עוצרת את ה-debugging תיקח אותנו להוראה הבאה אחרי קליטת הקוד ומשם נמשיך את החקירה שלנו. נלחץ על התוכנה ונכניס את הקוד שלנו שאחריו נעקוב ונראה כיצד בודקים אותו, נשתמש ב-"code".

הגענו לקטע הבא:





נמשיך לרדת למטה ונגיע לקטע שנראה קצת "חשוד":

ההשוואה כאן קובעת האם הקפיצה תתבצע ולכן בוא נחזור על צעדנו ונבין יותר לעומק מה קרה עד ההשוואה וכיצד נקבע הערך של AX

אנחנו לא לוקחים את הקפיצה הזאת ולכן אנחנו מקבלים הודעה של קוד לא תקין כמו בהודעה של התוכנה

The screenshot shows a debugger window with the following assembly instructions:

```

004195C4 . E8 87 CALL m4atomp3.00416550
004195C9 . E8 82 CALL m4atomp3.00416550
004195CE . F7D0 NOT EAX
004195D0 . 66:3D CMP AX,0FFFF
004195D4 . 66:A3 MOV WORD PTR DS:[41C05C],AX
004195DA . 895D MOV DWORD PTR SS:[EBP-54],EBX
004195DD . 897D MOV DWORD PTR SS:[EBP-5C],EDI
004195E0 . 895D MOV DWORD PTR SS:[EBP-44],EBX
004195E3 . 897D MOV DWORD PTR SS:[EBP-4C],EDI
004195E6 . 895D MOV DWORD PTR SS:[EBP-34],EBX
004195E9 . 897D MOV DWORD PTR SS:[EBP-3C],EDI
004195EC . 75 67 JNZ SHORT m4atomp3.00419655
004195EE . 8D95 LEA EDX,DWORD PTR SS:[EBP-9C]
004195F4 . 8D4D LEA ECX,DWORD PTR SS:[EBP-2C]
004195F7 . C785 MOV DWORD PTR SS:[EBP-94],m4atomp3.00416550
00419601 . C785 MOV DWORD PTR SS:[EBP-9C],8
0041960B . FF15 CALL DWORD PTR DS:[<MSVBVM60.__vbaVarMSVBVM60.__vbaVarDup]
00419611 . 8D55 LEA EDX,DWORD PTR SS:[EBP-5C]
00419614 . 8D45 LEA EAX,DWORD PTR SS:[EBP-4C]
00419617 . 52 PUSH EDX
00419618 . 8D4D LEA ECX,DWORD PTR SS:[EBP-3C]
0041961B . 50 PUSH EAX
0041961C . 51 PUSH ECX
0041961D . 8D55 LEA EDX,DWORD PTR SS:[EBP-2C]
00419620 . 6A 10 PUSH 10
00419622 . 52 PUSH EDX
00419623 . FF15 CALL DWORD PTR DS:[<MSVBVM60.#595>]
00419629 . 8D45 LEA EAX,DWORD PTR SS:[EBP-5C]
0041962C . 8D4D LEA ECX,DWORD PTR SS:[EBP-4C]
0041962F . 50 PUSH EAX
00419630 . 8D55 LEA EDX,DWORD PTR SS:[EBP-2C]
  
```

The registers window shows the following values:

```

Registers (FPU)
EAX FFFFFFFF
ECX 0012F000
EDX 0014BC68
EBX 80020004
ESP 0012EEFC
EBP 0012F014
ESI 0016F090
EDI 0000000A
EIP 004195EC m4atomp3.004195
  
```

The message box text is: "Unicode \"Sorry, the registration code is not correct! Please try i...\""

ECX נדחף כפרמטר לשתי קריאות שאחריו

הקוד שהאזנו נימצא ב-ECX

The screenshot shows a debugger window with the following assembly instructions:

```

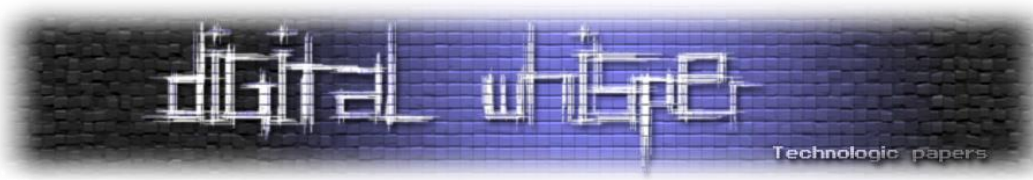
0041958B . 8B00 MOV EDI,ESI
0041958D . 8D4D LEA ECX,DWORD PTR SS:[EBP-18]
00419590 . FF15 CALL DWORD PTR DS:[<&MSVBVM60.__vbaStrMove]
00419596 . 8D85 LEA EAX,DWORD PTR SS:[EBP-8C]
0041959C . 8D4D LEA ECX,DWORD PTR SS:[EBP-7C]
0041959F . 50 PUSH EAX
004195A0 . 8D55 LEA EDX,DWORD PTR SS:[EBP-6C]
004195A3 . 51 PUSH ECX
004195A4 . 8D45 LEA EAX,DWORD PTR SS:[EBP-5C]
004195A7 . 52 PUSH EDX
004195A8 . 8D4D LEA ECX,DWORD PTR SS:[EBP-4C]
004195AB . 50 PUSH EAX
004195AC . 51 PUSH ECX
004195AD . 8D55 LEA EDX,DWORD PTR SS:[EBP-3C]
004195B0 . 8D45 LEA EAX,DWORD PTR SS:[EBP-2C]
004195B3 . 52 PUSH EDX
004195B4 . 50 PUSH EAX
004195B5 . 6A 07 PUSH 7
004195B7 . FF15 CALL DWORD PTR DS:[<&MSVBVM60.__vbaFreeVarList]
004195BD . 8B4D MOV ECX,DWORD PTR SS:[EBP-18]
004195C0 . 83C4 ADD ESP,20
004195C3 . 51 PUSH ECX
004195C4 . E8 87 CALL m4atomp3.00416550
004195C9 . E8 82 CALL m4atomp3.00416350
004195CE . F7D0 NOT EAX
004195D0 . 66:3D CMP AX,0FFFF
004195D4 . 66:A3 MOV WORD PTR DS:[41C05C],AX
004195DA . 895D MOV DWORD PTR SS:[EBP-54],EBX
004195DD . 897D MOV DWORD PTR SS:[EBP-5C],EDI
004195E0 . 895D MOV DWORD PTR SS:[EBP-44],EBX
004195E3 . 897D MOV DWORD PTR SS:[EBP-4C],EDI
    
```

The registers window shows the following state:

```

Registers (FPU)
EAX 0000000A
ECX 0016D094 UNICODE "code"
EDX 00000003
EBX 80020004
ESP 0012EEFC
EBP 0012F014
ESI 0016F090
EDI 0000000A
EIP 004195C3 m4atomp3.004195C3
    
```

שתי הקריאות האלו לוקחות את ECX שמכיל את הקוד שלנו כפרמטר ומשנות את הערך של EAX ולכן גם משנות את AX שקובע את הקפיצה וגורם לנו להגיע להודעה של קוד שגוי.
כנראה שבהם מתבצע בדיקות תקינות הקוד.



כנס לכל אחת מהקריאות ונראה את התוכן שלהם. בקריאה הראשונה אין עדיין בדיקה של הקוד שלנו ויש בעיקר פעולות שקשורות לקבצים.

כנראה שהקריאות
התייחסו לקובץ הזה

ניתן לראות קריאות שקשורות לקבצים

The screenshot shows a debugger window with the following assembly instructions:

```

004165FA . 68 24 PUSH m4atomp3.0040F224
004165FF . 52 PUSH EDX
00416600 . E8 0B CALL m4atomp3.00416E10
00416605 . 8BD0 MOV EDX,EAX
00416607 . 8D4D LEA ECX,DWORD PTR SS:[EBP-18]
0041660A . FF15 CALL DWORD PTR DS:[&MSVBVM60.__vbaStrMove]
00416610 . 8D4D LEA ECX,DWORD PTR SS:[EBP-20]
00416613 . FF15 CALL DWORD PTR DS:[&MSVBVM60.__vbaFreeStr]
00416619 . 8D4D LEA ECX,DWORD PTR SS:[EBP-24]
0041661C . FF15 CALL DWORD PTR DS:[&MSVBVM60.__vbaFreeObj]
00416622 . 8D45 LEA EAX,DWORD PTR SS:[EBP-34]
00416625 . C745 MOV DWORD PTR SS:[EBP-2C],80020004
0041662C . 50 PUSH EAX
0041662D . C745 MOV DWORD PTR SS:[EBP-34],0A
00416634 . FF15 CALL DWORD PTR DS:[&MSVBVM60.#648<]
0041663A . 8D4D LEA ECX,DWORD PTR SS:[EBP-34]
0041663D . 8BF0 MOV ESI,EAX
0041663F . FF15 CALL DWORD PTR DS:[&MSVBVM60.__vbaFreeVar]
00416645 . 8B4D MOV ECX,DWORD PTR SS:[EBP-18]
00416648 . 51 PUSH ECX
00416649 . 56 PUSH ESI
0041664A . 6A FF PUSH -1
0041664C . 6A 02 PUSH 2
0041664E . FF15 CALL DWORD PTR DS:[&MSVBVM60.__vbaFileOpen]
00416654 . 8B55 MOV EDX,DWORD PTR SS:[EBP-14]
00416657 . 52 PUSH EDX
00416658 . 56 PUSH ESI
00416659 . 68 38 PUSH m4atomp3.0040F238
0041665E . FF15 CALL DWORD PTR DS:[&MSVBVM60.__vbaPrintFile]
00416664 . 83C4 ADD ESP,0C

```

The registers window shows the following values:

```

Registers (FPU)
0000000A
001588B4 UNICODE "C:\Program Files\
00000003
80020004
0012EE8C
0012EEF0
00150001
00000000
00416648 m4atomp3.00416648
ES 0023 32bit 0(FFFFFFFF)
CS 001B 32bit 0(FFFFFFFF)
SS 0023 32bit 0(FFFFFFFF)
DS 0023 32bit 0(FFFFFFFF)
FS 003B 32bit 7FFDD000(FFF)
GS 0000 NULL
LastErr ERROR_SUCCESS (00000000)
00000246 (NO,NB,E,BE,NS,PE,GE,LE)
empty -??? FFFF 00FF00FF 00FF00FF
empty -??? FFFF 00FF00FF 00FF00FF
empty -??? FFFF 00FE00E8 004A0011
empty -??? FFFF 00FE00E7 0044000B
empty -NAN FFFF FFE8440B FFE94A11
empty -??? FFFF 00FF00E8 0044000B
empty -??? FFFF 00000000 00000000
empty -??? FFFF 00800080 00800080
0000 Cond 0 0 0 0 Err 0 0 0 0 0 0

```

מבדיקה מהירה ניתן לראות שהקובץ lic.dat הוא קובץ שנמצא בתיקיית התוכנה ואם נפתח אותו ב- Notepad נגלה שהוא מכיל את הקוד שהכנסנו, נכון לעכשיו זה לא כזה שימושי בשבילנו ולכן נעבור לקריאה הבאה.



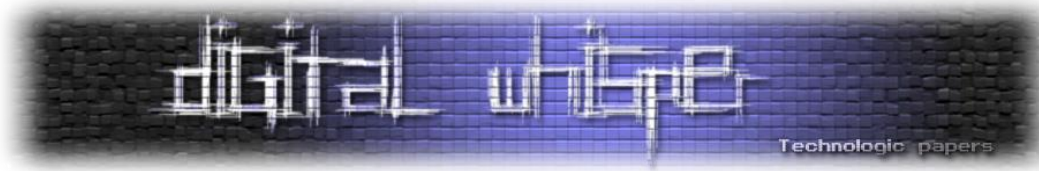
נעבור לקריאה הבאה:

אם נסתכל על הפרמטרים של הפונקציה לא נמצא זכר לקוד שהאזנו אלא רק השם של lic.dat ועוד כמה ערכים.

הקריאה הזאת נראית חשודה, היא קריאה ל-module של vbaStrCmp, המערכת הפעלה עם פונקציה שנקראת vbaStrCmp, מחיפוש באינטרנט על הפונקציה הזאת יתגלה לנו שזו פונקציה שמקבלת שתי ערכי String, משוואה ביניהם ומחזירה 0 אם הם שווים ו-1 או -1 אם הם לא שווים.

The screenshot shows a debugger window with the following components:

- Assembly Window:** Shows instructions starting from address 00416434. Key instructions include:
 - 00416441: `PUSH EAX` (highlighted)
 - 00416442: `PUSH m4atomp3.0040E0A0` (highlighted)
 - 00416443: `CALL DWORD PTR DS:[&MSVBVM60.__vbaStrCmp]` (highlighted)
 - 00416444: `MOV ESI, EAX`
 - 00416445: `LEA ECX, DWORD PTR SS:[EBP-24]`
 - 00416446: `NEG ESI`
 - 00416447: `SBB ESI, ESI`
 - 00416448: `NEG ESI`
 - 00416449: `NEG ESI`
 - 0041644A: `CALL EBX`
 - 0041644B: `TEST SI, SI`
 - 0041644C: `JE m4atomp3.004164F1` (highlighted)
 - 0041644D: `LEA EDX, DWORD PTR SS:[EBP-3C]`
 - 0041644E: `MOV DWORD PTR SS:[EBP-34], 80020004`
 - 0041644F: `PUSH EDX`
 - 00416450: `MOV DWORD PTR SS:[EBP-3C], 0A`
 - 00416451: `CALL DWORD PTR DS:[&MSVBVM60.rtcFreeFile]`
 - 00416452: `LEA ECX, DWORD PTR SS:[EBP-3C]`
 - 00416453: `MOV ESI, EAX`
 - 00416454: `CALL DWORD PTR DS:[&MSVBVM60.__vbaFreeVar]`
 - 00416455: `MOV EAX, DWORD PTR SS:[EBP-1C]`
 - 00416456: `PUSH EAX`
 - 00416457: `PUSH ESI`
 - 00416458: `PUSH -1`
 - 00416459: `PUSH 1`
- Registers (FPU) Window:** Shows register values:
 - EAX: 00176E34 (Unicode "lic.dat")
 - ECX: 0012ED38
 - EDX: 00176E34 (Unicode "lic.dat")
 - EBX: 73506A30 (MSVBVM60.__vbaFreeStr)
 - ESP: 0012ECF0
 - EBP: 0012ED5C
 - ESI: 00CA5DBC
 - EDI: 73506A74 (MSVBVM60.__vbaStrMove)
 - EIP: 00416441 (m4atomp3.00416441)
- Call Stack Window:** Shows the current function call:
 - 00416441: m4atomp3.00416441
 - 00416443: MSVBVM60.__vbaStrCmp
 - 0041644A: MSVBVM60.__vbaFreeStr
 - 00416451: MSVBVM60.rtcFreeFile
 - 00416454: MSVBVM60.__vbaFreeVar
- Memory Dump Window:** Shows hex dump and ASCII for address 00416441.



בקיצור, בקטע הזה אין את תהליך הבדיקה של הקוד שלנו לכן נמשיך הלאה. נרד עוד למטה ונגיע לקטע מעניין:

שימו לב ש-EAX קיבל ערך חדש שנראה מאוד כמו Cd key של תוכנה. בדרך כלל תוכנות בערכים כאלה באלגוריתמים של בדיקה והשוואה עם הקוד שהשתמש מביא לתוכנה כדי לבדוק אם הוא קוד לגיטימי.

The screenshot shows a debugger window with the following components:

- Assembly Window:** Shows instructions at addresses 0041648F to 004164D0. Key instructions include:
 - 0041648F: 6A 01 PUSH 1
 - 00416491: FF15 CALL DWORD PTR DS:[&MSVBVM60.__vbaFileOpen]
 - 00416497: 8D4D LEA ECX, DWORD PTR SS:[EBP-18]
 - 0041649A: 56 PUSH ESI
 - 0041649B: 51 PUSH ECX
 - 0041649C: FF15 CALL DWORD PTR DS:[&MSVBVM60.__vbaLineInputStr]
 - 004164A2: 56 PUSH ESI
 - 004164A3: FF15 CALL DWORD PTR DS:[&MSVBVM60.__vbaFileClose]
 - 004164A9: 8B55 MOV EDX, DWORD PTR SS:[EBP-18]
 - 004164AC: 52 PUSH EDX
 - 004164AD: E8 0E CALL m4atomp3.004166C0
 - 004164B2: 8BD0 MOV EDX, EAX
 - 004164B4: 8D4D LEA ECX, DWORD PTR SS:[EBP-24]
 - 004164B7: FFD7 CALL EDI
 - 004164B9: 50 PUSH EAX
 - 004164BA: E8 31 CALL m4atomp3.004167F0
 - 004164BF: 8BD0 MOV EDX, EAX
 - 004164C1: 8D4D LEA ECX, DWORD PTR SS:[EBP-28]
 - 004164C4: FFD7 CALL EDI
 - 004164C6: 50 PUSH EAX
 - 004164C7: FF15 CALL DWORD PTR DS:[&MSVBVM60.__vbaStrCmp]
 - 004164CD: F7D8 NEG EAX
 - 004164CF: 1BC0 SBB EAX, EAX
 - 004164D1: 8D4D LEA ECX, DWORD PTR SS:[EBP-24]
 - 004164D4: 40 INC EAX
 - 004164D5: F7D8 NEG EAX
 - 004164D7: 8945 MOV DWORD PTR SS:[EBP-14], EAX
 - 004164DA: 8D45 LEA EAX, DWORD PTR SS:[EBP-28]
 - 004164DD: 90 RETN
- Registers (FPU) Window:** Shows EAX=001750DC (UNICODE "L8HKW23ERAJU8X5YGW4X5MA"), EDX=00140608, and EIP=004164BF.
- Memory Dump Window:** Shows a hex dump of memory at address 004164D0, with ASCII values like "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000", "00000000".

הנה עוד דבר חשוב, הפעולה שמשוואה בין שתי ערכים מקבלת את ה-cd key שראינו כפרמטר.

כנס לתוך הקריאה של פונקציה ההשוואה ועכשיו הכל ברור:

הערך של הקוד שלנו
שנשלח כפרמטר

הערך של ה-key cd
שנשלח כפרמטר

The screenshot shows a debugger window with the following components:

- Assembly List:**
 - 735093DA FF7424 PUSH DWORD PTR SS:[ESP+8]
 - 735093DE FF7424 PUSH DWORD PTR SS:[ESP+8]
 - 735093E2 6A 00 PUSH 0
 - 735093E4 E8 44E6 CALL <MSVBVM60.Compare Function>
 - 735093E9 C2 0800 RETN 8
 - 735093EC FF7424 PUSH DWORD PTR SS:[ESP+8]
 - 735093F0 FF7424 PUSH DWORD PTR SS:[ESP+8]
 - 735093F4 6A 01 PUSH 1
 - 735093F6 E8 32E6 CALL <MSVBVM60.Compare Function>
 - 735093FB C2 0800 RETN 8
 - 735093FE FF7424 PUSH DWORD PTR SS:[ESP+8]
 - 73509402 FF7424 PUSH DWORD PTR SS:[ESP+8]
 - 73509406 6A 00 PUSH 0
 - 73509408 E8 41EE CALL MSVBVM60.7350824E
 - 7350940D 0FBFC0 MOVX EAX,AX
 - 73509410 C2 0800 RETN 8
 - 73509413 FF7424 PUSH DWORD PTR SS:[ESP+8]
 - 73509417 FF7424 PUSH DWORD PTR SS:[ESP+8]
 - 7350941B 6A 01 PUSH 1
 - 7350941D E8 2CEE CALL MSVBVM60.7350824E
 - 73509422 0FBFC0 MOVX EAX,AX
 - 73509425 C2 0800 RETN 8
 - 73509428 FF7424 PUSH DWORD PTR SS:[ESP+C]
 - 7350942C FF7424 PUSH DWORD PTR SS:[ESP+C]
 - 73509430 6A 00 PUSH 0
 - 73509432 FF7424 PUSH DWORD PTR SS:[ESP+10]
 - 73509436 E8 3FF9 CALL MSVBVM60.73508D7A
 - 7350943B 8B4424 MOV EAX,DWORD PTR SS:[ESP+4]
- Registers (FPU):**
 - EIP 735093DA MSVBVM60.__vbaStrCmp
 - ESP 0012EE7C
- Stack:**
 - SS:[0012EE84]=0015B74C, (UNICODE "CODE")
- Registers (General Purpose):**
 - EAX 001750DC UNICODE "L8HKMW23ERAJU8X5YGWK4X5MA"
 - ECX 0012EECC
 - EDX 001750DC UNICODE "L8HKMW23ERAJU8X5YGWK4X5MA"
 - EBX 73506A30 MSVBVM60.__vbaFreeStr
 - ESI 00150001
 - EDI 73506A74 MSVBVM60.__vbaStrMove
 - EIP 735093DA MSVBVM60.__vbaStrCmp

הפונקציה משווה בין שתי הפרמטרים ומחזירה 0 אם השווים או 1 או -1 אם הם לא

כאן התוכנה מקבלת את הקוד שהאזנו ומשווה אותו עם ה-cd key, מסתבר שה-cd key הוא לא סתם ערך שהתוכנה משתמשת באלגוריתמיות הבדיקה שלה, אלא הוא למעשה הקוד הלגיטימי של התוכנה והתוכנה פשוט משווה אותה עם הקוד שהאזנו כדי לבדוק אם הם שווים.

לאחר שגילינו את המידע על הקוד של התוכנה נוכל פשוט לרשום את הקוד הזה כאשר התוכנה מבקשת קוד לגיטימי והתוכנה תהיה רשומה. נקבל את ההודעה הבאה:

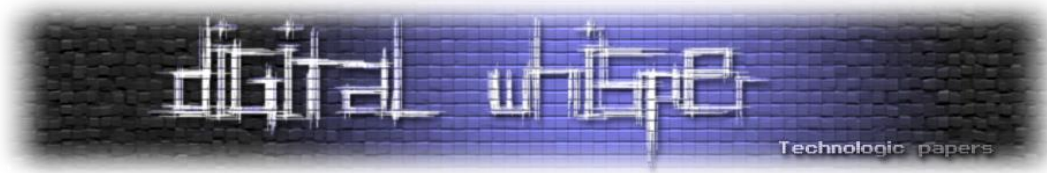


המידע על התוכנה ב-About לפני הרישום:



המידע לאחר הרישום:





לסיכום

הנדסה הפוכה היא הוכחה שאין הגנות שלא ניתן לעבור בתחום המחשבים, אמנם במאמר הזה השתמשתי בדוגמא יחסית קלה זה עדיין מראה לנו על היכולת של הנדסה הפוכה בגילוי סודות של תוכנה ותדמיינו את היכולת הזאת ביד ארגון או ממשלה.

מקווה שנהנתם והתעניינתם!

על המחבר

דוד א. הוא תלמיד תיכון אשר שנתיים מתעסק בתחום ה-Reverse Engineering, מתעניין מאוד במחשבים ובזמנו החופשי הוא אוהב לפתח תוכנות ולחקור על נושאים שונים מתחום הסייבר. אתם מוזמנים לפנות אלי בשאלות ובחוות דעת.

ניתן ליצור איתי קשר באימייל: davidalias17@gmail.com

מקורות וקישורים

- מדריכי הנדסה הפוכה של Lena151:

<https://tuts4you.com/download.php?list.17>

- OllyDbg 1.1 להורדה:

<http://www.ollydbg.de/download.htm>

- התוכנה שפרצנו:

<http://www.m4a-to-mp3-converter.com/>

אבטחת קוד פתוח ע"י ניתוח קוד סטטי

מאת ראג'דיפ קאנאבארה (Raghudeep Kannavara) - Security Center of Excellence, חברת אינטל

[תורגם ע"י אפיק קסטיאל / cp77fk4r]

הקדמה

ניתוח קוד סטטי (Static Code Analysis - SCA) היא שיטת ניתוח קוד המתבצעת כך שהקוד עליו מתבצע המחקר אינו מורץ (Execute), ובדרך כלל - ע"י כלי אוטומטי. כיום, ניתן לראות בחברות רבות, כי שימוש בניתוח מסוג זה, הפך לחלק אינטגרלי מתהליך הפיתוח של התוכנה וכאחד הכלים המרכזיים לזיהוי שגיאות פיתוח בשלביו הראשונים של המוצר.

למרות ששימוש בכלי SCA נעשה באופן קבוע בעת פיתוח תוכנות מסחריות לטובת ביצוע בקרה על איכות התוכנה, שימוש בכלים אלו בשדה הנרחב של תוכנות מבוססות קוד-פתוח (Open Source), מהווה אתגר מעניין ולא פשוט, בייחוד במקרים בהן הקוד מוצא דרכו אל תוך הקוד המסחרי.

לאחרונה נעשו לא מעט מאמצים בשדה זה, במאמר זה אציג דרכים להתמודד עם הבעיה הנ"ל ע"י החלת שיטה זו על פרויקטי קוד-פתוח כגון הקרנל של לינוקס, בדיון אודות תוצאות ניתוח המבוססות על ניתוח כמו שנציג במהלך המאמר, נציע זרימת עבודה אלטרנטיבית לזו המוכרת כיום, אשר ניתן יהיה לאמץ אותה בעת השימוש בקוד פתוח בעת פיתוח תוכנה מסחרית. בנוסף, במהלך המאמר נדון על ההיתרונות והאתגרים העומדים בעת אימוץ זרימת-העבודה מסוג זה.

חשיבותו של מחקר זה

ניתוח קוד סטטי אינה טכנולוגיה חדשה, אך לאחרונה היא מקבלת לא מעט יחס ונראה כי השימוש בה הולך וגובר, ונראה שכיום היא חלק בלתי נפרד ממחזור החיים של פיתוח מוצרים רבים במטרה לשיפור איכות, אמינות והאבטחה של התוכנה. ברוב הארגונים המפתחים תוכנה קניינית, קיים צוות יעודי של אנשי מקצוע שכל מטרת עבודתם היא לדאוג לאיכות הקוד. בצוות זה, נראה לא פעם כי בין שאר הפעולות שמתבצעות, מוצרים כליים אוטומטיים לניתוח סטטי, הן במודלי פיתוח Agile והן במודלי פיתוח קלאסיים יותר.

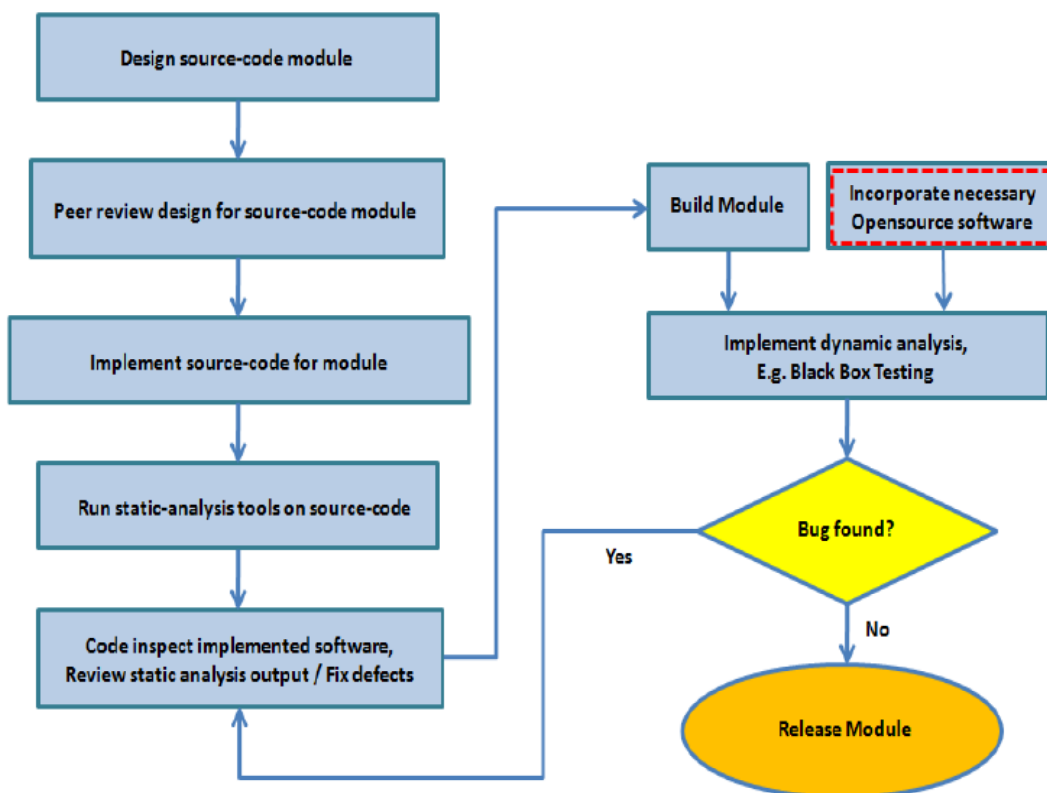
אבטחת קוד פתוח ע"י ניתוח קוד סטטי

www.DigitalWhisper.co.il

כלי SCA מסוגלים לבצע ניתוח קוד לתוכנות אשר פותחו בשפות תכנות שונות ותחת סביבות פיתוח שונות כגון JAVA, .NET, C++/C ועוד. בין אם מדובר בפיתוח מערכת שלמה, או תוכנה קטנה שנועדה לבצע משימה קריטית ותוטמע על Firmware מסויים, שיטת הניתוח הסטטית הוכחה את עצמה כחלק יעיל בעת איתור כשלי קוד ומניעת באגים בתוכנה עוד בשלבים המוקדמים של הפיתוח, והיא ייצבה עצמה כחלק קריטי להצלחה הכוללת של פרויקט התוכנה.

הדעה הרווחת כיום היא כי קוד פתוח הוא קוד שנעשה בו שימוש רב ונבדק על ידי משתמשיו הרבים, בהן נכללים גופי אקדמיה שונים, חברות מסחריות ומשתמשים פרטיים, ובנוסף - ברור שיש אינטרסים מסחריים בעת בהחלת כלי ניתוח סטטי יקרים על איזורי קוד או תוכנות עם סיכויים נמוכים לכשל. וכך שנוצר מצב שנוצרים "מרחבי קוד" רבים שלא נחקרים או לא נחקרו מעולם.

לאחרונה, חברות המפתחות כלי SCA, הרימו את הכפפה בכיוון הזה, אך הקצב בהן גרסאות של תוכנות מבוססות קוד פתוח מעמידה מכשול למאמצים אלו. אך למרות זאת, עקב משברים פיננסיים או אי-תקצוב נכון של פרוייקטים גורמים לכך שתוכנות מבוססות קוד פתוח נכנסות לשימוש מסחרי, כדוגמאת פרוייקטים כגון OpenSSL, Linux, Apache, MySQL ועוד.



[תרשים 1 - תהליך פיתוח "סטנדרטי" הכולל את שלב ניתוח הקוד הסטטי]



בתרשים 1 ניתן לראות זרימת עבודה סטנדרטית כאשר הדו"חות של כלי ה-SCA הינם חלק מתהליך הפיתוח.

למרות שברוב חברות הפיתוח, ניתן לראות כי תהליכי הבדיקה מתבצעים בדרך כלל בעזרת בדיקות דינמיות (כגון בדיקות Black-Box או בדיקות תחת Fuzzer-ים שונים), ופיתוח של קוד חדש נבדק תחת בדיקות סטטיות, נראה כי הטמעה של מרכיבים מבוססי קוד-פתוח אינה שלב שנעשה בו הקפדה יתרה כמו בשאר התהליכים. נראה כי עובדה זו נגרמת עקב ההנחה כי תוכנה המבוססת קוד-פתוח הינה מאובטת יותר, ומכילה פחות באגים מאשר תוכנה המבוססת על קוד-סגור. למרות שההנחה הנ"ל הכן עומדת במבחן הזמן, עדיין ראינו לנכון כי יהיה מעניין לבצע ניתוח קוד סטטי על מספר פרוייקטים מבוססי קוד-פתוח ולנתח את התוצאות.

למרות שכבר קיימים לא מעט ניסיונות קודמים לביצוע ניתוח קוד סטטי על פרוייקטי קוד-פתוח באמצעות כלי SCA, אנו נרצה לבדוק, במאמר זה, האם ניתן למצוא כשלי קוד בשלבים מוקדמים של זרימת העבודה על ידי כלי SCA. יתר על כן, אנו נציע זרימת עבודה חלופית שניתן לאמץ כאשר מטמיעים קוד-פתוח בתהליך הפיתוח המסחרי. פרוייקט שכזה יהיה מעניין הן לקהילת הנדסת התוכנה והן לקהילת הקוד-הפתוח.

לטובת הניסוי, נבחר בכלי בשם [Klocwork Insight](#), מאחר והוא אחד מכלי ה-SCA המובילים כיום בעולם ניתוח הקוד הסטטי. הפרוייקט אותו נחקור יהיה פרוייקט מבוססת קוד-פתוח פופולארי כגון הקרנל של לינוקס. במהלך המחקר נריץ את Klocwork Insight כנגד הקוד של הקרנל של לינוקס ונצפה בתוצאות של הניתוח. במקרה שלנו Klocwork והקרנל של לינוקס הן רק "נציגים", יכולנו לבחור כל כלי SCA אחר וכל פרוייקט מבוסס קוד-פתוח אחר.

קצת על הקרנל של לינוקס

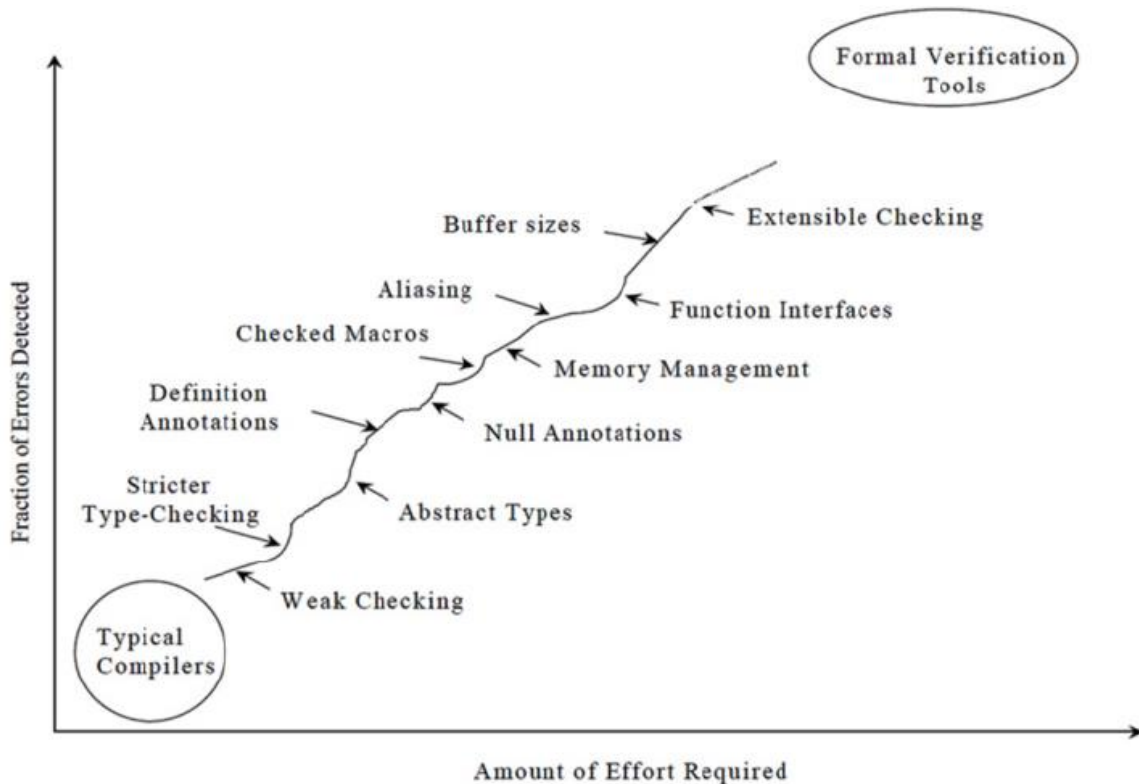
הקרנל של לינוקס הוא ליבת מערכות ההפעלה ממשפחת לינוקס. מדובר באחד הפרוייקטים הבולטים ביותר בתור דוגמא לחופש ולקוד-פתוח, ולכן בחרנו דווקא בפרוייקט זה. הקרנל של לינוקס שוחרר תחת הגרסה השנייה של הרישיון GNU General Public License (GPLv2) והיא פותחה (וממשיכה להתפתח) ע"י תורמים רבים מכלל רחבי העולם. מתקיימים דיונים מדי יום אודות הפיתוח של הקרנל ברשימת התפוצה של הפרוייקט, ושורותיו נכתבו על ידי אלפי מתכנים. פורסמו הפצות לינוקס רבות המבוססות על פרוייקט זה¹, ולשם המאמר בחרנו בגרסאתו ה-2.6.32.9 של הפרוייקט, ששוחררה בפברואר, בשנת 2010.

¹ Linux Kernel, <http://www.kernel.org/pub/linux/kernel/v2.6/>

קצת על ניתוח קוד סטטי

היתרון הייחודי של ניתוח קוד סטטי, הוא היכולת לסרוק את הקוד בשלמותו על מנת לאתר כשלים לוגים וכשלי אבטחה. בדיקה מסוג זה מקיפה מספר מונים מבדיקות כגון Black-Box, עם זאת, קיימים לא מעט כשלים וכשלי אבטחה בעיקר שלא ניתן לזהות על ידי סריקה סטטית ועל מנת לאתרם ישנו הצורך לבצע הרצה של הקוד. כך שניתוח קוד סטטי אינו יכול לבוא לבד, אלא כחלק משלים לסט בדיקות שונות.

בתרשים 2 ניתן לראות את את העקומה המייצגת את ה"עלות מול תועלת" המתקבלת בעת שימוש בבדיקות SCA:



[תרשים 2 - עלות מול תועלת טיפוסית בעת בדיקות SCA]²

ניתן לראות כי כאשר משתמשים בבדיקות מסוג זה, בדיקות רבות יותר נאכפות והחלק היחסי של השגיאות ובעיות שהתגלו עולה ביחד עם כמות המאמץ הנדרש על מנת לאכוף בדיקות אלו. דוגמא למינימום מאמץ שמופעל הוא שימוש קומפיילרים טיפוסיים כגון GCC אשר מבצעים מספר ניתוחים סטטיים על הקוד על מנת להציף למפתח שגיאות או אזהרות בזמן תהליך הקימפול, ואילו מהצד השני - ביצוע וריפיקציה פורמלית לקוד ("Formal Verification") מורכב במספר לא מועט של מונים, דורש משאבים ומאמצים רבים יותר, אך בשל זאת, מסוגל למצוא כשלים מתוחכמים ומורכבים יותר, אך עם זאת, ניתן לראות כי ביצוע וריפיקציה פורמלית לא אומצה באופן ניכר בשוק, וזאת מפני חוק התשואה

² Splint User's Manual, <http://www.splint.org/manual/html/sec1.html>

הפוחתת (ישנן מספר מערכות הפעלה שעברו וריפיקציה פורמלית כגון seL4 microkernel Secure Embedded L4 של ICTA³).

כאמור, הבדיקות שאנו נבצע במהלך המחקר יבוצעו בעזרת Klocwork Insight, הכלי הנ"ל פותח מטרות בקרת איכות ואבטחה בתוכנות שנכתבו בשפות C, C++, ו-JAVA. המוצר מגיע עם שלל פלאגינים עבור המפתחים, כלים לניתוח ארכיטקטורה, כלים לביצוע מדדים ודיווח. לכלי גרסאות למערכות ההפעלה Windows של מיקרוסופט ולהפצות השונות של מערכת ההפעלה Linux⁴.

באופן כללי, בדיקות SCA בודקות מקרים כגון התנגשויות בטיפוסי המשתנים, שימוש במשתנים לפני הצהרה עליהם, הצהרה על משתנים ללא השימוש בהם, קטעי קוד שלא ניתן להגיע אליהם, ערכי חזרה שגויים, לולאות אינסופיות, משפטי CASE לא אבסולוטים וכו', בנוסף, ברוב הכלים ניתן לבחור אילו בדיקות נרצה להריץ ואילו לא. בכלי SCA קצת יותר מתקדמים, ניתנת למשתמש האפשרות ליצור סט בדיקות מתואמות אישית על מנת למצוא כשלים או בעיות ספציפיות יותר בקוד-המקור.

לדוגמא, ב-Klocwork, ניתן ליצור פרופיל עם סט בדיקות שיעודכן ברשימת ה-API שהוחרגו במסגרת ה-SDL (Security Development Lifecycle) של מיקרוסופט (banned.h), גמישות כזו, מאפשרת למבצע הבדיקות, לבצע בדיקות ספציפיות ואיכותיות יותר, וככל שהבדיקות יהיו ספציפיות יותר - כך התוצאות שיתקבלו יהיו רלוונטיות יותר.

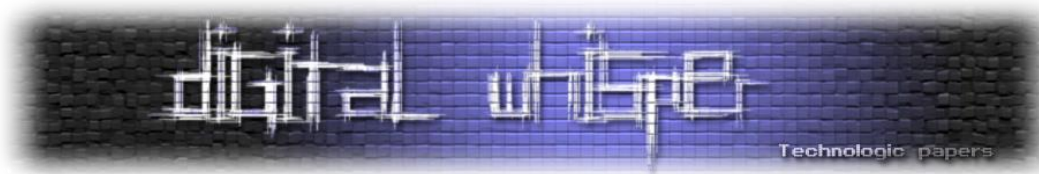
באופן כללי, רוב כלי ה-SCA נועדו להיות גמישים ולאפשר ולמתכנתים ולמבקרי איכות לבחור את "כמות המאמץ" אותו הם מעוניינים לבצע, וכך להגיע לנקודות המתאימות בעקומה שהוצגה בתרשים 2, כאשר מפעילים מספר בדיקות שונות על אותו הקוד, ניתן להגיע למצב של False Positive, במקרים כאלה, ניתן להוריד את הבדיקות שהציפו את אותן ה-False Positive, וכך "לנקות את הרעש" מדו"חות הבדיקה.

על הניתוח עצמו

כאמור, את הניתוח עצמו אנו מעוניינים לבצע על הקרנל של לינוקס, בעזרת Klocwork, ולאחר הניתוח נדון על תוצאותיו. גרסאת ה-Klocwork שבה השתמשנו לטובת הניתוח הינה 9.2.0.6223 וגרסאת הקרנל שעליה בוצעו הבדיקות הייתה 2.6.32.9. בתמונה בעמוד הבא, ניתן לראות את הקונפיגורציה (סט הבדיקות) שהגדרנו בעת הפעלת הבדיקה.

³ Secure Microkernel Project (seL4), <http://ertos.nicta.com.au/research/seL4/>

⁴ Klocwork Insight, <http://www.klocwork.com/products/insight/>



- C and C++
 - Attempt to Use Memory after Free
 - Buffer Overflow
 - C/C++ Warnings
 - COM defects
 - Calculated Values Never Used
 - Concurrency
 - DNS Spoofing
 - Ignored Return Values
 - Improper Memory Deallocation
 - Inappropriate Iterator Usage
 - Memory Leaks
 - Mismatched Return Types
 - Null Pointer Dereference
 - Parse warning defects
 - Porting issues
 - Print functions format
 - Registry Manipulation
 - Resource Handling Issues
 - Scan functions format
 - Strong Type Checkers
 - Suspicious Code Practices
 - Unreachable Code
 - Unused Local Variables
 - Unvalidated User Input
 - Use of Uninitialized Data
 - Weak Encryption

כל V מסמל משפחה של סט בדיקות שמטרתן לאתר בקוד המקור כשלים העונים לסוג הספציפי של הבדיקה, לאחר ניתוח הקרנל, המדדים של Klocwork החזירו את הנתונים הבאים:

Total number of files	13999
Total number of C/C++ files analyzed	13868
Total number of system files analyzed	131
Total lines of code (Source LOC)	4309863
Total lines of comments	1358746
Total number of entities	944835
Total number of functions/methods	162814
Total number of classes/types	42797

ניתוח הפגיעויות

בטבלה מצד שמאל ניתן למצוא את רשימת ה-CVE של הפגיעויות פומביות אשר נמצא בקרנל של לינוקס ואפקטיביות, בין היתר גם לגרסא 2.6.32.9 שמעניינת אותנו כרגע (CVE הינו קיצור של Common

CVE-ID	Issue Details	CVSS Score
CVE-2011-2695	Multiple off-by-one errors	Medium
CVE-2011-1770	Integer underflow	High
CVE-2011-2534	Buffer overflow	Medium
CVE-2011-1746	Multiple integer overflows	Medium
CVE-2011-1013	Integer signedness error	High
CVE-2011-1593	Multiple integer overflows	Medium
CVE-2011-0711	Improper memory initialization	Low
CVE-2011-0712	Multiple buffer overflows	Medium
CVE-2010-3876	Improper memory initialization	Low
CVE-2010-3859	Multiple integer signedness errors	Medium
CVE-2010-3861	Improper memory initialization	Low
CVE-2010-4082	Improper memory initialization	Low
CVE-2010-3310	Multiple integer signedness errors	Low
CVE-2010-3084	Buffer overflow	High
CVE-2010-2478	Integer overflow	High
CVE-2010-3015	Integer overflow	Medium
CVE-2010-4656	Improper memory allocation	Medium
CVE-2010-4655	Improper memory initialization	Low

Vulnerabilities and Exposures - מאגר וסיווג של חולשות שהתפרסמו באינטרנט).

חשוב לציין כי הרשימה הנ"ל חלקית ביותר, קיימות עוד מספר רוב של פגיעויות שזוהו ופורסמו באינטרנט תחת המאגר של ה-NVD⁵ (קיצור של National Vulnerability Database) גרסאת הקרנל 2.6.32.9 שוחררה בפברואר 2010, והפגיעויות המופיעות בטבלה משמאל פורסמו ב-NVD בין פברואר 2010 לבין יולי 2011, תקופה זו הינה תקופת הדגימה שלנו.

ע"י ביצוע SCA על הגרסה הנ"ל של הקרנל, יכולנו לזהות את כלל הפגיעויות שמופיעות בטבלה, מדובר ב-10% מכלל החפגיעויות שהתפרסמו בכל תקופה הדגימה.

אומנם, לא את כל הפגיעויות שדווחו ופורסמו ב-NVD הנוגעות לגרסאת הקרנל שאותה בדקנו ניתן למצוא בעזרת SCA בלבד, אך הפגיעויות שנמצאו (כאמור, 10 אחוז מכלל אלו שנמצאו) כוללים מספר לא מבוטל של פגיעויות שקוטלגו כ-"סיכון גבוהה" (כ-22.3% מהן), ו-44% מהן דורג כ-"סיכון בינוני", ו-33% מהן דורגו כ-"מהוות סיכון נמוך". כלל הדירוגים בוצעו ע"י מחשבון ה-CVSS (קיצור של Common Vulnerability Scoring System).

מדגם זה הינו דוגמא למקרה בו מספר לא מבוטל של פגיעויות יכלו להמצא קודם לכן, עוד בשלב הפיתוח, לו היה מבוצע SCA כחלק מתהליך בניית הקוד. ובנוסף - חשוב לזכור כי יש פגיעויות אשר בהגדרה לא ניתן למצוא בעזרת SCA, סוגי הפגיעויות אשר ניתן למצוא ע"י ביצוע SCA כוללות בין היתר: Buffer Overflows, Integer Signedness Error, Integer Overflows / Underflows, Improper Memory Initialization. מדובר בנקודות תורפה שבדרך כלל מנוצלות על מנת ליזום התקפות בעלות אופי זדוני על

⁵ National Vulnerability Database, <http://nvd.nist.gov/>



המערכות השונות. דוגמא טובה לכך, היא הפגיעות Buffer overflow שנוצלה על ידי התולעת Morris (1988) והתולעת קונפיקר (2008).

כאשר משלבים קוד מבוסס קוד-פתוח עם קוד קנייני, בהחלט כדאי לאתר פגיעויות כאלה בשלבים מוקדמים יותר של התהליך. והדוגמא שראינו פה על הקרנל של לינוקס הינה אינדיקציה טובה מאוד לכך.

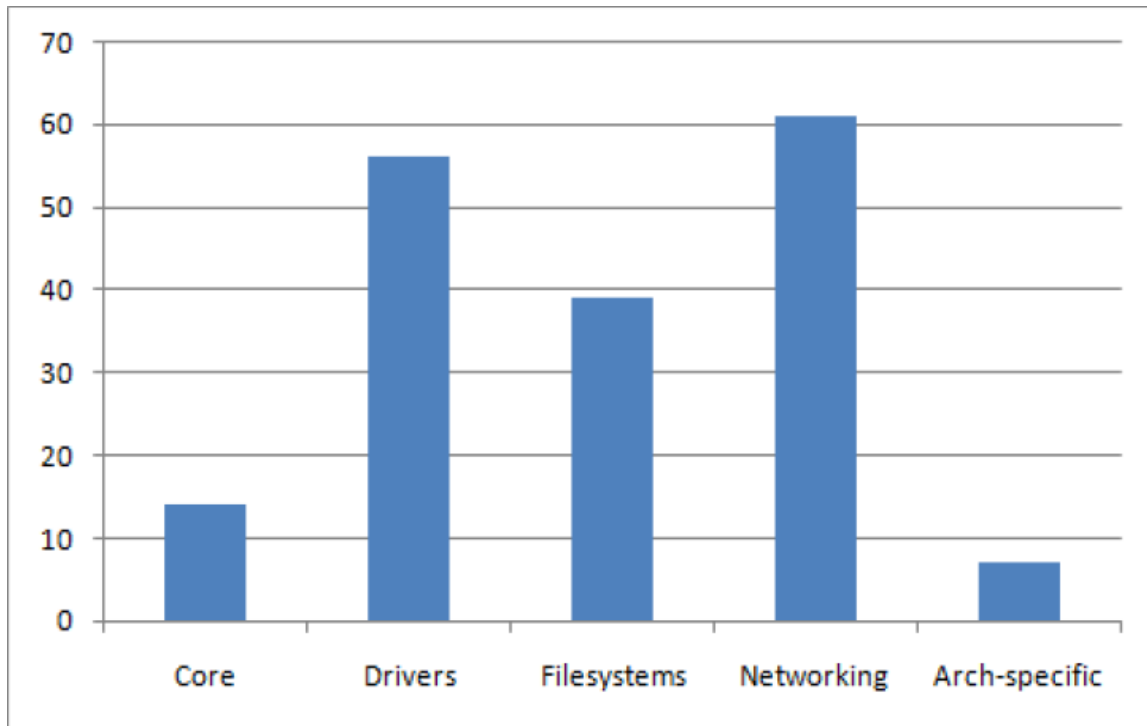
מעבר ליכולת של זיהוי מוקדם של פגיעויות כגון אלו המופיעות בטבלה מהעמוד הקודם, כלי ה-SCA מסוגל לזהות קטעי קוד קריטיים אשר עשויים לדרוש חקירה נוספת של המתכנת, על מנת לבדוק האם אכן קיימות בו פגיעויות והסבירות ליכולת לניצול שלהן. האינטרס של כל ספק תוכנה הוא לאתר את הפגיעויות הללו עוד לפני יציאת המוצר לשוק. וכך גם כאשר מדובר בקוד-פתוח, האינטרס של ספק התוכנה הינה לאתר את הפגיעויות והבעיות התוכנה לפני קהילת הקוד-הפתוח, אינטרס זה נובע, בין היתר, מפני שהרבה יותר זול לתקן באג בעת שלב הפיתוח מאשר אחרי שהמוצר נמצא בשוק.

על מנת לבדוק באילו רכיבי קרנל קיימים יותר פגיעויות חילקנו את קטעי הקוד לשש קטגוריות שונות⁶:

- **Core** - קטגוריה זו כוללת את הקבצים ב-init, block, ipc, kernel, lib, mm ותיקיות משנה וירטואליות.
- **Drivers** - קטגוריה זו כוללת את הקבצים ב-crypto, drivers, sound, include/acpi, include/crypto, include/drm, include/media, include/mtd, include/pcmcia, include/rdma, include/rxrpc, include/scsi, include/sound, include/video ותיקי-התיקיות.
- **Filesystems** - קטגוריה זו כוללת את הקבצים ב-fs ובתיקי-התיקיות.
- **Networking** - קטגוריה זו כוללת את הקבצים ב-include/net ובתיקי-התיקיות.
- **Arch-Specific** - קטגוריה זו כוללת את הקבצים ב-arch, include/xen, include/math-emu, ב-include/asm-generic ובתיקי-התיקיות.

⁶ Kernel development statistics for 2.6.35, <http://wn.net/Articles/395961/>

מבט מהיר ב-NVD, מלמד כי רוב הפגיעויות שפורסמו (בתקופת הדגימה שלנו), התגלו ברכיבים הנמצאים בקטגוריות "רשת", "דרייברים", וב-"מערכת הקבצים". בגרף הבא ניתן לראות את הפילוג מוחשית יותר:



[תרשים 3 - פילוג הפגיעויות שפורסמו עבור כל קטגוריה]

ניתן ליחס את מצב הפילוג הנ"ל לכך שה-Network Stack, אשר עוסקת, כמובן, בניהול ומימוש יכולות הרשת של מערכת ההפעלה, היא אחד היעדים האטרקטיביים ביותר למחקר. איתור וניצול פגיעות איכותות ברכיב זה מאפשר לתוקפים ליזום הרצת קוד מרוחקת על המערכת וכך להשתלט עליה מרחוק.

ניתוח מורכבות

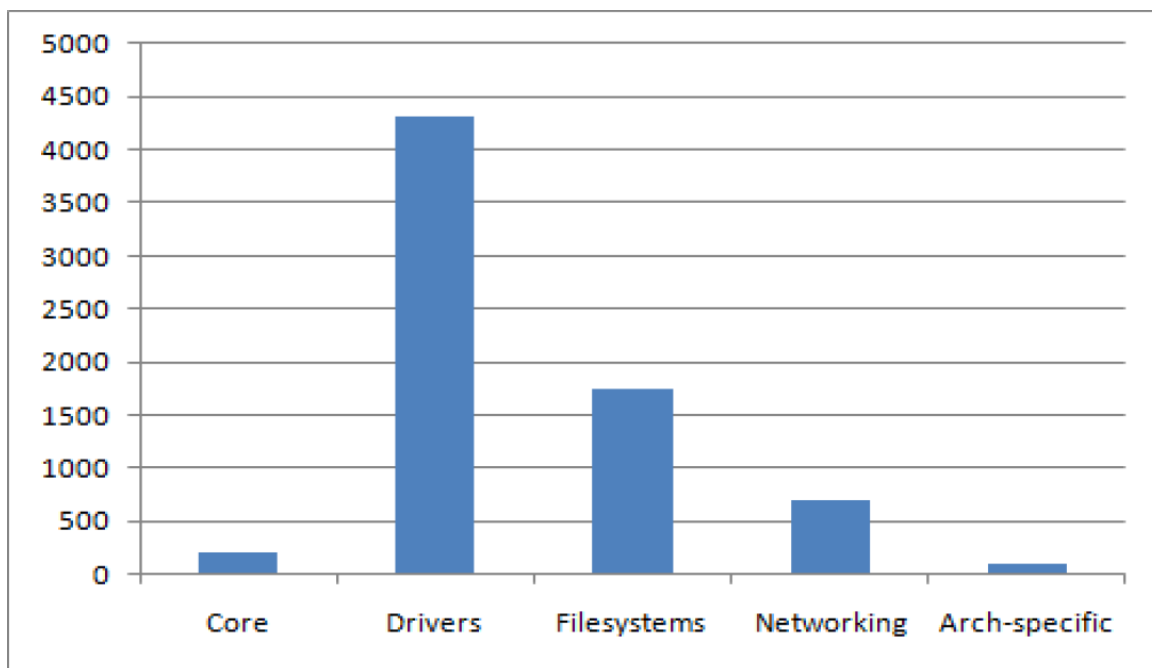
בדרך כלל, כלי SCA מסוגלים לחשב את מדד המורכבות עבור קוד (Cyclomatic Complexity) התוכנית אותה הם מנתחים, בקצרה מאוד, מדובר בחישוב של נוסחה המורכבת בעיקר ממספר ההחלטות שלש בקוד התוכנית, או במילים אחרות - מדידה של מספר נתיבי הזרימה שבהן הקוד יכול לזרום, ככל שיש יותר נתיבי זרימה - כך מורכבות הקוד עולה.

המכון הלאומי לתקנים וטכנולוגיה (NIST) ממליץ למפתחים לבצע בקרה על מורכבות הקוד שלהם, ולפצל אותם, לקטעי קוד קטנים יותר - ומורכבים פחות, כאשר רמת המורכבות עולה על 10 (במקרים חריגים במיוחד, ניתן לעמוד בתקן, גם עם מורכביות ברמה 15, אך מדובר במקרים חריגים שדורשים אישור פרטני ומיוחד⁷).

כאשר באים לבצע בקרה על קוד בעל מורכבות גבוהה, הדבר נעשה כמעט בלתי אפשרי למתכנת אנושי, כמעט ולא ניתן לעקוב בצורה נכונה אחרי כלל זרמי התוכנית, ולכן, במידה וקוד מורכב מתעדכן על ידי מתכנת חדש - הסבירות להכנת באג עולה עם רמת המורכבות של הקוד. אם מורכבות התוכנה עולה מעל 50, התוכנה נחשבת כ-"בלתי ניתנת לבדיקה ומסוכנת ביותר", לא מעט מחקרים מראים קשר ישיר בין רמת המורכביות של קטעי קוד והאפשרות לתחזוק הבדיקות שלהם, וככל שהאפשרות לתחזוק הבדיקות שלהם קטנה - כך ההסתברות להכנסת קוד עם פגיעות בעת תיקון, שיפור או ביצוע Refactoring גדל.

בפרייקטים כגון הקרנל של לינוקס, רוב מאמץ הפיתוח מועבר באמצעות Mailing-list (רשימות דיור), המפתחים הרבים פרושים על פני הגלובוס, וביניהם מחולקות רמות שונות של כישורי פיתוח, פיתוח במצב שכזה מהווה אתגר לא פשוט עבור כל ישות מרכזית אשר אחראית על תיאום מאמצי הפיתוח.

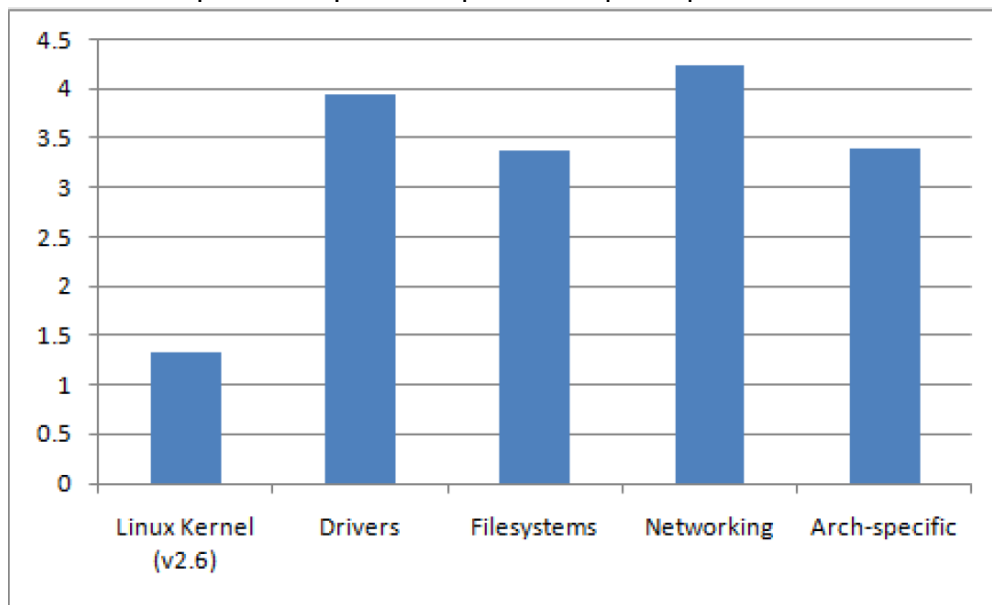
הגרף הבא מציג את כמות קטעי הקוד העוברים במדד המורכבות את 20, עבור כל אחת מהקטגוריות שחילקנו:



[תרשים 4 - מספר קטעי הקוד בגרסה שנבדקה העובר את רמת מורכבות 20 עבור כל אחת מהקטגוריות]

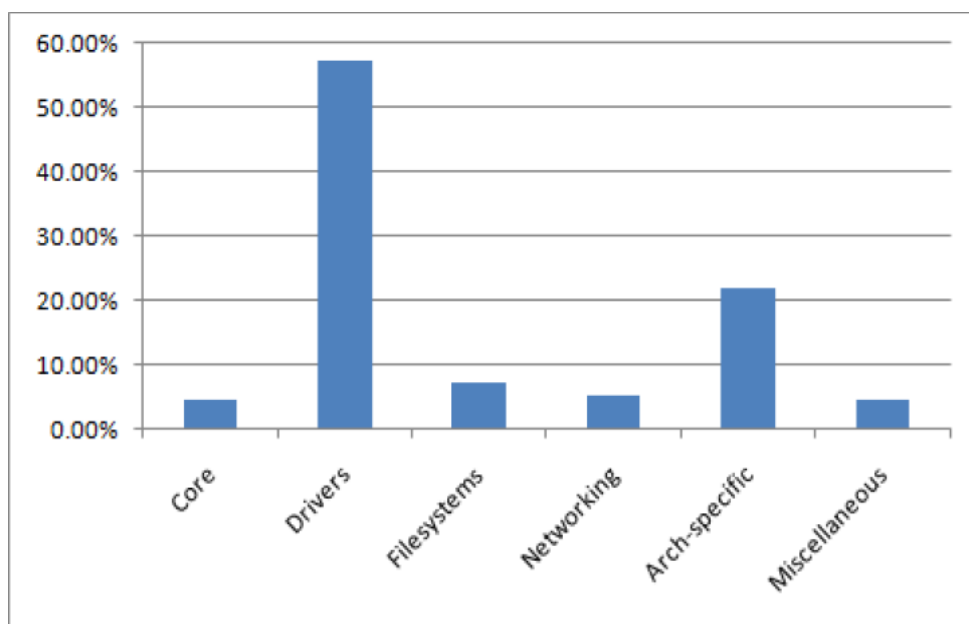
⁷ Cyclomatic complexity, Wikipedia, http://en.wikipedia.org/wiki/Cyclomatic_complexity

את רמת המורכבות הממוצעת של קטעי הקוד עבור כל קטגוריה ניתן לראות בגרף הבא:



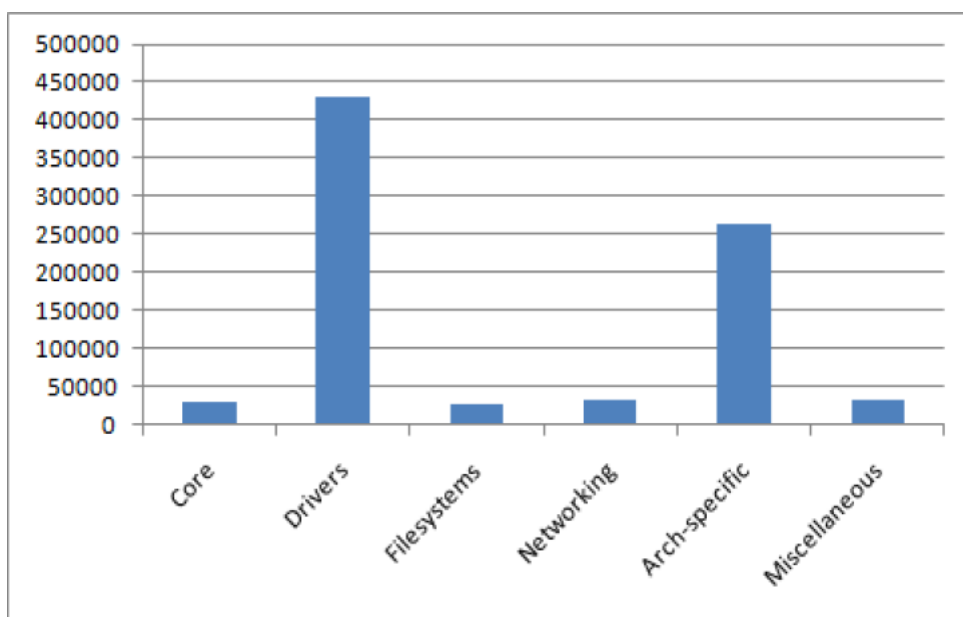
[תרשים 5 - רמת המורכבות הממוצעת של קטעי הקוד בגרסה שנבדקה עבור כל אחת מהקטגוריות]

העמודה השמאלית ביותר, מציגה את רמת המורכבות הממוצעת עבור כלל הקרנל (גרסה 2.6). גרף זה מציג בצורה חד-משמעית כי המורכבות של כל רכיב בודד בקרנל של לינוקס היא גבוהה בהרבה מהמורכבות הממוצעת בכלל הקרנל עצמו. מבט מהיר ב-NVD מראה כי רוב הפגיעויות שפורסמו בקרנל של לינוקס מנצלות כשלים הנמצאים בקטעי קוד המכילים מספר רב של קטעי קוד עם מורכבות גבוהה, כגון הרכיבים בקטגוריית "Drivers", "Filesystems" ו-"Networking", כמו שהוצג בתרשים 3. בתרשים הבא ניתן לראות באחוזים, את הנתח של כל קטגוריה מתוך כלל הקרנל של לינוקס (גרסה 2.6):



[תרשים 6 - נתח כל קטגוריה מכלל הקרנל של לינוקס בגרסה שנבדקה]

ובתרשים הבא, ניתן לראות את את כמות השורות שהשתנו בגרסה 2.6 עבור כל קטגוריה בקרנל של לינוקס:



[תרשים 7 - כמות השורות שהשתנו בגרסת הקרנל שנבדקה]

מעניין לראות כי למראות שהקטגוריה "Networking" מכילה מספר קטן יותר של קטעי קוד מורכבים (תרשים 4) ומספר קטן יותר של LoC ("Line Of Code") מאשר קטעי הקוד בקטגוריה "Drivers" או "Filesystems", עדיין רוב הפגיעויות שפורסמו ב-NVD מנצלים כשלים ברכיבים הנמצאים תחת הקטגוריה "Networking". הסיבה לכך מופיעה בפסקה הקודמת - "ניתוח הפגיעויות שנמצאו".

יתר על כן, למרות שקטעי הקוד תחת הקטגוריה "Networking", מכיל מספר קטן יותר של LoC, המורכבות הממוצעת שלו היא הגבוהה ביותר בהשוואה לשאר קטעי הקוד בקטגוריות הנוספות, מה שיכול להסביר את העובדה שברכיב זה נמצאו הכי הרבה פגיעויות ולהשליך על כל שככל שקטעי הקוד ברכיב מורכבים יותר - כך הסבירות למצוא בהם פגיעויות עולה.

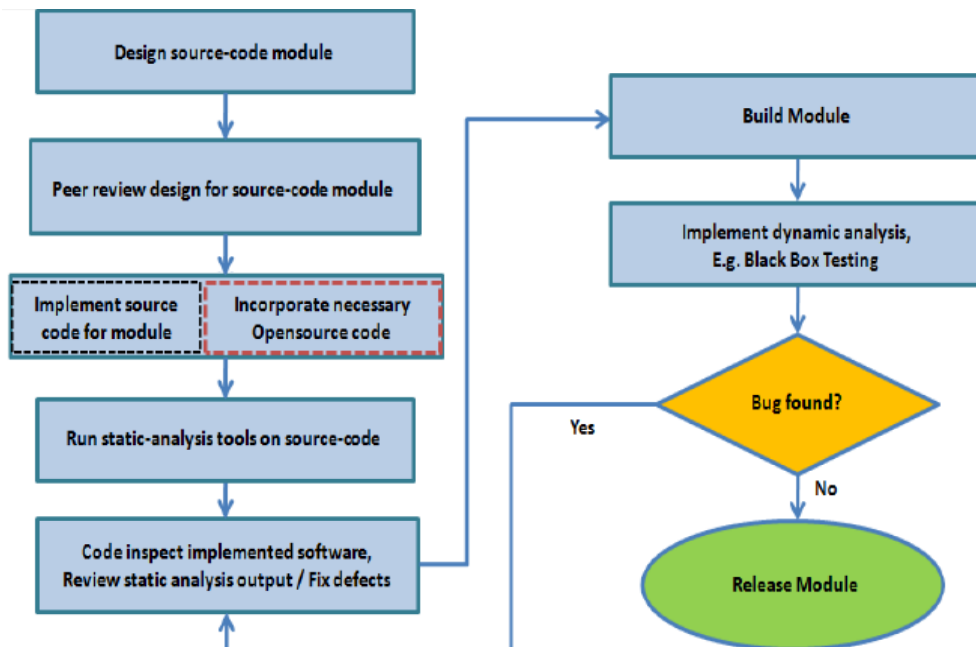
נתון מעניין נוסף הוא, שלמרות שמספר קטעי הקוד המשוייכים לקטגוריה "Arch-specific" עם רמת מורכבות גבוהה, הוא הקטן ביותר (תרשים 4), עדיין בוצעו בהם לא מעט שינויים (תרשים 7). ויתר על כן - על פי הנתונים בתרשים 5, ניתן לראות כי המורכבות הממוצעת של קטעי הקוד בקטגוריה "Arch-specific" היא בין הגבוהות בהשוואה לשאר הקטגוריות, מה שיכול להעיד על כך שברכיבים עם ממוצע מורכבות גבוהה יש נטייה לבצע שינויים רבים (תרשים 7).

באופן כללי, מהניתוח שביצענו ניתן לראות את הדפוסים הבאים:

- קיימת הסתברות גבוהה יותר מציאת פגיעויות ברכיבים עם רמות המורכבות הגבוהות ביותר (תרשימים 3,4 ו-5) - לדוגמא, רכיבים המשוייכים לקטגוריות כגון: "Drivers", "Filesystems" ו-"Networking".
 - רכיבים מורכבים, עם מספר רב יותר של שורות יעברו שינויים רבים יותר (תרשימים 6 ו-7), לדוגמא, רכיבים המשוייכים לקטגוריות כגון: "Drivers" ו-"Arch-specific".
 - ברכיבים "קריטיים" (כגון רכיבים המשוייכים לקטגוריה "Networking") מקרי הדווח אודות פגיעויות שנמצאו הוא מהגבוהים ביותר.
- מהתבוננות בנתונים אלה, אנו מסוגלים לרכז את משאבי ה-SCA שלנו על עבר אותם רכיבים קריטיים, כפי שיתואר בפסקאות הבאות.

הצעת זרימת עבודה חלופית

כפי שהוזכר קודם כן, והוצג בתרשים 1, כאשר, בעת תהליך שילוב קוד מבוסס קוד-פתוח עם קוד של תוכנה מסחרית, מבצעים מספר מועט יותר של בדיקות וניתוח סטטי בהשוואה למספר הבדיקות שמבצעים עבור הקוד של המוצר עצמו. בזרימת העבודה החלופית - כמו שמחקר זה מציע, אשר מוצגת בתרשים הבא:



[תרשים 9 - זרימת העבודה החלופית]⁸

⁸ Integrate static analysis into a software development process, Design Article, EE Times, <http://www.eetimes.com/design/embedded/4006735/Integrate-staticanalysis-into-a-software-development-process>

אנו ממליצים להכליל את הפלט של כלי הניתוח הסטטי שהועפל עבור הקוד קוד התוכנה הקנייני ועבור הקוד-הפתוח בזרימת העבודה הרשמית. זרימת העבודה הנ"ל תכניס את הבקרה על הקוד המבוסס קוד-פתוח לתהליך מסודר. כל פגיעות שתמצא הן בקוד הקנייני והן בקוד הפתוח תתוקן לפני שתגיע לשלב הבדיקה הדינאמית וכמובן - לפני שהמוצר יגיע לשוק. כמובן שתהליך זרימה זה יפעל תחת תנאי הרשיונות של הקוד-הפתוח וכל באג או שינוי שיבוע ידרוש תקשורת עם המתחזקים של הקוד במקור.

תהליך זה אומנם לא ימצא את כלל הפגיעויות, אך יכול עזור לאתר הרבה כשלים בשלבים מוקדמים של תהליך הפיתוח, נתון שיינתן הזדמנות לתקן את הכשלים הללו קודם לכן, כי שהוצג בפרק אשר דן בניתוח הפגיעויות. יתרה מזאת, בתהליך זה תנתן האפשרות של לשקול את לעשות שדרוג של קטעי קוד מבוססי הקוד-הפתוח שכבר שולבו במוצר.

זרימת העבודה החלופית תהיה יעילה אף יותר בתרחיש הבאים:











- במקרים שבהם ישנו הצורך לבחון האם יש שדרוג מסוים הוא מעשי - ע"י בדיקת באגים שותקנו ובאגים חדשים שהוצפו, ואף הערכת המאמץ הנדרש להשקיע על מנת תקן את אותם הבאגים החדשים שהוצפו. הערכת מאמץ זה לפעמים היא המכריעה בעת קבלת ההחלטה האם לשחרר גרסה חדשה או לא.
- במקרים בהם יש להעריך את הפוטנציאליות של ביצוע הסבה של קוד מסביבת עבודה אחת לחברתה (Porting) - ע"י הבנת מערכת הייחוסים הקיימת בין רכיבי הקוד-הפתוח לבין רכיבי התוכנה הקניינית.

האתגרים בעת אימוץ זרימת העבודה החלופית

זה נכון שניתוח סטטי של קוד מבוסס קוד-פתוח הוא שימושי ויעיל בעת זיהוי כשלים ופגיעויות בקוד עוד בשלבי הפיתוח המוקדמים. אך יש אתגרים טכניים ולעיתים אף אתגרים ספציפיים בפרויקט מסוים אפשר יכולים להפוך עיצה זאת ללא מעשית. מקרים כגון:

- מקרים שבהם כלי ה-SCA מפיקים הודעות False Positive. תהליך הסקירה וביטול הודעות מסוג זה יכולה להוות משימה מרתיעה גם לצוות בקרת האיכות וגם לצוות אבטחת-המידע, במיוחד כאשר מדובר בפרוייקטים אשר נמצאים תחת אילוצי זמן, תקציב ומשאבים נמוכים. ברגע שהתראות הללו יבוטלו ניתן יהיה לתקן את הבעיות האמיתיות ע"י צוות הפיתוח, אך לפעמים תהליך זה עשוי לדרוש כי התיקונים בקוד יעברו לידי המתחזקים המקוריים של הקוד-הפתוח לפני השקת המוצר, בעקבות הרשיונות תחתיהם הופץ הקוד-הפתוח.

- הכלי Klocwork תומך עשרה "דרגי חומרה" שניתן להצמיד לכל כשל שנמצא, כאשר 1 מסמל כשל ברמה "קריטי" ו-10 מסמל כשל ברמה "אינפורמטיבית", ניתן לראות את החלוקה בתמונה הבאה:

Severities		
Number	Name	Icon
1	Critical	
2	Severe	
3	Error	
4	Unexpected	
5	Investigate	
6	Warning	
7	Suggestion	
8	Style	
9	Review	
10	Info	

ובדומה לכך עובדים רוב כלי ה-SCA. בשל המספר הרב של אירועי ה-False Positive יש נטייה מסויימת להתעלם מסוגיות שאינן מסווגות כ"קריטיות", נטייה זאת עלולה לגרום לכך שיתעלמו מכשלים ופגיעויות אמיתיות. לכן, ישנו הצורך לנפות סוגיות אלו. משימה לא פשוטה אך ברוב המקרים - שווה את המאמץ.

- ביצוע בדיקות SCA איכותיות לכל ישן או חדש שנכנס דורשת משאבים ואנשים מנוסים ביותר, שבעקבות כך לא יהיו זמינים לפרוייקטים אחרים שיכול להיות שכבר ככה נמצאים עם חוסר של כח אדם איכותי.

כמו שראינו, אימוץ זרימת העבודה כמו שהוצגה במאמר דורשת לענות על לא מעט אתגרים, ובמקרים מסוימים עלולה אף להראות כלקיחת סיכון מיותר. ולכן נציג מספר דרכי פעולה אפשריות על מנת להתמודד עם אתגרים אלו:

- דרך אחת להתמודד עם אתגרים אלו, הינה לרכב את מאמץ הניתוח והבדיקות ברכיבים בעלי מורכבויות גבוהות בקוד-הפתוח שאומץ, ובכך לצמצם את סקופ הבדיקה ובכך גם לצמצם את עלות המשאבים. ניתן לבצע זאת כפי שהצגנו בסעיף הקודם, ראינו כי רכיבים בעלי מורכבות גבוהה נוטים לכלול יותר כשלים. בעת ביצוע ניתוח סטטי לקוד בעל מורכבות גבוהה קשה יותר לסווג תקלות כ-False Positive, כי יש להתעמק בקוד המורכב על מנת לסווג את סוג ההתראה.
- דרך נוספת שבעזרתה ניתן להתמודד עם אתגרים אלו, היא לזהות את הרכיבים המסווגים כ-"קריטיים" בפרוייקט (כגון רכיבי הקוד תחת הקטגוריה "Networking") ותרכז את המאמץ בהם, כפי שהוצגה בפרק אשר דן על ניתוח הפגיעויות.
- לא תמיד יהיה פשוט לחקור את טבען של התראות ה-False Positive, אך בהרבה מקרים יהיה שווה את המאמץ. במקרים רבים השגיאות הללו יגרמו מתקלה בכלי הבדיקה עצמו או מהרצתו תחת קונפיגורציה אשר לא מתאימה לבדיקה. אך במקרים אחרים, התראות ה-False Positive נובעות

אבטחת קוד פתוח ע"י ניתוח קוד סטטי

www.DigitalWhisper.co.il



מחוסר ידע של המפתח או הבודק, בייחוד במקרים בהם מדובר בכשלי אבטחה, ששם כשלים אלו עדינים ופחות נראים-לעין לא מקצועית. במקרים כאלה הבודק עשוי שלא להבין למה אותה שורת קוד נחשבת לבעייתית ולחשוב כי הכלי לא פועל כשורה. ניתן לפתור זאת על ידי הוספת מפתחים בכירים יותר למשוואה זו.

מסקנות

תוכנות המבוססות על קוד-פתוח זמינות לכל דורש, כדוגמת לינוקס, וניתן להשתמש בהן כל עוד דובקים בתנאי הרישיון תחת הן מופצות. בניתוח שלנו, בחרנו להשתמש ב-Klockwork ככלי SCA ובקרנל של לינוקס כמטרה. גם הכלי וגם המטרה הן רק דוגמאות על מנת להדגיש את הנושא של אבטחת מידע בקוד-פתוח המשולב עם תוכניות קנייניות, באופן כללי, ניתן להרחיב את הדוגמאות הנ"ל לכלי SCA אחרים ולפרוייקטי קוד-פתוח אחרים.

למרות שדיווח על כשלים בפרוייקטים מבוססי קוד-פתוח יש ערוצים רשמיים ומקובלים, ובהם נעשה השימוש, אך אותם כשלים התגלו על ידי מפתחים אשר התעסקו עם מקרים ספציפיים של אותן התוכנות ולא במסגרת מאמץ מרוכז, מאמצים כאלו חסרים בקהילת הקוד-הפתוח. אך עם זאת, בזמן האחרון חברות כגון SCA Klockwork ו-Coverity⁹ לקחו את היוזמה והחלו לפעול בכיוון זה. אך גם אז, הקצב שבו גרסאות קוד מבוססות קוד פתוח משוחררות מציב אתגר לא פשוט ליוזמות מסוג זה.

מאמצים כגון המאמצים של CAS (קיצור של Center Assured Software¹⁰) של הסוכנות לבטחון לאומי (ה-NSA), אשר הציג מחקר של כלי ניתוח סטטי עבור קוד הנכתב ב-C/C++ ו-Java בשנת 2010 ע"י שימוש במקרי בדיקה זמינים כגון Juliet Test Suites¹¹ הם צעדים בכיוון הנכון. אך גם אז - אין מדדים מדויקים ומוחלטים לבחירה של כלי SCA מסוים. כלי SCA שונים נוטים למצוא כשלים שונים באותו קטע קוד, והחפיפה בין התוצאות הופכת להיות כמעט אפסית כאשר מבצעים את הבדיקה ע"י שלושה כלי SCA שונים. כלל, לא כל הכשלים אשר נמצאו וסומנו ע"י כלי ה-SCA הם אכן כשלים, אחוז לא מבוטל של כשלים עשוי להעלם כאשר תבצע בדיקה בדיקה קפדנית יותר. אך עם זאת, במספר לא מבוטל של המקרים, שווה להמשיך עם הבדיקה בעת בדיקות דינמיות יותר, כגון בדיקות מבוססות Fuzzing¹², או על-ידי בדיקות DART (Directed Automated Random Testing)¹³, המחקר שלנו מראה מעבר לכל ספק, כי יישום בדיקות SCA לקוד המבוסס על קוד-פתוח מאפשר לאתר כשלים בשלבים מוקדמים יותר של תהליך

⁹ Coverity Scan 2010 Open Source Integrity Report, <http://www.coverity.com/html/press/coverity-scan-2010-reportreveals-high-risk-software-flaws-in-android.html>

¹⁰ On Analyzing Static Analysis Tools, National Security Agency Center for Assured Software, Black Hat Technical Security Conference, 2011

¹¹ Juliet Test Suites, <http://samate.nist.gov/SRD/testsuite.php>

¹² Peach Fuzzer, <http://peachfuzzer.com/>

¹³ DART: directed automated random testing, Patrice Godefroid, Nils Klarlund, Koushik Sen, Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, Volume: 40, Issue: 6, Publisher: ACM, Pages: 213-223

אבטחת קוד פתוח ע"י ניתוח קוד סטטי

www.DigitalWhisper.co.il



הפיתוח. יצרניות תוכנה אשר משלבות המוצריהן קוד-פתוח המבוסס על רשיונות GPL, ייתכן שידרשו להפוך את כל המוצר שלהן לקוד-פתוח. ניתן להשוות זאת אל מול שילוב קוד עם קוד-פתוח המבוסס על רשיונות MPL או Apache, אשר אינו מחייב את ספק התוכנה לשחרר את כלל המוצר שלו כקוד-פתוח. מקרים כאלה עלולים להוביל למצב שבו חלק או כל קוד התוכנה הינו חופשי, מה שיקל על מחפשי החולשות לאתר חולשה בקוד ללא כל צורך בהשקעה בביצוע הנדסה-לאחור. כתוצאה מכך, ניתן לראות כי כאשר משלבים קוד פתוח במוצר קנייני, רמת האבטחה של הקוד הפתוח הינה קריטית בדיוק באותה הרמה של הקוד הקנייני. ולכן חשוב להכליל את הבדיקות אשר מבצעים לקוד הקנייני גם על הקוד-הפתוח שנקלט לפרוייקט.

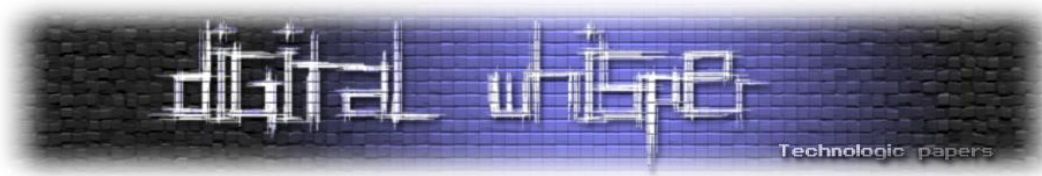
אנלוגיה לנושא הזה ניתן לראות כאשר מפתח מעוניין לבצע שימוש בסיפריית קוד המבוססת על קוד-פתוח, במקרה הזה המפתח יוריד את הבינארי עצמו וכמו-שהוא יכניס אותו לפרוייקט או שהוא יוריד את קוד-המקור ויכניס אותו בצורה מסודרת לסביבת הבניה של הפרוייקט? וזאת על מנת לא להיות תלוי באופן הבניה של אותו בינארי, שלא תמיד עשוי לעלות בקנה מידה אחד עם הפרוייקט עצמו. אם כך, למה לא לכלול את אותו הקוד בסקופ הבדיקות הכולל?

מובן בכל תעשיית פיתוח התוכנה כי ככל שהכשל בתוכנה ימצא בשלבים מוקדמים יותר - כך יהיה קל וזול יותר לתקן אותו. במקרים מסוימים, הדבר עלול לחסוך תביעות משפטיות יקרות ואף נזק בלתי הפיך למוניטין החברה.

במובנים רבים, המאמץ הנוסף הנ"ל, הינו קריטי בשיפור איכות המוצר, אמינותו ורמת האבטחה שלו. בסופו של דבר, ארגון אשר יכול להרשות לעצמו את העלות ויש להם צורך בנושא, ימצאו ערך רב במאמץ זה.

תודות

מחבר המאמר מבקש להודות ל-Wayne Trantow ולכל צוות ה-Opensource PDT של חברת אינטל על מתן תובנות רבות ויקרות ערך בעת מלאכת כתיבת המאמר.



ביבילוגרפיה וקישורים לקריאה נוספת

- Splint User's Manual, <http://www.splint.org/manual/html/sec1.html>
- Integrate static analysis into a software development process, Design Article, EE Times, <http://www.eetimes.com/design/embedded/4006735/Integrate-static-analysis-into-a-software-development-process>
- Klocwork Insight, <http://www.klocwork.com/products/insight/>
- Peach Fuzzer, <http://peachfuzzer.com/>
- The Myths of Security: What the Computer Security Industry Doesn't Want You to Know, John Viega
- Secure Microkernel Project (seL4), <http://ertos.nicta.com.au/research/seL4/>
- Linux Kernel, <http://www.kernel.org/pub/linux/kernel/v2.6/>
- Linux Kernel, Wikipedia, http://en.wikipedia.org/wiki/Linux_kernel
- Cyclomatic complexity, Wikipedia, http://en.wikipedia.org/wiki/Cyclomatic_complexity
- Coverity Scan 2010 Open Source Integrity Report, <http://www.coverity.com/html/press/coverity-scan-2010-report-reveals-high-risk-software-flaws-in-android.html>
- On Analyzing Static Analysis Tools, National Security Agency Center for Assured Software, Black Hat Technical Security Conference, 2011
- Kernel development statistics for 2.6.35, <http://lwn.net/Articles/395961/>
- National Vulnerability Database, <http://nvd.nist.gov/>
- Juliet Test Suites, <http://samate.nist.gov/SRD/testsuite.php>
- Integrating Static Analysis into a Secure Software Development Process, Kleidermacher, D.N., IEEE Conference on Technologies for Homeland Security, Issue Date: 12-13 May 2008, On page(s): 367-371
- A survey of static analysis methods for identifying security vulnerabilities in software systems, Pistoia, M., Chandra, S., Fink, S. J., Yahav, E., IBM Systems Journal, Issue Date: 2007, Volume: 46 Issue:2, On page(s): 265-288



- On the value of static analysis for fault detection in software, Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J.P., Vouk, M.A., IEEE Transactions on Software Engineering, Issue Date: April 2006, Volume: 32 Issue:4, On page(s): 240-253
- Using Static Analysis to Find Bugs, Ayewah, N., Hovemeyer, D., Morgenthaler, J.D., Penix, J., Pugh, W., IEEE Software, Issue Date: Sept-Oct 2008, Volume: 25 Issue:5, On page(s): 22-29
- DART: directed automated random testing, Patrice Godefroid, Nils Klarlund, Koushik Sen, Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, Volume: 40, Issue: 6, Publisher: ACM, Pages: 213-223

אז למה לי פוליטיקה עכשיו? / על רגולציה ואבטחת מידע

מאת דניאל ליבר

מעשה בחברה...

התקופה היא סוף המילניום, בעת גדילת בועת הדוט קום. חברות רבות בעלות תחום פעילות 'מסורתי' מחפשות למנף את התפתחות האינטרנט לטובת כיווני רווח חדשים, ואחת מהן הייתה אנרון (Enron). אנרון התחילה את פעילותה במסחר בגז וב-1992 הפכה לספק הגדול ביותר לגז טבעי בצפון אמריקה^[1]. על מנת למקסם רווחים אפשריים ולפזר את הסיכונים העסקיים, החליטה אנרון על גיוון בתחום ההשקעות שלה והחליטה למצוא חוזים גם בתחום תשתיות החשמל, מים, פסולת ושירותים אחרים ברחבי העולם. כמו כן, הוחלט על פתיחת EnronOnline - האתר הראשון למסחר בסחורות שונות בשוק האנרגיה העולמי. היקף המסחר היומי עמד על יותר מ-6 מיליארד דולרים^[2]. הפיתוח העסקי של החברה גם לכך ששווי המניות של אנרון עלה ביותר מ-300% בין השנים 1998-1990, ובעוד כ-140% בשנים 2000-1999.

במהלך שנת 2000 התברר כי אנרון הגישה דו"חות כספיים מבלבלים ומטעים מתוך כוונה להסתיר את היקף העסקאות והרווחים האמיתיים שביצעה, שהיה נמוך משמעותית מההיקף שדווח. מהלך כזה היה אפשרי בעקבות שיתוף פעולה בין בכירי אנרון ורואי החשבון החיצוניים של החברה אשר חתמו על הדוחות שפורסמו (חברת ארתור אנדרסן - Arthur Andersen), שנחשבה באותה תקופה לאחת מחמש החברות הגדולות בעולם לראיית חשבון ולאחר משבר זה הופסקה פעילותה^[3] וכן מניפולציות פיננסיות כדוגמת חתימה על עסקאות עם חברות בת (כך שהכסף לא באמת הגיע כרווח ממקור חיצוני). ערך המניות התרסקו מאיזור שווי של 90 דולר למנייה בקיץ 2000 ועד 26 סנט בנובמבר 2001^[4]. חודש לאחר מכן אנרון הגישה לבית המשפט בקשה להכרזה כפושטת רגל (נכון לאותה תקופה, מדובר היה בפשיטת הרגל בסכום הגדול ביותר אי פעם).

החקירות לאחר המקרה הובילו לשורה מזעזעת של חשיפות. כספי הפנסיה של מרבית עובדי החברה היו מושקעים בכספי מניות של החברה. בכירים באנרון מכרו את מניותיהם זמן מה לפני תחילת המשבר ובעקבות כך הרוויחו מאות מיליוני דולרים. לאחר הגשת הבקשה לפשיטת הרגל, הסנאט התחיל לחקור את החברה ומנהליה בחשד להונאה. התברר כי חלק גדול מתכתובות המייל והמידע בארגון נגרס ונמחק בחודשים הסמוכים לפשיטת הרגל ע"י מזכירות החברה בהוראת ההנהלה. את המידע הנוטר שנדרש ע"י הסנאט בצו מיוחד העבירה החברה באופן חלקי בלבד. כמו כן, המידע היה מוצפן ולצורך פענוחו הממשל

אז למה לי פוליטיקה עכשיו? / על רגולציה ואבטחת מידע

www.DigitalWhisper.co.il

נאלץ לשכור מומחים חיצוניים על מנת לשבור את ההצפנה^[6]. בסופו של עניין, הסנאט פרסם דו"ח שבו האשים כי מדובר באחת מההונאות החמורות שהתרחשו וכי הדירקטוריון של החברה לא עשו מספיק כדי לעצור את ההתרחשות^[7]. בכירי החברה נשפטו לעונשי מאסר וקנסות אך עונשים אלה לא השיבו את הכספים שאבדו.

חברת אנרון לא הייתה האחרונה שקרסה כתוצאה מאי פיקוח והעדר רגולציה. כשנה לאחר מכן קרסה חברת וורלדקום (WorldCom) כתוצאה מהונאה פנימית גם כן^[8]. בחינה רחבה של הנושא הובילה לתוצאה מתבקשת - עקב פשיטות הרגל של החברות והעובדה כי היו מעורבות בסקנדלים נוספים (לדוגמא, שערוריית הפסקות החשמל בקליפורניה^[9]), ועדת הסנאט הקשורה בתחום הבנקאות הובילה רגולציה חדשה ע"י הסנאטור פול סארבנס (Paul Sarbanes) ונציג בית הנבחרים מייקל אוקסלי (Michael Oxley), שלימים נקראה SOX על שמו. הרגולציה הכילה מנגנון פיקוח הדוק על מגוון היבטים בחברות נסחרות וציבוריות^[10].

יכול להיות שאתם מגרדים בראש ומנסים להבין מה הקשר של התיאור הנ"ל לאבטחת מידע; קודם כל, חשוב להסביר כי נזקים שאינם בהכרח טכניים (פגיעה בתדמית, הונאות, התכחשות וכד') יכולים להמנע על ידי אבטחת מידע ראויה. שנית, אומנם כתוצאה מקריסות החברות נולדה רגולציה ה-SOX, אבל זו הייתה רק הסנונית וחשוב שנכיר רגולציות נוספות.

אבטחת מידע ורגולציות - מי נגד מי?

בעת כינון רגולציה בתחום אבטחת המידע, מול עיניו של הרגולטור עומדים מספר היבטים^[11]. יש לזכור כי לרוב הגופים הנתונים לרגולציה הם בדרך כלל גופי ענק בעל השפעות רבות בתחומם וכן בעלי מידע על לקוחות רבים ובמגוון רחב של תשתיות, מיקומים וטכנולוגיות. סיכוי רב שבמקרה של בדיקה בתוך ארגונים כאלה, ימצאו סיכונים רבים כדוגמת:

- אמצעי זיהוי בלתי הולמים.
- הרשאות גורפות או הרשאות יתר.
- העדר מנגנוני וכלי אבטחה.
- ליקויי הקשחה וקונפיגורציה בתשתיות ואפליקציות.
- חולשות באבטחה פיזית.
- חוסר מודעות בנוגע לנהלי אבטחת מידע בתהליכי עבודה בארגון.

הסיכונים הנ"ל נחשבים לרוחביים ומקורם הוא לא רק בטכנולוגיות הארגון ובמאפייני המערכות, אלא גם בגורם האנושי ובעובדה כי ישנם תהליכי עבודה ויישומים חדשים לאורך תקופות זמן ארוכות. במקרים

מסוימים העדר אחדות בנהלים גרמה לכך כי קבוצות שונות בתוך הארגון אשר אחראיות על פעילויות דומות יפתחו ויתקינו מימושים שונים, ולשם כך הרגולטור מעוניין לקבוע סף אחיד.

הכנסת הרגולציה לארגון גורמת לכך כי אבטחת המידע הופכת לאבן מרכזית בתחום הפיקוח וכלי אשר באמצעותו ההנהלה יכולה למדוד את רמת התאימות שלה מול הסטנדרט הקבוע בחוק. חשוב להדגיש כי הרגולטור אינו אמור לקבל סמכות מהארגון כדי לנהל את מדיניות אבטחת המידע שלו, או לספק מתווה לגבי פתרונות טכנולוגיים והמלצות על מוצרים וספקים. מטרת הרגולטור היא לאפשר ניהול סיכונים מתמשך בהתאם למסגרת אחידה, כך שאפילו בזמנים דינמיים ותחת מספר רב של פרויקטים ופיתוחים חדשים יהיה ניתן לקבל את אותה תמונה לגבי המתנהל באופן רחבי בין כלל הארגונים הכפופים לרגולציה.

הגורמים הרלוונטיים בארגון לרגולציה אבטחת מידע הינם ה-CISO\CIO וההנהלה. לרוב, ה-CISO אחראי על ליווי מקרוב של הטמעת הבקורות ואמצעי המניעה אשר אמורים לעזור למערכות הארגון לעמוד בתקנים השונים. ה-CISO הינו הצינור בין המתרחש 'בשטח' לבין ההנהלה, שלרוב משמשת בעיקר בתור גורם-על בעל סמכויות שונות כדוגמת יישום מדיניות הדירקטוריון והתרבות הארגונית, כתיבת תוכנית עבודה וביצוע דיונים תקופתיים, קביעת סקרים תקופתיים ודיון בממצאים חריגים שעלו בסקרים אלה.

במסגרת פעילות הארגון, נבחנים פרמטרים רבים מול הרגולציה:

- פעילות המשתמשים - מי משתמשי הקצה? האם מדובר בקבוצות עובדים רגילות? האם מדובר בלקוחות? לכל סוג משתמש יש אופי פעילות וסיכון משלו.
 - סביבות העבודה - האם מדובר בעבודה מתוך הרשת של הארגון? אולי מדובר בעבודה מהאינטרנט, או ברשת שמיועדת לספקים ועבודה של B2B (Business to Business)? לכל סביבה יש ארכיטקטורה והגבלות משלה.
 - רגילות המידע - האם מדובר במידע תפעולי פשוט? אולי במידע פיננסי רגיש? גם מידע לגבי תשתיות המערכת וסוגי הטכנולוגיות יכול לשמש בתקיפות כנגד הארגון.
 - מנהלי המערכת - האם מדובר בעובדים פנימיים של הארגון? אולי מדובר במערכת שנתמכת ב-outsourcing? אילו הרשאות דרושות לניהול המערכת?
- ישנם פרמטרים רבים נוספים שניתן להציג אך הנ"ל מהווים נדבך חשוב בבחינת תאימות הארגון לרגולציה.

רגולציות ו-SMB (עסקים קטנים/בינוניים)

כפי שצויין בסעיף הקודם, הארגונים הכפופים לרגולציות הם לרוב גדולים, בסדרי גודל של היקפי מסחר במיליוני דולרים לחודש. אך האם אין חשיבות גם לעסקים קטנים בינוניים? הרי בכל זאת, גם הם מכילים מידע מסוים ופרטים רגישים על לקוחות. חלקם, על אף שאינם ארגונים קריטיים בקנה מידה לאומי, יכולים בהחלט להיות מורגשים במקרה של קריסה או אירוע אבטחת מידע (לדוגמא, בית תוכנה קטן אשר אחראי על תמיכה במספר ארגונים גדול במשק). כמו כן, המידע שנשמר בארגונים קטנים אינו רב כאשר מדובר על פריצה לארגון ספציפי, אך במקרה ומדובר במספר ארגונים יכול להיות שהמידע הכולל יהיה בעל נפח משמעותי (לדוגמא, אירוע ההאקר הסעודי ב-2012 [12] [13]).

עם זאת, ארגונים קטנים \ בינוניים אינם בעלי יכולות פיננסיות רבות כדי לעמוד ברגולציות מחמירות של אבטחת מידע. יכול להיות שבעתיד הקרוב נראה רגולציות המתאימות את עצמן לגודל העסק כדי למנוע מצב של 'הכל או כלום' [14].

קשיים בתאימות רגולציות

כנציגי אבטחת מידע בארגונים, אתם עלולים להתקל לא מעט בקשיים בעת נסיון להבין איך בדיוק אמורים לעמוד בתור סט הדרישות שהוגדר על ידי הרגולטור. באופן מפתיע, לא תמיד הקשיים מגיעים דווקא מהתקן עצמו, אלא לפעמים גם מגופי פיתוח ו-IT ובמקרים מצומצמים גם מההנהלה ומנהלי התקציבים בארגון [15].

- נוסח הרגולציה - לעתים הרגולטור, שאינו מגיע ממקום טכנולוגי, נוטה להשתמש במילים כלליות לביצוע פעולות אבטחה. לדוגמא, לעתים ישנה דרישה להצפנה של נתונים מסויימים. מן הסתם, ניתן לשים לב מיד שמדובר בניסוח מעורפל ביותר; אין הסביר איזה אלגוריתם מקובל ואיזה לא עבור תאימות לרגולציה, לא ניתנים הסברים על אורכי המפתחות וכד'. ישנן פעולות נוספות אשר מנוסחות באופן דומה במספר תקני רגולציה שמשאירות חלק גדול מההחלטות בידי ההנהלה ומנהל אבטחת המידע (ומכאן נמשיך לנקודה הבאה...)
- גופי פיתוח ו-IT - עקב העובדה שבמקרים מסויימים ההחלטה על אופן המימוש של הרגולציה הוא בידי הארגון עצמו, יכולים להתפתח מאבקים פנימיים שנוגעים למשאבים (חודשי אדם בפיתוח, התקנות ומוצרים ב-IT). יש צורך ביכולות שכנוע מיוחדות לעתים על מנת להדגים לגופים אלה כיצד הרגולציה רלוונטית ואילו תרחישי תקיפה היא יכולה למנוע.

- מנהלים תקציבים והנהלה - בסופו של דבר, הרגולציה משפיעה גם על התקציב. בין אם באמצעות הטמעה של מוצרים חדשים, הקצאת חודשי עבודה לטובת תאימות או אפילו ביצוע סקרים על מנת לקבל תמונת מצב בנוגע לארגון והטמעת הרגולציה. התקציב שעומד לרשותנו לפעמים משפיע גם על הסיכונים שמציגים מנהלי אבטחת המידע להנהלה - בסופו של דבר, הדירקטוריון והנהלה הם אלה אשר מדווחים לרגולטור על תאימות הארגון. במקרים מסוימים נוטה ההנהלה להתפשר על בקרה זולה ו-'רכה' יותר בנוגע לסיכון מסוים אם יתברר שמדובר שהסיכון הוא בסבירות ובחומרה נמוכות. ניהול סיכונים נכון צריך גם לכלול לעתים את האפשרות כי הארגון יחליט שלא לעמוד ברגולציה מסיבות פיננסיות (הקנס שניתן על אי עמידה ברגולציה יכול להיות זול משמעותית מתיקון סעיף כלשהו של אי תאימות לרגולציה).

אבטחת מידע בעולם הרגולטורי

כדי להדגים את התחומים והרגולציות השונות הקיימות הקשורות לאבטחת מידע, מובאים בפניכם מספר סעיפים מדגמיים מתוך כל רגולציה (מטרת המאמר אינה לבאר כל רגולציה ורגולציה אלא לתת מסגרת כללית). עם זאת, כמעט בכל רגולציה קיימות מספר נקודות חופפות שהן גם עקביות בנוגע לאבטחת מידע:

- אחריות הדירקטוריון והנהלה בפעילות אבטחת המידע בארגון.
- טיפול בפעילויות ובמידע הלקוח השמור בארגון.
- ניהול ספקים ומיקור חוץ.
- ניהול סיסמאות וזהויות.
- הפרדת משאבים ותפקידים בין סביבת ייצור וסביבת פיתוח.
- אבטחה פיזית.

^[16]SOX

- תעשייה: חברות ציבוריות וסחירות.
 - רגולטור: SEC (ועדה בסנאט האמריקאי).
 - תקציר: אומנם התקנות נחקקו בארה"ב, אך בארץ הרשות לניירות ערך פירסמה את הוראות תקנות ניירות הערך בשנת 2009^[17] על מנת לגבש חוקים דומים גם בישראל.
- סעיף הבקרה הרלוונטי לאבטחת מידע הינו ה-ITGC (IT General Contols) - סעיף אשר בוחן את יכולת התאגיד לשמור, לאחסן ולאחזר מידע במערכות שיש להן נגיעה לדיווח כספי. כמו כן, ישנה בחינה מדוקדקת של היכולת להפרדת תפקידים בגישה ובשימוש במערכת.

אז למה לי פוליטיקה עכשיו? / על רגולציה ואבטחת מידע

www.DigitalWhisper.co.il

[18] PCI-DSS

- תעשייה: חברות כרטיסי האשראי.
- רגולטור: PCI-SSC (מועצה המורכבת מ-5 שחקניות עיקריות בעולם כרטיסי האשראי).
- תקציר: הגרסא האחרונה לתקן יצאה בסוף 2013 (PCI-DSS 3.0). בבסיסו, התקן מדבר על הצפנת המידע הקשור בתשלומים ע"י כרטיסי אשראי, העברת הנתונים באופן מאובטח ושמירה על תיעוד מתאים. השינויים ^[19] בתקן החדש הינם הגברת המודעות (Awareness), גמישות ביישום התקן, חלוקת האחריות על היישום ומבהיר נקודות שהיו מעורפלות בתקן הישן.

[20] HIPAA

- תעשייה: ארגוני בריאות בארה"ב.
- רגולטור: משרד הבריאות האמריקני.
- תקציר: בשנת 1996 הועברה תקינת ה-HIPAA, כאשר החלק השני בתקינה עסק בין היתר בשמירת פרטיות החולים בפן האלקטרוני ^[21]. בין היתר, ארגוני הבריאות בארה"ב מחוייבים בהצפנת נתוני הלקוחות, תיעוד נתוני הקונפיגורציה של הרשתות ויצירת מדיניות אבטחת מידע, התקנת מערכות למניעת חדירה (IDS\IPS) והקמת מערך ניהול סיכונים טכנולוגי.

ISO27001

- תעשייה: כל גוף המעוניין להיות מוסמך בתחום אבטחת המידע.
- מסמך: מכון התקינה הבינלאומי (ISO).
- תקציר: גופים אשר מעוניינים בהסמכה אשר מציינת כי תחום אבטחת המידע בארגון עומד בתקינה הבינלאומית פונים למכון התקינה הבינלאומי לצורך בדיקה. בהתאם למספר קריטריונים ^[22] הגוף הבודק מבצע בחינות תקופתיות על מנת לאשר את ההסמכה. התקן הראשוני שיצא ב-2005 הוחלף בשנה שעברה בתקן מעודכן.

[23] חוק הגנת הפרטיות

- תעשייה: כל בעל מאגר מידע.
- רגולטור: רשם מאגרי המידע ומשרד המשפטים.
- תקציר: בשנת 1981 הוחלט על יצירת חוק להגנה על פרטיותו של אדם. החוק מגדיר מהו מידע רגיש בעיני המחוקק, כיצד ניתן להגיש בקשה לרשום מאגר מידע וכן קובע בסעיפים כלליים באילו תנאים יש למנות נציג אבטחת מידע למאגרים ועל מי חלה האחריות בתחום (בעל המידע ומנהל מאגר המידע).

[\[24\]](#) הוראה 357

- תעשייה: בנקאות בישראל.
- רגולטור: המפקח על הבנקים.
- תקציר: על מנת לשמור על המידע הפיננסי הרגיש של לקוחות הבנקים, פירסם המפקח רגולציה שמסדירה נושאים רבים ביניהם בנקאות בתקשורת, ניהול סיסמאות והצפנה, העברת מידע בדואר אלקטרוני וכן חלוקה לרמות פעילות הלקוח.

[\[25\]](#) הוראה 257

- תעשייה: ביטוח וגופים מוסדיים בישראל.
- רגולטור: הממונה על שוק ההון, ביטוח וחסכון.
- תקציר: באופן דומה לתחום הבנקאות ומתוך הסתמכות על ההוראה הנ"ל, הממונה על שוק ההון פרסם הוראה לגבי הגופים המוסדיים. התחומים של ההוראות חופפים, אך באופן כללי ניתן לשים לב כי הוראה זו הינה מפורטת ומחמירה יותר מאשר זו המקבילה לה בבנקאות.

סיכום

עולם הרגולציה הוא גם רחבי (במובן של מספר התעשיות המפוקחות) וגם עמוק (הרגולציות דנות לעתים בתכנים פרטניים), ועם זאת הוא מאוד דינמי בשל התפתחות הטכנולוגיה והתעשיות עצמן. תמיד יהיו פערים בין המצב בשטח לבין הרגולציה, וכדי להקל על כך הרגולטורים משאירים גם מקום לפרשנות במקומות מסוימים ברגולציות; צריך לזכור, שיש מקרים בהם גופי הפיקוח בארגון שאחראים על סגירת פערים אלה עושים את עבודתם נאמנה, אך יש גם מקרים בהם הם מתרשלים ומטשטשים את המצב האמיתי; לא צריך לקחת את הדיווחים על הטיפול בפערים כתורה מסיני ורצוי גם לבדוק באמצעי נוסף (גוף פיקוח אחר לצורך בקרה כפולה או לוודא בעזרת ה-CISO).

אחרי הכל, גם גופי פיקוח יכולים להיות לא אמינים [\[26\]](#). רצוי ואף מומלץ להשתמש במספר כללי אצבע בעת בחירת גוף פיקוח חיצוני [\[27\]](#).

אודות

שמי דניאל ליבר, ואפשר להגיד שאני אדם סקרן. אני נמצא בתחום אבטחת המידע (יעוץ, פיתוח מאובטח, ארכיטקטורה, רגולציה, PT ו-א"מ תפעולית) מספר שנים טובות כשאת צעדי הראשונים עשיתי בגיל יחסית מאוחר. עם זאת, אני מוצא את העולם הזה מרתק ומתפתח.



מקורות

1. http://en.wikipedia.org/wiki/Enron_scandal#Rise_of_Enron
2. <http://en.wikipedia.org/wiki/EnronOnline#EnronOnline>
3. http://en.wikipedia.org/wiki/Enron_scandal#Financial_audit
4. <http://content.time.com/time/interactive/0,31813,2013797,00.html>
5. <http://en.wikipedia.org/wiki/File:EnronStockPriceAug00Jan02.jpg>
6. <http://seclists.org/isn/2002/Jun/87>
7. <http://www.nytimes.com/2002/07/07/us/senate-panel-says-enron-s-board-could-have-stopped-high-risk-practices.html>
8. http://en.wikipedia.org/wiki/MCI_Inc.#Accounting_scandals
9. http://en.wikipedia.org/wiki/EnronOnline#California.27s_deregulation_and_subsequent_energy_crisis
10. http://en.wikipedia.org/wiki/Sarbanes-Oxley_Act
11. <http://www.p2080.co.il/go/p2080h/files/9062696158.ppt>
12. <http://www.themarket.com/technation/1.1607776>
13. http://he.wikipedia.org/wiki/%D7%A4%D7%A8%D7%A9%D7%AA_%D7%94%D7%94%D7%90%D7%A7%D7%A8_%D7%94%D7%A1%D7%A2%D7%95%D7%93%D7%99
14. <http://www.this.co.il/Company/itonut/regolation.aspx>
15. <http://www.isaca.org/chapters1/israel/events/Documents/%D7%9E%D7%A6%D7%92%D7%95%D7%AA%20%D7%9B%D7%A0%D7%A1%20%D7%A9%D7%A0%D7%AA%D7%99%202010/234%20yossi%20gavish.pdf>
16. <http://www.gpo.gov/fdsys/pkg/PLAW-107publ204/html/PLAW-107publ204.htm>
17. http://www.isa.gov.il/Download/IsaFile_3607.pdf
18. https://www.pcisecuritystandards.org/documents/PCI_DSS_v3.pdf
19. https://www.pcisecuritystandards.org/documents/DSS_and_PA-DSS_Change_Highlights.pdf
20. <http://www.gpo.gov/fdsys/pkg/PLAW-104publ191/html/PLAW-104publ191.htm>
21. <http://www.hhs.gov/ocr/privacy/hipaa/understanding/consumers/privacy-security-electronic-records.pdf>
22. <http://www.isaca.org/chapters3/Atlanta/AboutOurChapter/Documents/ISO%2027001.pdf>
23. http://www.nevo.co.il/law_html/Law01/087_001.htm
24. <http://www.boi.org.il/he/BankingSupervision/SupervisorsDirectives/DocLib/357.pdf>
25. <http://ozar.mof.gov.il/hon/2001/mosdiym/memos/2006-9-06.pdf>
26. <https://www.brighttalk.com/webcast/188/39117>
27. <http://www.sans.org/reading-room/whitepapers/analyst/choose-qualified-security-assessor-34920>

אז למה לי פוליטיקה עכשיו? / על רגולציה ואבטחת מידע

www.DigitalWhisper.co.il



דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-52 של Digital Whisper, אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב- למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה אבודות כדי להביא לכם את הגליון.

אנחנו מחפשים כתבים, מאיירים, עורכים ואנשים המעוניינים לעזור ולתרום לגליונות הבאים. אם אתם רוצים לעזור לנו ולהשתתף במגזין Digital Whisper - צרו קשר!

ניתן לשלוח כתבות וכל פניה אחרת דרך עמוד "צור קשר" באתר שלנו, או לשלוח אותן לדואר האלקטרוני שלנו, בכתובת editor@digitalwhisper.co.il.

על מנת לקרוא גליונות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המגזין:

www.DigitalWhisper.co.il

"Talkin' bout a revolution sounds like a whisper"

אם הכל יעבור כשורה, הגליון הבא ייצא ביום האחרון של חודש יולי.

אפיק קסטילאל,

ניר אדר,

30.06.2014