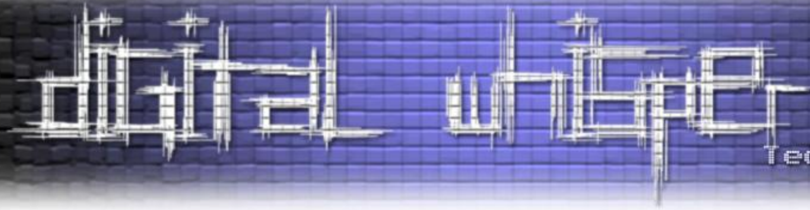


# על המימוש הפנימי של אובייקטים וטיפוסים ב-NET.

נכתב ע"י סשה גולדשטיין

## הקדמה

אם מאמינים ל-TIOBE Programming Community Language Index, אז C# ו-Visual BASIC חולשות על למעלה מ-12% מהקוד שנכתב בעולם, מה שממקם את NET. במקום השלישי והמכובד אחרי C ו-Java. כיוון ש-NET. היא סביבה מנוהלת, שבה הזיכרון, מבנה האובייקטים, ואפילו כתובות ומצביעים הם דברים שהמפתחים "לא מתעסקים בהם", קיים מעט מידע יחסית על המימוש הפנימי של NET. והקומפוננטות המרכיבות אותה.

במאמר זה אתחיל לסקור את המבנה הפנימי של אובייקטים בערימה המנוהלת (GC heap) ואעמוד על המבנה הפנימי של טיפוסים NET.-יים. לפני הכל, אסקר בקצרה את מנגנון בטיחות הטיפוסים של NET., וחולשות אבטחה שהתגלו במנגנון זה בעבר.

## בטיחות טיפוסים

הפרדיגמה הבסיסית של פיתוח ב-NET. היא שהקוד מוכרח להיות בטוח-טיפוסים (type-safe). כלומר, אין אפשרות - למעט במקרים חריגים וע"י שימוש במילות מפתח מיוחדות - לייצר מצבים לא בטוחים שבהם מערכת הטיפוסים נפגעת. למשל, לא ניתן ב-NET. לייצר סיטואציה שבה מבצעים השמה של מספר במצביע לאובייקט, ואז משתמשים במצביע הזה כדי לקרוא למתודות על אובייקט "מזויף".

יש מספר מצבים שבהם המתכנת - מרצונו וביודעין - מקבל החלטה לכתוב קוד שאינו בטוח-טיפוסים (עם זאת, רוב הקוד שנכתב ב-NET. היום הוא בטוח-טיפוסים ודורש אמצעים חיצוניים כדי לייצר חולשות). להלן מספר דוגמאות למצבים כאלה:

1. ניתן להשתמש במילות המפתח unsafe ו-fixed על מנת לעבוד ישירות עם מצביעים ב-C#. לדוגמא:

```
unsafe static void CorruptMemoryAtRandom(Random rng) {
    int* p = (int*)rng.Next(1<<16, 1<<31);
    *p = rng.Next(int.MinValue, int.MaxValue);
}
```

2. ניתן להשתמש במתודות של המערכת, כגון Marshal.PtrToStructure ו/או Marshal.StructureToPtr כדי לכתוב ולקרוא מקומות שרירותיים בזיכרון. למשל:

```
Marshal.StructureToPtr(DateTime.Now, new UIntPtr(0x0DEADC0W));
```

3. ניתן לקרוא לקוד שכתוב ב-C או כל שפה לא מנוהלת אחרת, שאינו מוגבל ביכולתו להשחית את הזיכרון.
4. ניתן לבנות union שבו מספר שדות חופפים המאפשרים עקיפה של מנגנון בטיחות הטיפוסים. למשל:

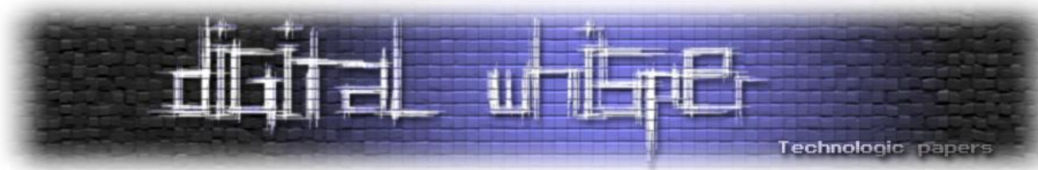
```
class Union1 { public UIntPtr Address; }
class Union2 { public object Ref; }

[StructLayout(LayoutKind.Explicit, Pack=1)]
struct Foo
{
    [FieldOffset(0)] public Union1 U1;
    [FieldOffset(0)] public Union2 U2;
}

Foo f = new Foo();
f.U2 = new Union2();
f.U2.Ref = new object();
//Now f.U1.Address can be manipulated to point outside of the heap
etc.
```

כאמור, תוכניות שלא עושות שימוש באמצעים אלה, אינן פגיעות בברירת מחדל. יש צורך בחולשות חיצוניות (כגון באג במנגנון בטיחות הטיפוסים של NET.) על מנת לתקוף תוכניות כאלה.

## דוגמאות לחולשות אבטחה ב-NET.

להלן שתי דוגמאות של חולשות אבטחה ב-NET. - שתיהן ניתנות לניצול מרחוק באמצעות הפעלת קוד דרך הדפדפן (למשל, באמצעות Silverlight), ומאפשרות בריחה מארגז החול של NET. בתוך הדפדפן.

המתודה `Marshal.AllocHGlobal` מאפשרת לתוכנית NET.-ית להקצות זיכרון מערימות של Win32. מתודה זו נמצאת בשימוש רחב בתוך ה-NET Framework. עצמו, בעיקר באזורים הנמצאים על התפר בין Windows לבין NET. עצמה. באפריל השנה מיקרוסופט הוציאו את התיקון [MS12-025](MS12-025) שאחת החולשות שהוא מתקן היא ב-System.Drawing.dll (ספריה האחראית על ציור באמצעות GDI), המאפשרת integer



overflow בהעברת פרמטר ל-Marshal.AllocHGlobal. כאשר הפונקציה מקבלת פרמטר שלילי, היא נכשלת בהקצאת הזיכרון, אך הקוד הקורא ב-System.Drawing.dll אינו בודק את תוצאת ההקצאה ומשתמש בה באופן המשחית את הזיכרון.

ביוני 2011 מיקרוסופט הוציאו את התיקון MS11-039 שמתקן חולשה במחלקה Socket. המתודות Send ו-Receive של מחלקה זו מקבלות IList<ArraySegment<byte>>, המתאר רשימה של חוצצים לשליחה או קבלה. ArraySegment<byte> הוא טיפוס פשוט העוטף "חלון" חלקי למערך - למשל, אם קיים מערך של 1,000 בתים, ניתן ליצור new ArraySegment<byte>(array,30,50) המגדיר חלון בגודל 50 בתים המתחיל במקום ה-30 במערך. למרות שהבנאי של ArraySegment מוודא שלא יוצרים חלון בעל היסט שלילי לתוך המערך, ניתן לשנות את התכונות Offset ו-Count של האובייקט לאחר בנייתו. כיוון שהמתודות Send ו-Receive הנ"ל לא ביצעו ולידציה של הסגמנטים שהועברו להן, ניתן היה להשתמש בהן כדי להשחית את הזיכרון, ואף ליצור מערך (byte[]) המאפשר גישה לאינדקסים שליליים.

## קטגוריות טיפוסים ב-.NET

בדומה לסביבות מנוהלות אחרות כגון JVM, גם ב-.NET הטיפוסים מתחלקים לשני סוגים: value types ו-reference types. המבנה הפנימי שלהם בזיכרון (memory layout) לא ניתן כמעט לשליטה על ידי המתכנת, אולם הבנה מדויקת של המבנה הזה חשובה הן לביצועים והן לניצול חולשות בקוד הכתוב ב-.NET; כך למשל, כדי לנצל heap overflow יש להבין היטב את מבנה ה-heap ומתי מתרחשים בו שינויים שאפשר לנצל לטובתנו.

סוג הטיפוסים הראשון, value types, נועד לאובייקטים קטנים שנוצרים בקצב גבוה ומועברים לפי ערך (by value) בין פונקציות. למשל, מספרים, תאריכים, תווים - הם כולם value types ב-.NET., וניתן להגדיר גם value types נוספים באמצעות מילת המפתח struct ב-C#. על value types חלות מגבלות רבות: לא ניתן לרשת מהם, לא ניתן להגדיר בהם מתודות וירטואליות, לא ניתן להשתמש בהם כאובייקט סנכרון, ועוד.

לעומתם, reference types הם אובייקטים לכל דבר, התומכים בירושה, רב-צורניות (polymorphism), סנכרון, ושירותים רבים נוספים. על מנת לאפשר זאת, המימוש הפנימי שלהם עשיר ומורכב (יחסית ל-C++ למשל), ומשתנה בין גרסאות בהרף עין. אדון כאן במימוש של .NET 2.0 ו-.NET 4.0. בגרסאות ה-32 ביט שלהן, ואשאיר את יתר הגרסאות לקורא.

## מבנה Reference Types ב-NET 2.0.

לצורך הבנת המבנה של אובייקט בזיכרון, נשתמש בדוגמת המחלקה הבאה:

```
class Employee
{
    public Employee() {}
    public virtual void Work() {}
    public void Sleep() {}

    private uint id = 0xAABBCCDD;
    private string name = "David";
}
```

המבנה הפנימי של אובייקט Employee בזיכרון ייראה כך:

```
MEMORY 1
Address: 0x02677718
0x02677718  00000000 00000021 00000003 00000001  ....!...........
0x02677728  00000000 00159e44 00159e60 712f0440  ....Dž..`ž..@./q
0x02677738  00000000 712eec44 00000000 00000000  ....Dì.q........
0x02677748  00159e60 02677558 aabbccdd 00000000  `ž..Xug.ÝÌ»ª....
0x02677758  00000000 00000000 00000000 00000000  ................
```

השדה הראשון (המסומן באדום) מצביע לטבלת המתודות (method table) של הטיפוס. שם נמצאים המימושים של כל המתודות הווירטואליות של המחלקה Employee. השדה השני מצביע למחרוזת "David", והשדה השלישי הינו המספר הקל לזיהוי.[1] לבסוף, השדה הירוק, במפתיע, גם הוא חלק מהאובייקט - על אף שהוא נמצא בהיסט של ארבעה בתים מהכתובת שלו.

בטבלת המתודות יש דברים נוספים מלבד המצביעים למתודות, אולם זה הדבר שנתמקד בו כעת:

```
MEMORY 1
Address: 0x00159e60
0x00159E60  00080000 00000010 00060011 00000005
0x00159E70  712f0944 00158eec 00159ea0 005f71b0
0x00159E80  00000000 00000000 71246a90 71246ab0
0x00159E90  71246b20 712b7700 0015c578 0015c570
0x00159EA0  00000080 035c5d14 00000000 00000000
```

[1] כפי שראינו קודם, ניתן להשיג שליטה על סדר השדות באובייקט, אולם מרבית המפתחים לא עושים זאת. ואכן במקרה זה, NET. בחרה לשנות את סדר השדות באובייקט לעומת הסדר שבו הם הוגדרו במחלקה עצמה.

החלקים המסומנים באדום הם המצביעים למתודות של האובייקט. ארבעת המצביעים הראשונים הם למתודות וירטואליות של System.Object, שהמחלקה שלנו לא דורסת אך עדיין מקבלת בירושה. שני המצביעים לאחר מכן הם מצביע למתודה Work ולבנאי של המחלקה (שימו לב שהמתודה הלא-וירטואלית Sleep לא קיבלה כניסה בטבלה, וזאת משום שניתן לקרוא לה ללא שימוש בטבלה, כפי שנראה בהמשך).

נתבונן ב-disassembly של הנקודה בקוד בה אנו קוראים למתודה Work:

```
mov ecx, dword ptr [ebp-44] ; read objref from the stack to ECX
mov eax, dword ptr [ecx]     ; dereference to obtain method table
pointer
call dword ptr [eax+38]      ; call virtual method through method table
```

קל לראות שבהיסט 0x38 מתחילת הטבלה נמצא המצביע 0x0015c578. כדי להשתכנע שאכן מדובר במצביע למתודה Work, ניתן להשתמש למשל בפקודה u! של SOS (הרחבה ל-WinDbg הנועדה לניתוח תוכנית הכתובות ב-NET.) שנשתמש בה גם בהמשך:

```
0:008> !u 0015c578
Unmanaged code
0015c578 e9d3534800      jmp     005e1950

0:008> !u 005e1950
Normal JIT generated code
DigWhisper.Employee.Work()
Begin 005e1950, size 2e
005e1950 55              push    ebp
005e1951 8bec            mov     ebp,esp
…more code snipped
```

כלומר, בטבלה יש מצביע לטרמפולינה שמביאה אותנו בסופו של דבר למתודה Work. פשר הטרמפולינה נעוץ בעובדה שהמתודה Work עוברת הידור רק בזמן ריצה, ולכן הטרמפולינה מוחלפת בפעם הראשונה שהמתודה נקראת (פרטים נוספים על כך אולי נביא במאמר עתידי על עבודתו של ה-JIT). בהקשר זה יש לציין שגם הטרמפולינה וגם הקוד עצמו, במידה ועבר הידור בזמן ריצה, נמצאים באזור זיכרון בעל ההגנה PAGE_EXECUTE_READWRITE, גם במערכות שתומכות ב-DEP. המשמעות היא, למשל, שגם אם הערימה עצמה מוגנת מפני הרצה, ניתן לנסות לנצל חולשת כתיבה לזיכרון על מנת לדרוס קוד או טרמפולינה כפי שראינו כרגע.

על מנת לקרוא למתודה לא וירטואלית, אין צורך להשתמש בטבלת המתודות, כיוון שלא תיתכן רב-צורניות. למעשה, המהדר יכול לקבוע מראש איזו מתודה תקרא, ולצרוב את כתובתה לתוך התוכנית.

למעשה, גם כאן יש צורך בטרמפולינה לצורך הידור בזמן ריצה, אבל כתובת הטרמפולינה ידועה ולא תלויה בטיפוס:

```
mov ecx, dword ptr [ebp-44]  ; read objref from the stack to ECX
cmp dword ptr [ecx], ecx     ; verify that ECX is a valid address
call dword ptr ds:[001c34d8] ; call through static trampoline location

0:000> dd 001c34d8 L1
001c34d8  00690218

0:000> !u 00690218
Normal JIT generated code
DigWhisper.Employee.Sleep()
Begin 00690218, size 1b
00690218 55              push    ebp
00690219 8bec            mov     ebp,esp
…more code snipped
```

על מנת להציג את מבנה טבלת המתודות באופן קריא ונוח, המאפשר גם למצוא פרטים נוספים לגבי המחלקה שהטבלה שייכת אליה, ניתן להשתמש בפקודה !DumpMT של SOS. לאחר מכן, כדי להגיע לפרטים נוספים ניתן להשתמש בפקודות כגון !DumpMD, !DumpClass ואחרות. הנה, למשל, הפלט של !DumpMT על טבלת המתודות של המחלקה שראינו קודם:

```
0:000> !dumpmt -md 001c34e8
EEClass: 001c1600
Module: 001c2f2c
Name: DigWhisper.Employee
mdToken: 02000005  (D:\DigWhisper\DigWhisper.exe)
BaseSize: 0x10
ComponentSize: 0x0
Number of IFaces in IFaceMap: 0
Slots in VTable: 7


--------------------------------------
MethodDesc Table
   Entry MethodDesc      JIT Name
71246a90   710c4938   PreJIT System.Object.ToString()
71246ab0   710c4940   PreJIT System.Object.Equals(System.Object)
71246b20   710c4970   PreJIT System.Object.GetHashCode()
712b7700   710c4994   PreJIT System.Object.Finalize()
```

```
00690248    001c34c8       JIT DigWhisper.Employee.Work()
006901c8    001c34c0       JIT DigWhisper.Employee..ctor()
00690218    001c34d0       JIT DigWhisper.Employee.Sleep()
```

לפני שנוכל לסכם את מבנה האובייקט בזיכרון, נותר רק להזכיר שוב את השדה המופיע לפני תחילת האובייקט בזיכרון. שדה זה, שנקרא object header word, משמש לשתי מטרות מרכזיות: (1) שמירת ביטים שונים הנוגעים למצב האובייקט, למשל האם אוסף הזבל כבר ביקר אובייקט זה במהלך פעולת האיסוף האחרונה; (2) קישור בין האובייקט לבין מבנה נתונים פנימי שנקרא sync block, שמכיל פרטים נוספים על האובייקט שאין להם מקום בתחילתו.

השימוש העיקרי ב-sync block הוא לקישור האובייקט לאובייקט סנכרון. ב-.NET, לכל אובייקט ניתן להצמיד אוטומטית מנגנון סנכרון באמצעות מנגנון המכונה Monitor, והנגיש לשפות העלית באמצעות מילות מפתח. למשל, להלן דוגמת קוד ב-C# המשתמשת באובייקט Account כמנגנון סנכרון באמצעות מילת המפתח lock:

```
class Account
{
    private decimal balance;
    public void Deposit(decimal amount)
    {
        lock (this)
        {
            balance += amount;
        }
    }
}
```

כאשר האובייקט משוייך למנגנון סנכרון, הקישור מתבצע באמצעות ה-object header word. חלק מהביטים שלו מכילים מספר, שהוא אינדקס לתוך טבלת sync blocks המנוהלת על ידי .NET - ושם שמור מנגנון הסנכרון עצמו. למשל, כאשר 0x272a77c היא כתובת של אובייקט Employee נעול, אנו רואים ש-1 הוא האינדקס לתוך הטבלה הנ"ל שמשתמשים בו עבור אובייקט זה:

```
0:004> dd 0272a77c-4 L4
0272a778  08000001 003f3570 0272a56c aabbccdd
```

כמובן, ישנה פקודה של SOS שנקראת SyncBlk!, המאפשרת לצפות בכל ה-sync blocks התפוסים. כך גם ניתן להגיע לכתובת של ה-sync block עצמו, ואף לתחילת הטבלה:

```
0:004> !syncblk
Index SyncBlock MonitorHeld Recursion Owning Thread Info  SyncBlock
Owner
    1 005089bc            3         1 004d8028  2174   0   0272a77c
Employee
-----------------------------
```

כך ניתן לראות, למשל, שמנגנון הסנכרון ש-NET. משתמשת בו למימוש Monitor הוא למעשה event של Win32:

```
0:004> dd 005089bc L8
005089bc  00000003 00000001 004d8028 00000001
005089cc  80000001 000001c4 0000000d 00000000

0:004> !handle 000001c4 f
Handle 1c4
  Type           Event
  Attributes     0
  GrantedAccess  0x1f0003:
         Delete,ReadControl,WriteDac,WriteOwner,Synch
         QueryState,ModifyState
  HandleCount    2
  PointerCount   4
  Name           <none>
  Object Specific Information
    Event Type Auto Reset
    Event is Waiting
```

ל-sync block יש גם שימושים נוספים, למשל שמירת קוד הגיבוב (hash code) של האובייקט כאשר אין מקום לכך ב-object header word, ואחרים.

## מבנה Reference Types ב-NET 4.0.

השינוי המרכזי שהתרחש ב-NET 4.0. במבנה אובייקטים בזיכרון קשור בהפרדת טבלת המתודות ושבירתה למספר חלקים, על מנת לחסוך במקום כאשר טבלאות כאלה משוכפלות בין טיפוסים הדורסים מעט מאוד מתודות וירטואליות של מחלקות האב.

למשל, כאשר הוספנו מספר רב של מתודות וירטואליות למחלקה Employee, וכתבנו מחלקה אחרת היורשת ממנה, טבלת המתודות של המחלקה היורשת נראתה כך:

```
0:006> dd 00343a34
00343a34  01000200 00000010 00064188 0000000d
00343a44  00343994 00342e7c 00343a78 003415fc
00343a54  0034c0ad 00000000 003439c4 00343a64
00343a64  0034c085 0034c09d 0034c0a1 0034c0a5
00343a74  0034c0a9 00000080

0:006> dd 00343994
00343994  01000200 00000010 00054188 00000009
003439a4  50c88194 00342e7c 003439e8 003415a8
003439b4  00730388 00000000 003439c4 003439e4
003439c4  50bd5450 50bc06b0 50bc0270 50bc0230
003439d4  00730408 0034c079 0034c07d 0034c081
003439e4  0034c085 00000080
```



כאן, טבלת המתודות של המחלקה היורשת נמצאת למעשה בשני מקומות: חלק מהמצביעים למתודות נמצאים בטבלה של מחלקת האב, וחלק בטבלה של המחלקה היורשת. (הסיבה לשינוי, כמובן, היא הרצון לחסוך במקום כאשר טוענים טבלאות מתודות של מחלקות בעלות הרבה מאוד מתודות וירטואליות, שרק חלק קטן מהן נדרסות במחלקות יורשות.)

## סיכום

במאמר זה סקרנו בצורה בסיסית את המבנה בזיכרון של אובייקטים וטיפוסים NET.-יים. מטבע הדברים זו הצגה חלקית: פרטים רבים נוספים ניתן למצוא בספרים כגון Advanced .NET Debugging (מעודכן ל- NET. 3.5 בלבד) וספרי החדש Pro .NET Performance הצפוי לצאת באוגוסט השנה.

כפי שציינתי בתחילה, ניצול חולשות כגון heap overflow בערימה המנוהלת דורש הבנה מעמיקה יותר של מבנה הערימה וניהולה, מעבר לפרטים שראינו במאמר זה לגבי המבנה של אובייקטים אינדיבידואליים בה. אם יהיה עניין בכך, במאמרים הבאים נוכל להיכנס לפרטים נוספים, למשל לגבי אופן פעולתו של אוסף הזבל (garbage collector) ואופטימיזציות של ה-JIT.

משה גולדשטיין הוא ה-CTO של קבוצת סלע, חברת ייעוץ, הדרכה ומיקור חוץ בינלאומית עם מטה בישראל. משה אוהב לנבור בקרביים של Windows וה-CLR, ומתמחה בניפוי שגיאות ומערכות בעלות ביצועים גבוהים. משה הוא מחבר הספר Pro .NET Performance, ובין היתר מלמד במכללת סלע קורסים בנושא Windows Internals ו-CLR Internals. בזמנו הפנוי, משה כותב בלוג על נושאי פיתוח שונים.