מבחן, מגיש אלעד פישר, 213924624:

1) אם נסתכל בתוך הTLS אפשר לראות שהוא לא מאוד תמים כמו שאני הייתי חושב בהתחלה, אחרי הכל IDA מעיר לנו על דבר די זדוני שהוא עושה:

```
; Attributes: bp-based frame

TlsCallback_0_0 proc near

var_108= dword ptr -108h
var_4C= byte ptr -4Ch
var_3C= dword ptr -3Ch
pcbData= dword ptr -30h
pvData= dword ptr -24h
var_18= byte ptr -18h
var_C= byte ptr -0Ch
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 10Ch
push    ebx
push    esi
push    edi
lea     edi, [ebp+var_4C]
mov     ecx, 13h
mov     eax, 0CCCCCCCCh
rep stosd
mov     eax, ___security_cookie
xor     eax, ebp
mov     [ebp+var_4], eax
mov     ecx, offset unk_41C12C
call    sub_411343
mov     esi, esp
call    ds:IsDebuggerPresent
```

בתוך הTLS יש קוד זדוני שקורא לפונקציה שבודקת אם הדיבאגר פעיל, לצערי הTLS חשוב מידי כדי שאני אוכל להתעלם ממנו ולכן אני חייב להפעיל אותו כמו שצריך כי הוא גם חשוב להמשך הרצת התוכנית(למעשה כבר עכשיו אני יודע שהוא מאתחל משתנים מסויימים בזיכרון:

```
mov     eax, 1
imul    ecx, eax, 0
mov     byte_41A13C[ecx], 59h ; 'Y'
mov     eax, 1
shl     eax, 0
mov     byte_41A13C[eax], 68h ; 'h'
mov     eax, 1
shl     eax, 1
mov     byte_41A13C[eax], 65h ; 'e'
mov     eax, 1
imul    ecx, eax, 3
mov     byte_41A13C[ecx], 79h ; 'y'
mov     eax, 1
shl     eax, 2
mov     byte_41A13C[eax], 6Fh ; 'o'
mov     eax, 1
imul    ecx, eax, 5
mov     byte_41A13C[ecx], 31h ; '1'
mov     eax, 1
imul    ecx, eax, 6
mov     byte_41A13C[ecx], 2Ah ; '*'
mov     eax, 1
imul    ecx, eax, 7
mov     byte_41A13C[ecx], 3Ah ; ':'
mov     eax, 1
shl     eax, 3
mov     byte_41A13C[eax], 3Ah ; ':'
mov     eax, 1
imul    ecx, eax, 9
mov     byte_41A13C[ecx], 3Bh ; ';'
mov     eax, 1
imul    ecx, eax, 0Ah
mov     [ebp+var_108], ecx
cmp     [ebp+var_108], 0Bh
jnb     short loc_4119C7
```

כמו שאפשר לראות בקטן הוא מאתחל מחרוזת, אבל היא מחרוזת חשובה משום שעוד נשתמש בה להדפיס את הודעת ההצלחה...(למעשה נחזור לזה אחכ כי זה משומה לא מאותחל, אבל הפירוט יהיה על הסדר ויותר לקראת סוף המבחן).

הנקודה בקיצור שאי אפשר לדרוס את הקריאה לTLS או לבצע חזרה מיד בלי לשנות מלא קוד שקריטי לנו להמשך ההרצה ואני לא אוהב את זה ולכן הפיתרון הכי פשוט שיש זה לראות את הבדיקה עצמה:

```
call moeda2022.4B1343
mov esi,esp
call dword ptr ds:[<&IsDebuggerPresent>]
cmp esi,esp
call moeda2022.4B125D
test eax,eax
je moeda2022.4B18D3

moeda2022.004B18C2
 mov esi,esp
 push 0
 call dword ptr ds:[<&ExitProcess>]
 cmp esi,esp
 call moeda2022.4B125D

moeda2022.004B18D3
 lea eax,dword ptr ss:[ebp-C]
 mov dword ptr ss:[ebp-24],eax
 lea eax,dword ptr ss:[ebp-18]
 mov dword ptr ss:[ebp-30],eax
 mov esi,esp
 mov eax,dword ptr ss:[ebp-30]
 push eax
```

אז בTLS יש לנו קריאה בקטן לבדיקה האם הדיבאגר פעיל, אם הוא כן(הערך בEAX זה 1) אני אמשיך לקוד שיקרא ליציאה מהפרוסס והתוכנית, למעשה זה קצת יותר מבאס המנגנון הזה כי אם זה רק מקריס את התוכנית אפשר בקלות לראות לפי המחסנית קריאות איפה זה קרס ואז לעלות קצת לבדיקה.

בכל מקרה למעשה אני פשוט אשנה בית אחד בזיכרון:

כדי לדלג על הבדיקה אני מתכוון לקפוץ בכל מקרה על הקוד הבעייתי וככה גם אם התוכנית תזהה שיש דיבאגר פעיל היא תדלג על הקוד שאמור לזרוק אותנו מהתוכנית והיא תמשיך כרגיל...

ככה ייראה הקוד אחרי הפצ'פוץ':

```
004B18B9   E8 9FF9FFFF     call moeda2022.4B125D
004B18BE   85C0            test eax,eax
004B18C0   EB 11           jmp moeda2022.4B18D3
004B18C2   8BF4            mov esi,esp
004B18C4   6A 00           push 0
004B18C6   FF15 34B04B00   call dword ptr ds:[<&ExitProcess>]
004B18CC   3BF4            cmp esi,esp
004B18CE   E8 8AF9FFFF     call moeda2022.4B125D
004B18D3   8D45 F4         lea eax,dword ptr ss:[ebp-C]
004B18D6   8945 DC         mov dword ptr ss:[ebp-24],eax
```

ככה זה ייראה מבחינת הקוד עצמו, אפשר לראות את הבית שהשתנה כדי לעשות קפיצה לא מותנה באדום.

ככה זה ייראה בגרף:

```
mov esi,esp
call dword ptr ds:[<&IsDebuggerPresent>]
cmp esi,esp
call moedacracked.CD125D
test eax,eax
jmp moedacracked.CD18D3

moedacracked.00CD18D3
 lea eax,dword ptr ss:[ebp-C]
 mov dword ptr ss:[ebp-24],eax
 lea eax,dword ptr ss:[ebp-18]
 mov dword ptr ss:[ebp-30],eax
 mov esi,esp
 mov eax,dword ptr ss:[ebp-30]
 push eax
 mov ecx,dword ptr ss:[ebp-24]
```

כלומר לא משנה מה התוצאה, נקפוץ תמיד להמשך הקוד.

שאלה 2:

תפקידי המשתנים בIDA:

דבר ראשון זה למצוא את הMAIN. כדי לעשות את זה אני פשוט הלכתי תמיד לקריאה האחרונה עד שמצאתי את הבדיקה של הסקיוריטי קוקי ואז הלכתי לקריאה לפני זה שדחפה למחסנית את argC argV וכו' ואז הלכתי לקריאה של המיין, וככה הוא נראה:

```
push    ebp
mov     ebp, esp
sub     esp, 108h
push    ebx
push    esi
push    edi             ; char
lea     edi, [ebp+var_48]
mov     ecx, 12h
mov     eax, 0CCCCCCCCh
rep stosd
mov     eax, ___security_cookie
xor     eax, ebp
mov     [ebp+var_4], eax
mov     ecx, offset unk_41C12C
call    sub_411343
push    offset aWhatWouldYouLi ; "What would you like your test score to "...
call    printf
add     esp, 4
push    0Ah
lea     eax, [ebp+inputStr]
push    eax             ; char
push    offset Format   ; "%s"
call    scanF
add     esp, 0Ch
mov     [ebp+charsSum], 0
lea     eax, [ebp+inputStr]
push    eax             ; Str
call    j_strlen
add     esp, 4
000411DA0: main (Synchronized with Hex View-1)
```

כמובן שאפרות אחרת הייתה לחפש את המחרוזת של ...What would you אבל זה דרך אחרת ואם המחרוזת הייתה מוצפנת זה לא היה עובד כזה בקלות, בכל מקרה אני דבר ראשון סימנתי פונקציות קלט\פלט כי ככה קל יותר למצוא את המשתנים ומה התפקיד שלהם:

```
call    sub_411343
push    offset aWhatWouldYouLi ; "What would you like your test score to "...
call    printf
```

למשל אחרי הדחיפה יש הדפסה.

ויש בקשת קלט אחכ:

```
push    0Ah
lea     eax, [ebp+inputStr]
push    eax             ; char
push    offset Format   ; "%s"
call    scanF
```

ולכן ככה אפשר לסמן את המשתנה הזה בתור המחרוזת קלט(מהמשתמש).

לאחר מכן יש קריאה לSTRLEN ולכן יש לנו את המשתנה של אורך הקלט:

```
lea     eax, [ebp+inputStr]
push    eax             ; Str
call    j_strlen
add     esp, 4
mov     [ebp+inputStrLength], eax
```

ואחר כך יש השוואה עם משתנה שכל פעם עולה באחד עד שהוא מגיע לאורך של הקלט:

```
loc_411E17:
mov     eax, [ebp+counter(INDEX]
cmp     eax, [ebp+inputStrLength]
jge     short loc_411E2F
```

ההשוואה לאורך של הקלט.

```
mov     [ebp+counter(INDEX], 0
jmp     short loc_411E17
```

אתחול

```
loc_411E0E:
mov     eax, [ebp+counter(INDEX]
add     eax, 1
mov     [ebp+counter(INDEX], eax
```

קידום

ובגלל שכל זה קורה בלולאה מבחינת הקפיצות אז ברור שזה האינדקס ואפשר לקרוא לו i אבל לי יותר נוח counter.

```
mov     eax, [ebp+counter(INDEX]
movsx   ecx, [ebp+eax+inputStr]
add     ecx, [ebp+charsSum]
mov     [ebp+charsSum], ecx
jmp     short loc_411E0E
```

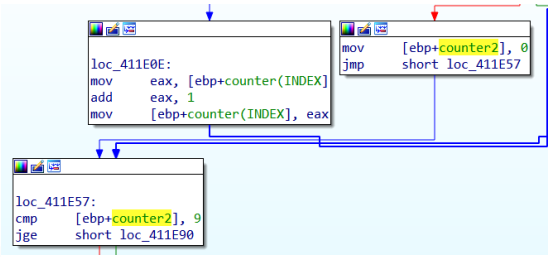
לאחר מכן אפשר לראות שסוכמים את הערכים של הasciiה לתוך משתנה אחר(שזה ההשמה היחידה אליו חוץ מהאתחול) ולכן נקרא לו charsSum כי הוא עושה סכום לכל הצ'ארים בקלט.

אחכ נשאר לי משתנה אחד שגם עושים לו קידום של 2 בכל פעם בכל סוף בלוק קוד:

```
mov     eax, [ebp+counter2]
movsx   ecx, byte_41A13C[eax]
xor     ecx, [ebp+charsSum]
mov     edx, [ebp+counter2]
mov     byte_41A13C[edx], cl
mov     eax, [ebp+counter2]
movsx   ecx, byte_41A13D[eax]
mov     edx, [ebp+charsSum]
add     edx, 1
xor     ecx, edx
mov     eax, [ebp+counter2]
mov     byte_41A13D[eax], cl
jmp     short loc_411E4E
```

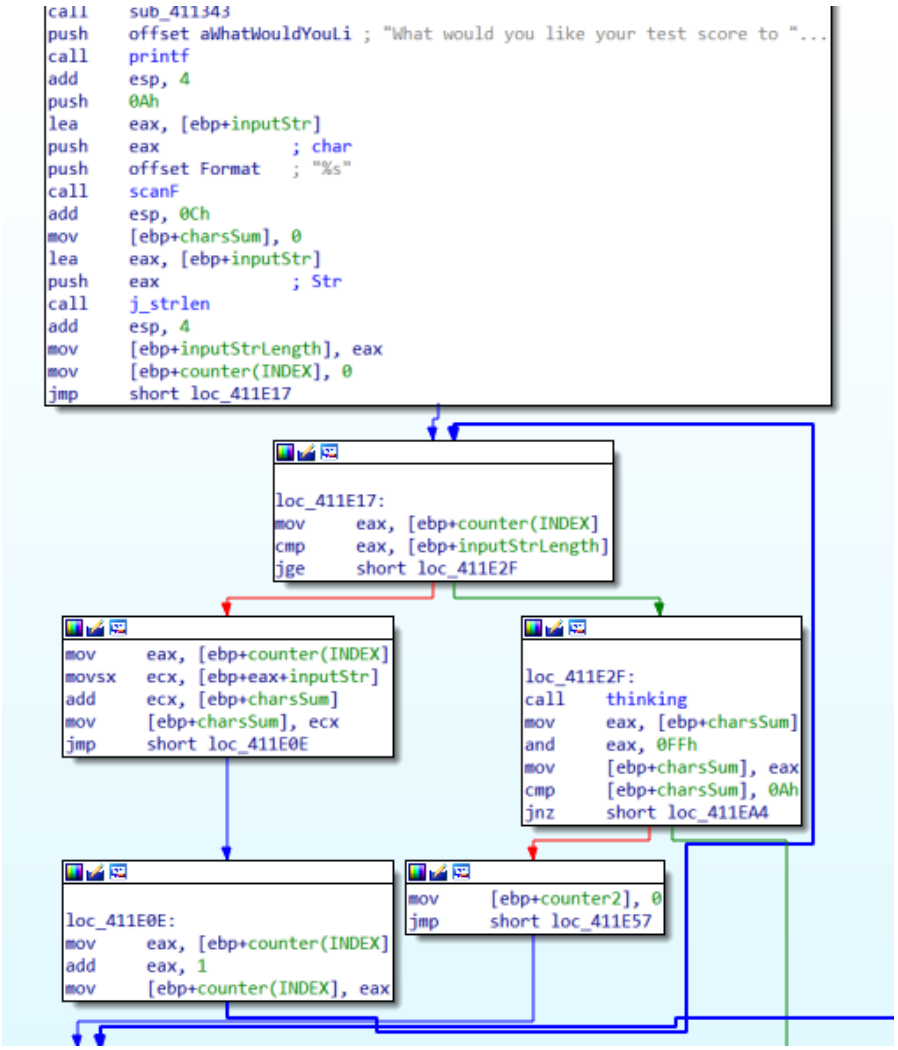
```
loc_411E4E:
mov     eax, [ebp+counter2]
add     eax, 2
mov     [ebp+counter2], eax
```

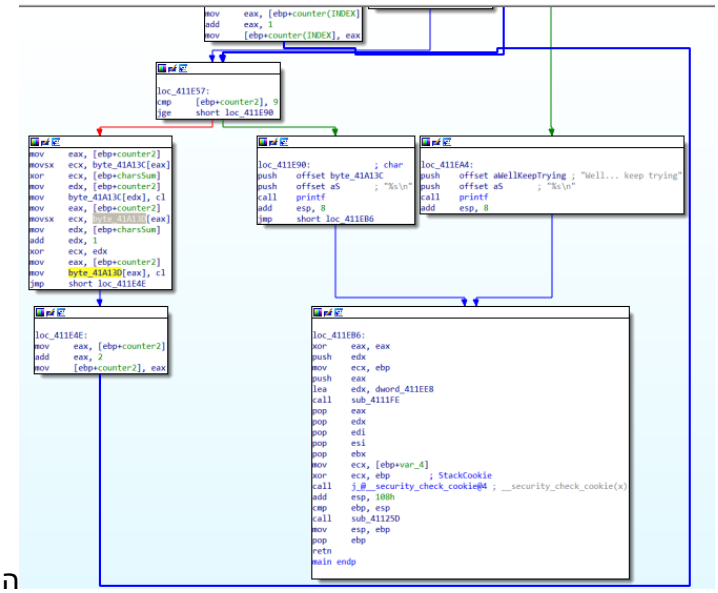
ונראה לפי האתחול ותנאי הסיום שזו לולאה שניה ולכן אני אקרא לזה counter2.



האתחול מימין למעלה והתנאי משמאל למטה.

הקוד של המיין בתמונה אחת:



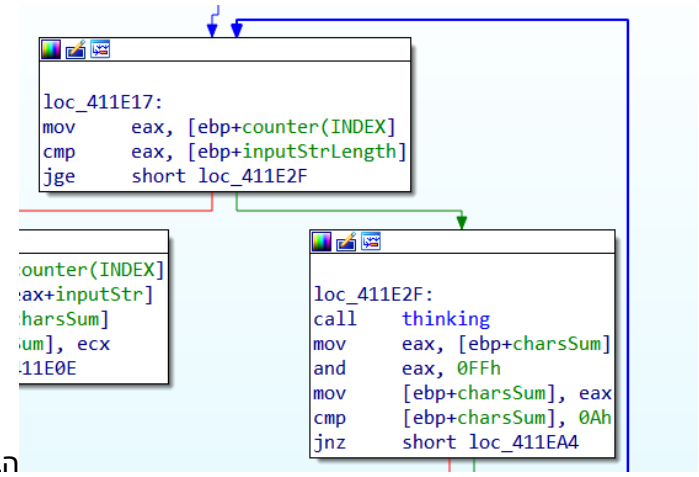


התמונה רק לקונטקסט,

בעיקרון כל הקוד מוסבר למעלה

עכשיו כאשר נתנו שמות להכל אפשר להתחיל לפענח את הקוד:

2ב. בגדול אפשר להבין מהר מאוד לפי הניתוח של המשתנים שיש את המשתנה שסוכם את הצ'ארים ושהוא משמעותי להמשך התוכנית, אז הבדיקה שעושים איתו אחרי הלולאה(בתוך הלולאה אין בדיקה איתו בכלל ולא אחת שהולכת להודעה של הכישלון בפרט) ולכן לפי הבדיקה איתו שהיא:



הבדיקה עצמה הבלוק הימני

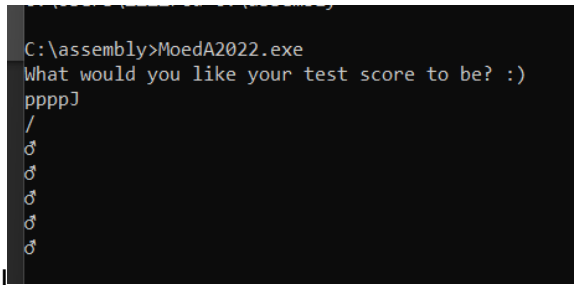
למטה. ולכן אפשר לראות שעושים פעולת and לוגית עם FF כלומר לוקחים רק את הבית הכי קטן, ואח"כ עושים השוואה לA0, אחרי ההשוואה אם זה לא שווה הולכים למקום שמדפיס את ההודעת כישלון ואם לא הולכים ללולאה השנייה שנראתה יותר מעניינת וגם איפה שאני רוצה ללכת.

לכן למעשה אני בתור משתמש צריך למצוא מחרוזת שהסכום ערכי ה asciiשל התווים שלה שווים ל 0a??כלומר אני צריך למצוא מחרוזת שהבית הכי קטן שלה יהיה שווה לA ולכן אני אתחיל לחשב מחרוזת שהסכום שלה יהיה 20Aאם כי זה שרירותי, גם אפשר 10Aאו כל דבר דומה.

אני אתחיל בלחסר תווים גדולים יחסית בעזרת מחשבון הקסא כדי לחפש כמה עוד אני צריך לחסר.

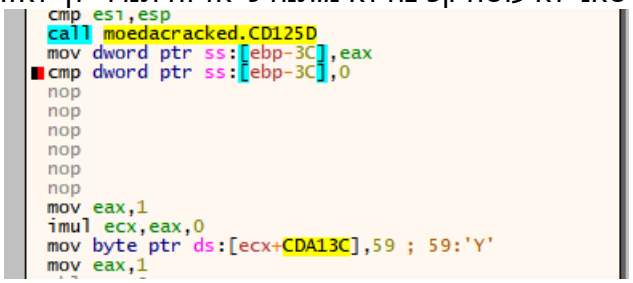
אני אקח את התו pכי הוא עם ערך 70 וזה טוב לי. אם אני מחסר 4 פעמים pמהסכום שלי אני רואה שנשאר לי 4Aשאני צריך למלא והרי לפי טבלת האסקיי שלי זה בדיוק המספר של התו J ולכן הסיסמה יכולה להיות מחרוזת שמורכבת מ4 פעמים pופעם אחת J. למעשה היה אפשר לקחת כל מחרוזת שאני רוצה באותה דרך ולחשב את הערך באותה דרך של חיסור של מספרים עד שאני מקבל את התווים שהסכום שלהם שווה למשהו שנגמר ע0A. דרך אחרת אני מניח יהיה להתחיל עם מחרוזת קבועה ולהוסיף תווים כדי למצוא משהו שנגמר עם המספר הנכון אם כי לדעתי זה מסובך יותר(וניסיתי ככה בהתחלה וזה פחות עבד).

הנה ההרצה שיש לי:

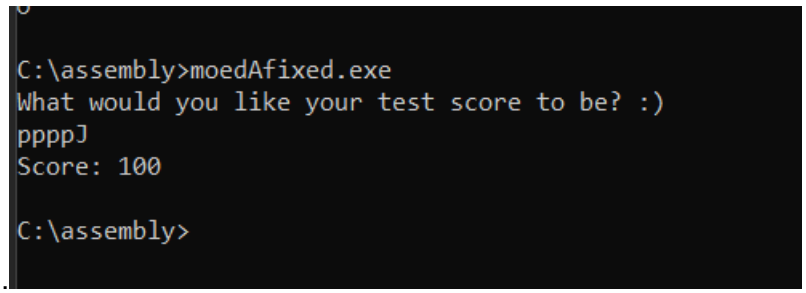


והרי זה לא הפלט הרצוי וגם זה לא נכון ואני כן עשיתי את הפצ'פוץ' כמו שצריך.

אבל פה מגיע מה שראיתי למעלה שיש את האתחול שכנראה לא התבצע טוב ולכן אם אני שם ברייקפוינט בX32 אני באמת רואה שאני מדלג על זה, וזה משומש בזיכרון, ולכן אני אפצ'פץ' את התנאי להיות nop כי מיד אחרי הקוד הזה מופיע האזור שאני רוצה להיכנס אליו, הסיבה שאני לא עושה קפיצה לא מותנה כי אז זה תמיד ילך לאזור של הלא אתחול.



ואחרי התיקון ושמירת השינויים באמת הסיסמה עובדת:



.

3.א.

עכשיו יש לכתוב את הקוד בהנדסה לאחור:

```python
def test():
    userInput = input("What would you like your test score to be? :)")
    sum=0

    for c in userInput:
        sum+=ord(c)

    if sum&255 == 10:
        print("Score: 100")
```

זה הקוד שמבצע הכל חוץ מהפענוח של ההצפנה, למעשה יש פה את הקלט\פלט בהתחלה, סכימה של הכל ואחרי זה בדיקה של הסכום אם הבית הכי קטן הוא A ובגלל שהכל פה עובד בצורה עשרונית אז FF מתוגם ל255, וA ל10. אבל אפשר לעשות הכל עם הקסא גם בפייתון, רק שאני לא רוצה להסתבך עם זה.

ועכשיו לפענוח מסר ההצלחה שזה סעיף ב:

דבר ראשון צריך להבין איך ההצפנה עובדת כי לא ממש היה חשוב עד עכשיו כי סך הכל ידעתי שזה שם לפי הקוד ואיפה שצריך ללכת, אבל זה לא שינה כדי למצוא את הסיסמה.

```
moeda2022fix.00E81E5D
mov eax,dword ptr ss:[ebp-44]
movsx ecx,byte ptr ds:[eax+E8A13C]
xor ecx,dword ptr ss:[ebp-20]
mov edx,dword ptr ss:[ebp-44]
mov byte ptr ds:[edx+E8A13C],cl
mov eax,dword ptr ss:[ebp-44]
movsx ecx,byte ptr ds:[eax+E8A13D]
mov edx,dword ptr ss:[ebp-20]
add edx,1
xor ecx,edx
mov eax,dword ptr ss:[ebp-44]
mov byte ptr ds:[eax+E8A13D],cl
jmp moeda2022fix.E81E4E
```

אבל זה איך שזה נראה הבלוק שעושה הצפנה לכל תו.

דבר ראשון יש את המחרוזת שבמקום E8A13C שאנחנו משתמשים בה אז לפי הקוד והזיכרון היא:

```
Hex                                                ASCII
.3C 59 68 65 79 6F 31 2A 3A 3A 3B 00 00 00 00 00 00 Yheyo1*:
.4C 00 00 00 00 24 00 00 00 00 00 00 00 00 00 00 00 ....$.....
```

ולכן אני אשתמש בה בהמשך.

אז למעשה אני לוקח כל תו ועושה לו קסור עם A כי זה הערך של המשתנה שנשאר לי מהסכום של הצ'ארים(בקוד שם זה עשה גם השמה כדי לשנות אותו לA0)

```
mov     eax, [ebp+counter2]
movsx   ecx, byte_41A13C[eax]
xor     ecx, [ebp+charsSum]
mov     edx, [ebp+counter2]
mov     byte_41A13C[edx], cl
```

כאן אני מחליף כל תו עם הקסור של A.

```
Hex                                                ASCII
.3D 68 65 79 6F 31 2A 3A 3A 3B 00 00 00 00 00 00 00 heyo1*:;
.4D 00 00 00 24 00 00 00 00 00 00 00 00 00 00 00 00 ...$.....
.5D 00 00 00 02 00 00 00 00 00 00 00 00 01 00 00 00 .........
```

וזה התמונה אחרי שבעקבות זה התו הראשון השתנה לh מה שכמובן שזה לא הכל, אז צריך לראות את החלק השני:

השלב הבא יהיה:

```
mov     eax, [ebp+counter2]
movsx   ecx, byte_41A13D[eax]
mov     edx, [ebp+charsSum]
add     edx, 1
xor     ecx, edx
mov     eax, [ebp+counter2]
mov     byte_41A13D[eax], cl
```

למעשה כאן אנחנו לוקחים כל תו, ואחרי זה מוסיפים 1(זמנית) לA שזה הסכום של הצ'ארים אחרי השינוי, ועושים עוד קסור עם התו שבמקום של קאונטר 2.

הנקודה המעניינת שזה 2 מחרוזות שונות שכל אחת מהן באורך 5 תווים, למעשה אני עושה קסור שונה לכל תו, ולכן כל תו אי זוגי מקבל קסור של 10 וכל זוגי מקבל 11.

```
def decrypt():
    ouptput = "Yhevo1*:;;"

    decrypted = ""

    for i in range(10):
        if i % 2 == 0:
            decrypted += chr(ord(ouptput[i])^10)
        else:
            decrypted += chr(ord(ouptput[i]) ^ 11)

    print(decrypted)
```

זה הקוד של הפענוח.

הערה: אני יודע שהקוד כאילו עושה את הבית במקום 41A13Dכי זה למעשה מביא לו מקום אי זוגי כי זה בעצם מוסיף 1 ידנית לכתובת הבסיס, אני כתבתי את הקוד לפי בדיקה עם תנאי של אם זה זוגי או לא ומה כל פעם, ככה שמבחינה פונקציונלית זה אותו קוד, רק שהתוכנית יותר מהירה כי היא לא באמת בודקת וקופצת 2 תווים כל פעם וככה משנה כל תו עם ערך אחר. אז זה אותו קוד בפועל..

גרסה סופית של הבדיקה:

```
def test():
    userInput = input("What would you li
    sum=0

    for c in userInput:
        sum+=ord(c)

    if sum&255 == 10:
        decrypt()
    else:
        print("Well... keep trying")
```

(ההבדל שראיתי שלא הדפסתי את ההודעה למקרה של כישלון)