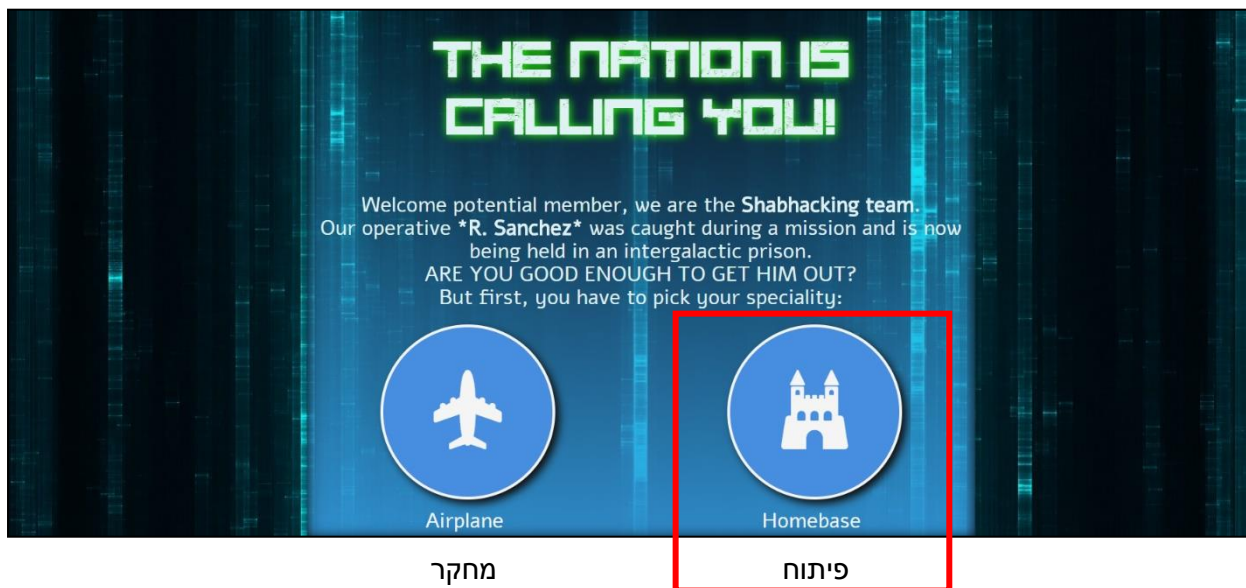


באתר הזה מופיעים האתגרים של השב"כ בשני חלקים כמו שרואים בתמונה:

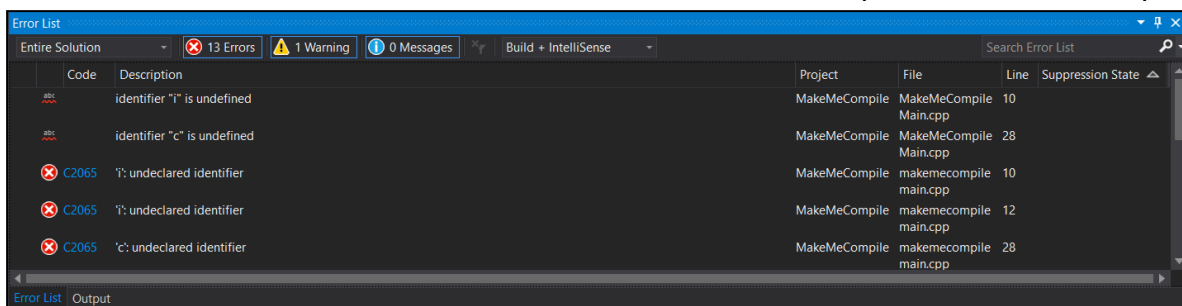


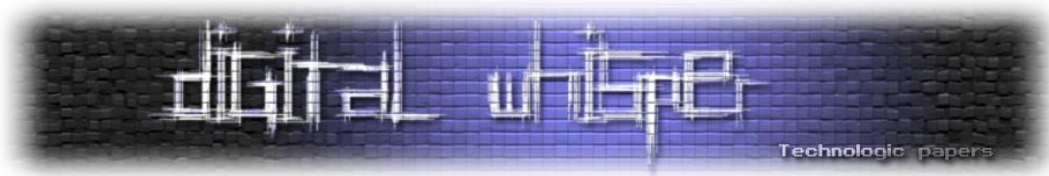
שלב ראשון: 99 bugs in the code



בתרגיל הזה קיבלנו קוד שלא מתקמפל. המטרה של התרגיל היא להבין את השגיאות בקומפילציה, לתקן אותן ולקבל את הסיסמה למשימה הבאה. בתור התחלה נפתח את הקובץ `MakeMeCompile.vcxproj` עם Visual Studio גרסת Community Edition (שכל אחד יכול כבר להוריד מהאתר של מיקרוסופט).

נלחץ **CTRL + Shift + B** כדי לקמפל ויחשכו עינינו:





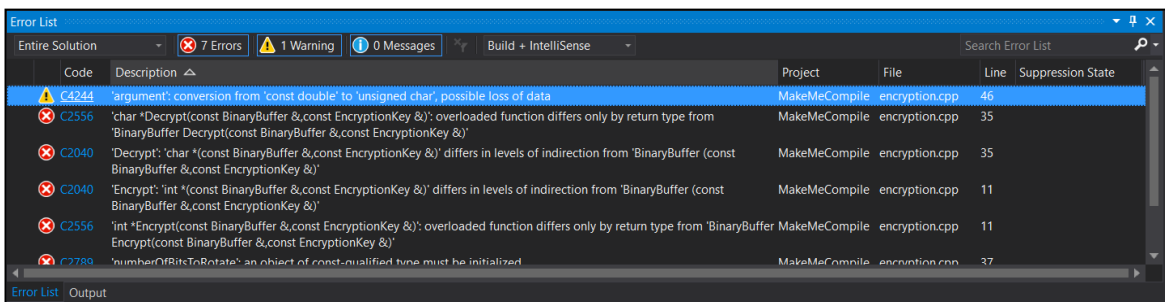
אנחנו רואים שיש בעיות בקובץ הראשי **MakeMeComplieMain.cpp** שברובן קשורות לסוגים של המשתנים, בואו נתחיל מהשגיאה הראשונה של סוג המשתנה **i** וסוג המשתנה **c**.

```

7 BinaryBuffer GetEncryptedBuffer()
8 {
9     std::string str = "Password";
10    for (i = 0; i < str.length(); i++)
11    {
12        str[i] += 1;
13    }
14
15    if (str[4] != 120)
16    {
17        return SomeFunction9936();
18    }
19    else
20    {
21        return SomeFunction145();
22    }
23 }
24
25 BinaryBuffer GetEncryptionKey()
26 {
27     short i = 15;
28     c = i % 6;
29     if (c > 6)
30     {
31         return SomeFunction1839();
32     }
33     std::vector<char> v;
34     for (; i < 100; ++i)
35     {
36         v.emplace_back(1);
37     }
38     return (v.size() > 80) ? SomeFunction1362() : SomeFunction4932();
39 }
40 }
41

```

עכשיו ננסה לקמפל שוב. **unsigned int-short** משתנים ללא סוג ל-**c** ו-**i** אז תיקנו את סוגי המשתנים



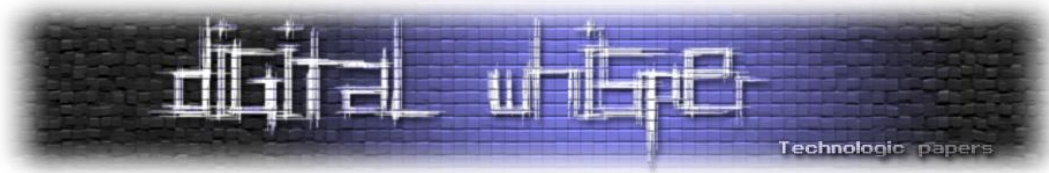
כבר ירד מספר השגיאות שלנו מ-13 ל-7, אז בואו ננסה להבין מהן השגיאות בקובץ **Encryption.cpp**.

```

11 int Encrypt(const BinaryBuffer& plainText, const EncryptionKey& key)
12 {
13     const auto xorKey = key[xorKeyLocation];
14     const auto numberOfBitsToRotate = key[numberOfBitsToRotateLocation];
15     const BinaryBuffer result;
16     do
17     {
18         std::transform(
19             plainText.begin(),
20             plainText.end(),
21             std::back_inserter(result),
22             [&](const auto byte)
23             {
24                 const auto xored = byte ^ xorKey;
25                 const auto shifted = _rotl8(xored, numberOfBitsToRotate);
26                 return shifted;
27             });
28     } while (0);
29     return result;
30 }
31
32 char* Decrypt(const BinaryBuffer& cipherText, const EncryptionKey& key)
33 {
34     const auto xorKey = key[xorKeyLocation];
35     const auto numberOfBitsToRotate = key[(std::vector<int>)numberOfBitsToRotateLocation];
36     const BinaryBuffer result;
37 }
38

```

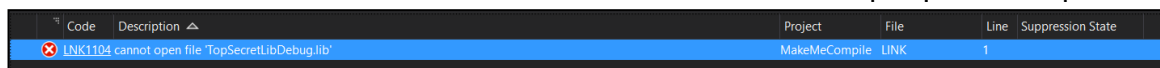
השינויים שאנחנו צריכים לעשות בקובץ Encryption גם הם קשורים לסוגי משתנים, אנחנו רואים שגם בפונקציית Encryption וגם בפונקציית Decryption מצהירים על משתנה בשם **result** מסוג **BinaryBuffer**, המשתנה הזה כמובן לא יכול להיות קבוע (**const**) כי הוא יעבור שינויים בהמשך הדרך אז אנחנו מוחקים את ה-**const**.



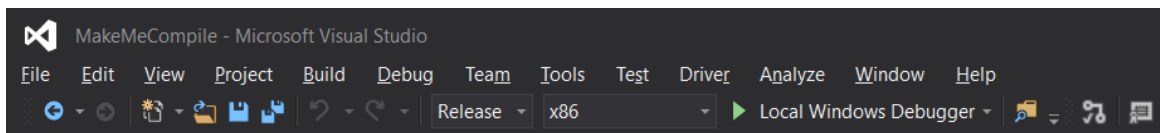
אז אם סוג המשתנה שחוזר הוא **BinaryBuffer** צריך להחליף את ההצהרה של הפונקציה שתחזיר את סוג המשתנה הזה: אנחנו מחליפים לפונקציה **Encrypt** את סוג ההחזרה מ-**int*** ל-**BinaryBuffer** וגם בפונקציה **Decrypt** מ-**char*** ל-**BinaryBuffer**.

נראה שהשינוי האחרון שנשאר לנו לעשות בקוד הוא למחוק את ההמרה (`std::vector<int>`) למשתנה `numberOfBitsToRotateLocation` בפונקציה **Decrypt** כי בפונקציה **Encrypt** היא לא קיימת.

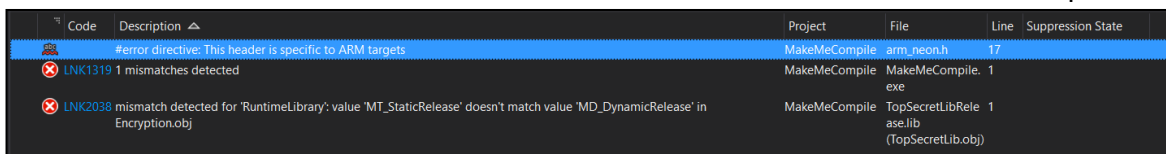
אנחנו מקמפלים ועדיין נתקלים בשגיאה:



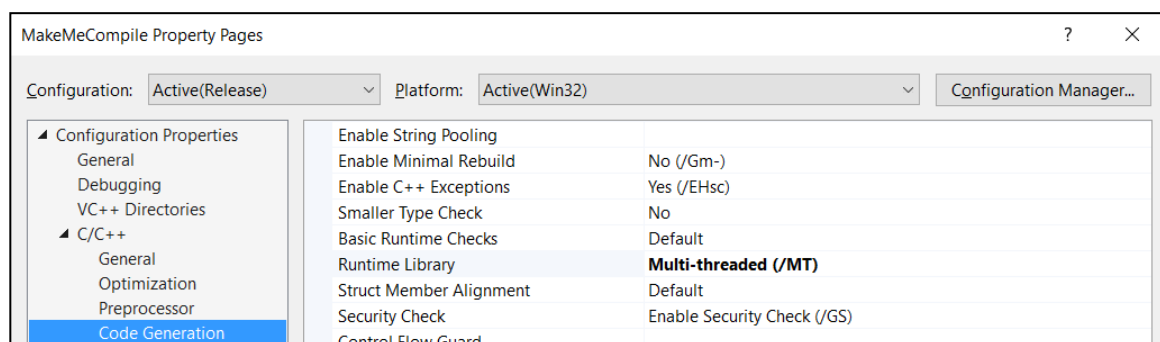
בשגיאה הזו נאמר לנו שיש לנו ספרייה סטטית ואנחנו לא מצליחים לפתוח אותה, כמובן הספרייה הסטטית שהוא מנסה לפתוח היא ספרייה בקונפיגורציה של **Debug** ולנו יש רק את הספרייה הסטטית בקונפיגורציה של **Release**.



אנחנו מקמפלים שוב את התוכנית ורואים את הבעיה שנראית כמו הבעיה האחרונה:



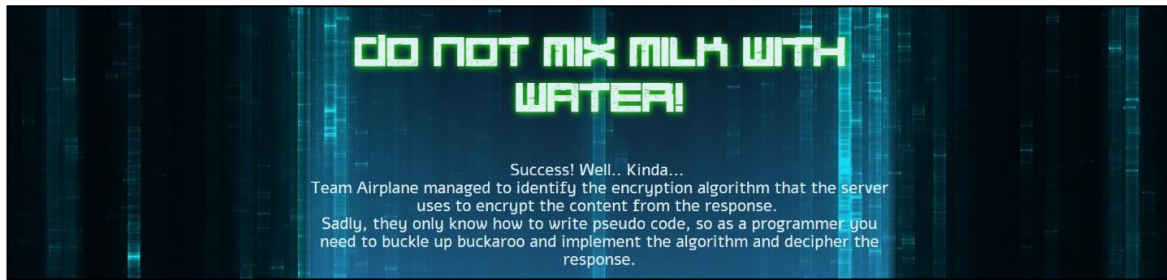
הבעיה הזו כבר לא קשורה לקוד עצמו אלא להגדרות הפרויקט, יש שוני בין הגדרות הקומפילציה של הפרויקט לספרייה הסטטית, לכן נכנס לאפשרויות של הפרויקט ונחליף את ה-**Runtime Library** מ-**Multi-threaded DLL (/MD)** ל-**Multi-threaded (/MT)**.



אנחנו מקמפלים בפעם האחרונה את התוכנית ולאחר מכן מריצים אותה עם `CTRL + Shift + F5` כדי שגם תעצור לאחר הריצה, והנה התשובה שחיפשנו: **RoadRage** היא הסיסמה לשלב הבא.



שלב שני: The Algorithm



בתרגיל הזה אנחנו צריכים לכתוב את הקוד ל-MegaDecryptor, מפתח ההצפנה בנוי ממערך של מבנים מהסוג הבא:

```
struct EncryptionStepDescriptor {
    UINT8 operationCode;
    UINT8 operationParameter;
    UINT32 lengthToOperateOn;
};
```

- **operationCode** - מספר בין 0 ל-2 אשר מציינ את הפעולה האריתמטית \ לוגית אשר תבצע על הטקסט המוצפן (0 - Xor, 1 - Add (חיבור), 2 - Subtract (חיסור)).
- **operationParameter** - הפרמטר עמו תבצע הפעולה האריתמטית \ לוגית (מספר בין 0-255).
- **lengthToOperateOn** - מספר הפעמים שתבצע הפעולה האריתמטית \ לוגית.

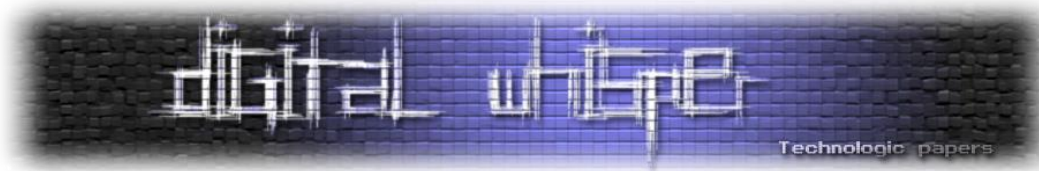
יחד עם הסבר על המנגנון ומבנה המפתח אנחנו מקבלים דוגמאות של טקסטים לפני ואחרי פיענוח.

לדוגמה:

הטקסט "ccccccc" בצירוף המפתח: { {Add, 1, 4}, {Subtract, 2, 3} }, יביא לתוצאה: "dddadaa"

הטקסט "aaaaaaaaaaaaeehhhhhhhhhgggghh" בצירוף המפתח: { {Add, 5, 10}, {Add, 1, 5}, {Subtract, 2, 9}, {Subtract, 1, 8} }, יביא לתוצאה "fffffffffffffffffffff".

אתם יכולים לשים לב שגודל הטקסט הוא: 30 ומספר הפעולות שיבצעו על הטקסט הוא: 32 שהוא יותר גדול מגודל הטקסט. מה שאומר שכשנגיע למצב ה"ל" "fffffffffffffffffffffee" נגמר לנו הטקסט, אז מה שהמנגנון אומר הוא שנפענח את התו האחרון בשנית ונחזור אחורה (במקרה שלנו עוד שני תווים אחורה יפוענחו).



וכעת נעבור להסבר על הקוד אשר עושה את כל תהליך הפיענוח:

- אני די בטוח שכולם יכתבו את הקוד ב-C, שזה די הגיוני כי C יותר מהירה ויותר מתאימה לפעולות מתמטיות מהסוג הזה (דבר שדווקא מדרבן אותי להראות לכם איך אותו הקוד היה נראה בפיתון + עצלנות לכתוב ב-C...).

```
from ctypes import Structure, c_uint8, c_uint32, sizeof

# EncryptionStepDescriptor C Structure.
class EncryptionStepDescriptor(Structure):
    pack = True
    _fields_ = [
        ("operationCode", c_uint8),
        ("operationParameter", c_uint8),
        ("lengthToOperateOn", c_uint32)
    ]

# Do Logical / Math calculation on cipher text.
def do_operation(cipher, code, parameter):
    if code == 0:
        return cipher ^ parameter
    elif code == 1:
        return (cipher + parameter) % 256
    elif code == 2:
        return (cipher - parameter) % 256

# brange is bidirectional range generator.
def brange(start=0, end=30, inc=1):
    while True:
        i = start
        while i < end:
            yield i
            i += inc

        i = end
        while i > start:
            i -= inc
            yield i

def decrypt(cipher_text, key):
    cipher_text = bytearray(cipher_text)

    # Create EncryptionStepDescriptors
    esd_list = []
    for i in xrange(0, len(key), sizeof(EncryptionStepDescriptor)):
        esd_list.append(EncryptionStepDescriptor.from_buffer_copy(key[i:]))

    iter_brange = brange(0, len(cipher_text))
    for esd in esd_list:
        for _ in xrange(esd.lengthToOperateOn):
            # Next index
            index = next(iter_brange)
            cipher_text[index] = do_operation(cipher_text[index], esd.operationCode,
            esd.operationParameter)

    return str(cipher_text)

if __name__ == '__main__':
    # Example 1
    print decrypt("ccccccc", ("\x01\x01\x04\x00\x00\x00"
    "\x02\x02\x03\x00\x00\x00"))

    # Example 2
    print decrypt("aaaaaaaaaeeeehehhhhhhggggh", ("\x01\x05\x0a\x00\x00\x00"
    "\x01\x01\x05\x00\x00\x00"
    "\x02\x02\x09\x00\x00\x00"
    "\x02\x01\x08\x00\x00\x00"))

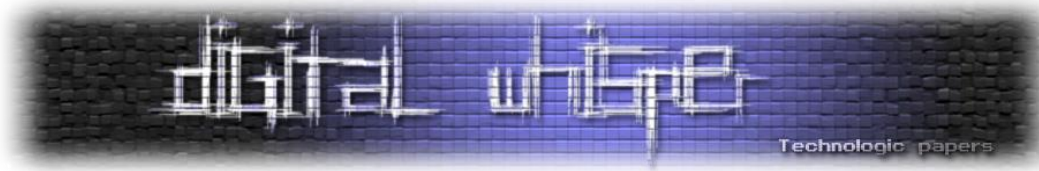
    # Solution
    print decrypt(open('EncryptedMessage.bin', 'rb').read(), open('Key.bin', 'rb').read())
```

- ctypes - ספרייה מובנית בפיתון אשר מחברת בין פיתון ל-C.

- EncryptionStepDescriptor - מבנה בשפת C (חלק מהמפתח).

פתרון אתגר השב"כ 2017 - פיתוח

www.DigitalWhisper.co.il



- **do_operation** - פונקציית עזר לביצוע פעולה אריתמטית \ לוגית על טקסט מוצפן.
- חדי העין ישימו לב לפעולת מודולו עם 256, המטרה של הפעולה הזו היא לדמות Integer Overflow ו- Underflow כמו בשפת C.
- הכוונה היא שכשבשפת C מבצעים את הפעולות הבאות:

```
UINT8 a = 122;
a += 172;
printf("%d\n", a);
```

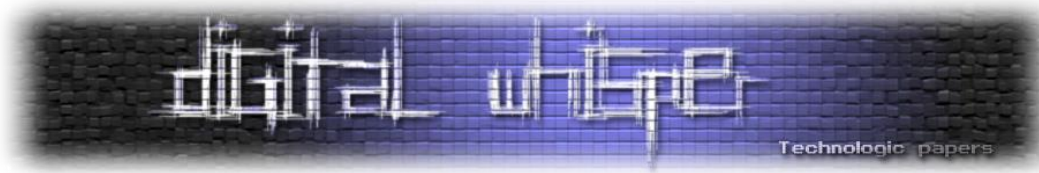
- במקום לקבל שגיאה, נקבל פלט עם המספר **38** (כלומר: $(122 + 172) \% 256$).
- **brange** - גנרטור אשר יאפשר לנו לרוץ הלוך חזור על הטקסט המוצפן (כמו בדוגמה השנייה).
- **decrypt** - פונקציה אשר מבצעת את פעולת הפיענוח על טקסט עם מפתח.
- בפונקציה זו אנו יכולים לראות שימוש בפונקציה **from_buffer_copy** ממבנה הנתונים בשפת C, זוהי פונקציה מאוד שימושית אשר לוקחת **Buffer** ומזינה ממנו מבנה נתונים בשפת C. כלומר הפעולה (`EncryptionStepDescriptor.from_buffer_copy("\x01\x01\x04\x00\x00\x00")`) תייצר לנו את האובייקט הבא:

```
struct EncryptionStepDescriptor {
    operationCode: 1,
    operationParameter: 1,
    lengthToOperateOn: 4
};
```

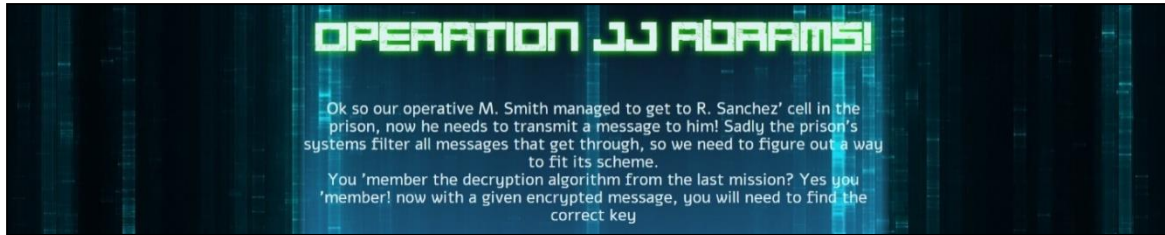
לאחר שבנינו את המפתח (רשימה של **EncryptionStepDescriptor**) אנחנו מיצרים את האיטרטור שלנו (**iter_brange**) וכל פעם שאנחנו רוצים לקחת את האיבר הבא באיטרטור אנחנו משתמשים בפונקציה **next**. וודאי שמתם לב שבהתחלה במקום להשתמש במחרוזת המרתי את המחרוזת ל-**bytearray**. **bytearray** מאפשר לנו לבצע פעולות אריתמטיות על מחרוזת (מייצג את המחרוזת בתור רשימה של מספרים) ומאפשר להמיר חזרה למחרוזת בקלות. בפעולה הראשית (`if __name__ == '__main__'`) אפשר לראות שתחילה לבדיקת שפיות, השתמשנו בדוגמאות ורק לאחר מכן בנתונים של התרגיל. נריץ את הקובץ ונקבל את הפלט הבא:

```
dddddaaa
ffffffffffffffffffffffffffffffffffff
Great job! You are the 1337est hacker of all times! The password for this stage is: 'Look at me I'm Mister programmer'
Now go on and enjoy your coffee break
```

זה אומר שאם נכניס את התשובה "Look at me I'm Mister programmer" נוכל להמשיך לשלב הבא.



שלב אחרון: BruteForce



באתגר הזה קיבלנו טקסט מוצפן ללא מפתח שעליו אנחנו צריכים לבצע ברוטפורס. אלה הנתונים שיש לנו:

- הטקסט המוצפן הוא בגודל 54 תווים.
- אין לנו באמת שלושה סוגים של פעולות, יש לנו רק שניים (XOR ו-חיבור/חסור) בגלל ה-Integer Overflow/Underflow.
- מספר הפעולות שנעשה אומנם כתוב במשתנה מסוג Unsigned Int אבל בתכלס לא יכול להיות גדול מגודל הטקסט * 2 - הכוונה היא שריצה הלורך חזור (היא ריצה של גודל הטקסט * 2) תייצר לנו בפועל ריצת הלורך עם ערך * 2 (במקרה של חיבור/חסור ובמקרה של XOR תחזיר את הטקסט לקדמותו).

השאלה הראשונה ששואלים לאחר פתרון האתגר הזה היא: **כמה זמן ערך הברוטפורס?** התשובה היא: **5 דקות!** (איך? זה לא הגיוני הרי יש המון אפשרויות אפילו עם הצמצומים הללו)... הייתה לי כמובן את האפשרות לעבור על כל האפשרויות (169075682574336) אך זה פתרון נאיבי שלוקח הרבה זמן. הייתה לי האפשרות לנסות למצוא אלגוריתם יותר מורכב (לא למדתי מדעי המחשב אז הידע שלי באלגוריתמיקה הוא די מוגבל), אז איך בכל מקרה הפתרון לקח 5 דקות? עכשיו אסביר...

אם אנחנו עוברים על כל הטקסט ומפענחים אותו עם Descriptor אחד בגודל של הטקסט, ישנו סיכוי גבוה שנקבל טקסט קריא, אז בואו נכתוב קוד שעושה את זה:

```
import itertools
from pprint import pprint
from functools import partial
from multiprocessing.pool import ThreadPool

def decrypt(cypher_text, code_param):
    code, param = Code param
    decrypted_block = bytearray()
    for c in bytearray(cypher_text):
        if code == 0:
            dc = c ^ param
        elif code == 1:
            dc = (c + param) % 256
        elif code == 2:
            dc = (c - param) % 256

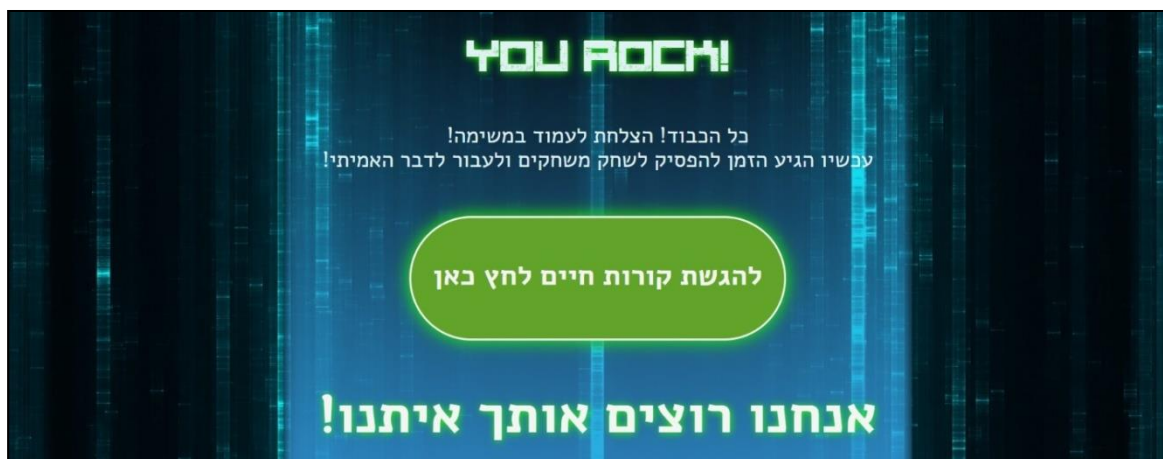
        decrypted_block.append(dc)

    return (code, param), str(decrypted_block)

def gen_operations():
    codes = range(2)
    params = range(0, 256)
```


Is it done, Yuri? No, Comrade Premier. It has only begun.

נסיר את ה-'!', (תו לא חוקי) ועדיין הטקסט גדול בתו אחד, אבל כמו שאנחנו רואים הטקסט מסתיים ב-begun בלי נקודה, מה שאומר שהגודל של הטקסט ללא הנקודה הוא 54, בול הגודל של הטקסט המוצפן. נכניס את הטקסט "Is it done, Yuri? No, Comrade Premier. It has only begun" ועברנו את המשימה האחרונה!



על המחברים

- **D4D**: עוסק בתחום ה-Reverse Engineering - בחברת IronSource במחלקת ה-Security ואוהב לחקור משחקי מחשב והגנות, לכל שאלה שיש או ייעוץ ניתן לפנות אלי דרך:
 - שרת ה-IRC של Nix בערוץ: #reversing
 - או באתר: www.cheats4gamer.com
 - או בכתובת האימייל: llcashall@gmail.com.
- **תומר זית (RealGame)**: חוקר אבטחת מידע בחברת F5 Networks וכותב Open Source.
 - אתר אינטרנט: <http://www.RealGame.co.il>
 - אימייל: realgam3@gmail.com
 - GitHub: <https://github.com/realgam3>