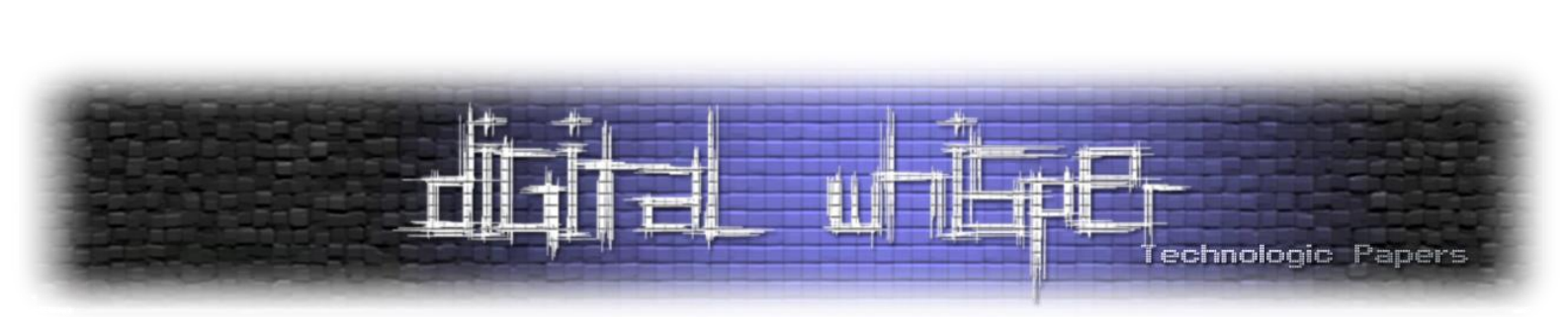


Reversing Compiled Python

מאת לירן פאר (reaction)

הקדמה

במאמר זה אציג ואסביר על הנושא Compiled Python Reverse Engineering תוך כדי עבודה מעשית. לאורך המאמר אשתמש באתגר #6 של Flare-On 2016 כתרגיל, כדי שנוכל ליישם תאוריה. Flare-On הוא סט אתגרים בתחום Reverse Engineering מבית חברת FireEye אשר מתפרסם במתכונת שנתית. כל אתגר מכיל בדרך כלשהיא סיסמה בצורת אימייל אשר נגמרת ב-flare-on.com@, והמטרה היא להשיג אותה.

תקופת התחרות של שנת 2016 הסתיימה, וכעת אפשר להוריד את כל עשרת האתגרים מהקישור הבא:

http://flare-on.com/files/Flare-On3_Challenges.zip

(סיסמה לארכיון: flare)

אסתמך במאמר זה כי לקורא ידע בסיסי לפחות בהנדסה הפוכה ופייתון.

סקירה בסיסית

הבה נתחיל! האתגר מכיל קובץ הרצה (PE Exe) אחד בשם kahki.exe. כשאנחנו מריצים אותו, אנו רואים כי הוא מבקש מאיתנו לנחש מספר מסוים מטווח של 1 עד 100, וסופר את מספר הנסיונות שלנו:

```
C:\Windows\system32\cmd.exe

C:\Users\Admin\Downloads\flare-on\6>khaki
(Guesses: 1) Pick a number between 1 and 100:50
 Too high, try again
(Guesses: 2) Pick a number between 1 and 100:25
 Too high, try again
(Guesses: 3) Pick a number between 1 and 100:12
 Too high, try again
(Guesses: 4) Pick a number between 1 and 100:6
 Too low, try again
(Guesses: 5) Pick a number between 1 and 100:9
 Too low, try again
(Guesses: 6) Pick a number between 1 and 100:10
 Too low, try again
(Guesses: 7) Pick a number between 1 and 100:11
Wahoo, you guessed it with 7 guesses
Status: 7 guesses
```

האם אנחנו יכולים להסיק מכך הרבה? לא ממש. אנו נדרשים להסתכל על הלוגיקה הפנימית של התוכנה.

דבר אחד מאוד בולט לנו ברגע שנבצע ניתוח סטטי בסיסי על התוכנה: זהו קובץ שנוצר ע"י py2exe. האינדיקציות רבות:

- גודל התוכנה הוא 3.63 MiB, זהו גודל לא אופייני לתוכנה שנראית יחסית פשוטה.
- נראה שב-resource section שנמצא ב-PE header יש directories בשמות "PYTHON27.DLL" ו-"PYTHONSCRIPT".
- חיפוש מחרוזות בקובץ ההרצה יניב שלל תוצאות, כמו "PY2EXE_VERBOSE" ו-"Could not load python dll".

אין כאן הרבה מקום לספק. אם יצא לכם לפתח לסביבת Windows בעזרת פייתון, יכול להיות שאתם מכירים את py2exe. זוהי תוכנה שלוקחת סקריפט/ים הכתובים בפייתון, ויוצרת תכנת native ל-Windows: ללא תלות בקיום פייתון על המחשב שמריץ את התוכנה, וללא תלות בספריות אשר קיימות אצל מריץ התוכנה. דבר זה כמובן עוזר עם portability ל-Windows בין מחשבים שונים. תהליך ההמרה הוא לא קסם, בגדול: מהדרים את קוד הפייתון לבייטקוד, מצרפים מפרש פייתון (כ-dll) ל-exe ונותנים לו להריץ את הבייטקוד. במקרה זה, גרסאת הפייתון המשומשת היא 2.7.

חשוב: על מנת להמשיך במאמר, נצטרך להשתמש בגרסה 2.7 של פייתון, כשם הגרסה של האתגר.

אז מה הדבר הראשון שאנחנו צריכים לעשות? להשיג את הבייטקוד כמובן. כשמהדרים סקריפט פייתון (.py), מקבלים קובץ מהודר (.pyc או .pyo) אשר כמובן לא קריא כטקסט. כדי לחלץ את קבצי ה-pyc שבשימוש נשתמש בסקריפט שנקרא unpy2exe.py, שאותו אפשר להשיג מכאן:

https://github.com/matiasb/unpy2exe

(אציין גם כי ניתן להשתמש בתוכנות אחרות שמבצעות את אותה הפעולה, כמו Py2ExeDumper[1] ו-Py2Exe Binary Editor[2]).

```
C:\Windows\system32\cmd.exe
C:\Users\Admin\Downloads\flare-on\6>py -2.7 unpy2exe.py khaki.exe
Magic value: 78563412
Code bytes length: 4386
Archive name: -
Extracting boot_common.py.pyc
Extracting poc.py.pyc
```

נקבל את שני הקבצים "boot_common.py.pyc" ו-"poc.py.pyc". אנחנו יכולים להתעלם מהראשון משום שהוא קוד ששייך ל-py2exe. אוסיף כי חילוץ הקבצים הללו אינו עסק מסובך: הם נמצאים בקובץ ההרצה כמשאב (resource). תוכלו להסתכל על קוד המקור הקל-להבנה של unpy2exe.py כדי להבין את התהליך.

ישנם כלים כמו uncomyple6[4], decompyle[3], ו-Easy Python Decompiler[5] אשר מסוגלים לבצע decompilation לקבצים מהודרים (רובם, לפחות) ולפשט את חיינו, וזה כמובן מה שנבחר לעשות בכל מקרה שכזה; אך אם ננסה להשתמש בהם, נגלה כי הם נכשלים להמיר את הקובץ המהודר לקוד פייתון!

```
C:\Users\Admin\Downloads\flare-on\6>uncompyle6 poc.py.pyc
# uncompyle6 version 2.9.3
# Python bytecode 2.7 (62211)
…
Traceback (most recent call last):
  File "c:\python27\lib\runpy.py", line 174, in _run_module_as_main
    "__main__", fname, loader, pkg_name) …
    jump_targets = self.find_jump_targets()
  File "c:\python27\lib\site-packages\uncompyle6\scanners\scanner2.py", line 844
, in find_jump_targets
    self.detect_structure(offset, op)
  File "c:\python27\lib\site-packages\uncompyle6\scanners\scanner2.py", line 621
, in detect_structure
    jmp = self.next_except_jump(i)
  File "c:\python27\lib\site-packages\uncompyle6\scanners\scanner2.py", line 462
, in next_except_jump
    self.jump_forward | frozenset([self.opc.RETURN_VALUE])
AssertionError
```

[1] https://sourceforge.net/projects/py2exedumper/
[2] https://sourceforge.net/projects/p2ebe/
[3] https://sourceforge.net/projects/decompyle/
[4] https://pypi.python.org/pypi/uncompyle6/
[5] זוהי תוכנה אשר מפשטת עניינים ומספקת חזית GUI לכלים uncompyle2 ו-decompyle++.
https://sourceforge.net/projects/easypythondecompiler/

מוצג חלק מדוח החריגה שנזרקת בזמן ה-decompilation. אם נפתח את הקובץ בו נזרקה החריגה נבין כי הכלי מצפה (בעזרת assert) למבנה אחיד מסוים שלא מתקיים ב-pyc, ולכן נכשל:

```
count_END_FINALLY = 0
count_SETUP_ = 0
for i in self.op_range(start, len(self.code)):
    op = self.code[i]
    if op == self.opc.END_FINALLY:
        if count_END_FINALLY == count_SETUP_:
            if self.version == 2.7:
                assert self.code[self.prev[i]] in \
                    self.jump_forward | frozenset([self.opc.RETURN_VALUE])
            self.not_continue.add(self.prev[i])
            return self.prev[i]
        count_END_FINALLY += 1
    elif op in self.setup_ops:
        count_SETUP_ += 1
```

אין לנו הרבה ברירות: נאלץ לנבור בבייטקוד שנמצא ב-poc.py.pyc.

קצת על python internals

אוקיי, אז לפני שנתחיל לחקור את הבייטקוד, יועיל אם נבין דבר או שניים לגבי איך העסק עובד.

לפני הכל, יש לציין כי הדברים שקובעים איך יראה ומה יכיל קובץ פייתון מהודר הם ההטמעה של פייתון (implementation) והגרסה שבשימוש; דבר זה אומר שאם נשתמש בהטמעה מסוימת, כמו PyPy, IronPython, ו-Jython, נקבל בינארי שלא מתאים לאחרים. אותו הדבר תקף גם לגרסת הפייתון שבה אנחנו משתמשים; אם נרצה להריץ קוד שהודר לגרסה אחרת, נתקל באי-תאימות—אלא אם כן השינוי הוא מינורי מאוד. דבר זה נכון גם לשינוי משמעותי כמו מעבר מפייתון 2 לפייתון 3 וגם למעבר תת-גרסה כמו 2.6 ל-2.7. מסיבה זו, לרוב לא מפיצים קבצי pyc לבדם, אלא אם אפשר להבטיח תאימות עם היעד, במקרה כמו embedded scripts. לכן אם נרצה להריץ את הקובץ poc.py.pyc, נצטרך להשתמש בגרסה 2.7 של פייתון.

CPython היא הטמעת ברירת המחדל, וגם הכי נפוצה של פייתון. זוהי ההטמעה שתהיה בשימוש אם נתקין פייתון מהאתר הרשמי (https://www.python.org/). בנוסף, py2exe תומך אך ורק בה נכון לזמן הכתיבה. נדון רק בהטמעה זו ובגרסה 2.7 במאמר. רק כדי להסיר ספקות: הטמעה בהקשר זה היא תוכנה שתפקידה הוא לקחת טקסט פייתון ובאמת להריץ אותו על המעבד (אשר כמובן לא מבין פייתון).

קובץ pyc מורכב משני חלקים: כותרת, ו-code object. הכותרת מורכבת משני שדות: ארבעה בתים של מספר המייצג גרסת פייתון (python magic number), וארבעה בתים אשר מייצגים חותמת זמן שינוי

אחרון של קובץ המקור. בפייתון 3.3 התווסף עוד שדה בגודל ארבעה בתים אשר מכיל את גודל קובץ המקור.

עובדה מעניינת: רק המספר שבשני הבתים הראשונים בשדה הגרסה באמת מייצג את הגרסה, בעוד ששני הבתים הנותרים הם CRLF ("\r\n"). מטרת ירידת השורה היא לנסות לגרום לשגיאת ניתוח במקרה שפותחים את הקובץ הבינארי במצב טקסט!

code object הוא יצוג פנימי של פיסת קוד הרצה, כמו פונקציה, מודול, מחלקה, או גנרטור. אובייקט כזה מכיל לא רק בייטקוד, אלא גם מידע חיוני לקוד, כגון קבועים, משתנים, דגלים, ושמות גלובליים בשימוש. אם נרצה לקבל את אובייקט הקוד של מחרוזת קוד מסוימת, נעזר בפונקציה הסטנדרטית compile(), אם נרצה ליצור אובייקט קוד ידנית, נשתמש ב-types.CodeType(). נוכל גם לגשת לאובייקט הקוד של פונקציה מסוימת בעזרת התכונה func_code, או אם אנו משתמשים בפייתון 3+ : __code__.

```
>>> code = """
from math import pi
if pi < 3.14:
    print('Oh no!')
"""
>>> compile(code, "<string>", "exec")
<code object <module> at 02288D58, file "<string>", line 2>
>>>
>>> def simple_func(int1, int2):
    result = int1 + int2 + 5
    print("The result is: {0}".format(result))

>>> simple_func.func_code
<code object simple_func at 022E8B60, file "<pyshell#14>", line 2>
>>> simple_func.func_code.co_name
'simple_func'
>>> dir(simple_func.func_code)
['__class__', '__cmp__', '__delattr__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', … , 'co_argcount', 'co_cellvars', 'co_code', 'co_consts',
'co_filename', 'co_firstlineno', 'co_flags', 'co_freevars', 'co_lnotab', 'co_name',
'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']
>>> simple_func.func_code.co_consts
(None, 5, 'The result is: {0}')
>>> simple_func.func_code.co_varnames
('int1', 'int2', 'result')
>>> simple_func.func_code.co_code
'|\x00\x00|\x01\x00\x17d\x01\x00\x17}\x02\x00d\x02\x00j\x00\x00|\x02\x00\x83\x01\x0
0GHd\x00\x00S'
>>> simple_func.func_code.co_names
('format',)
```

המכונה הווירטואלית (VM) של CPython היא stack-based, מה שאומר שפעולות אריתמטיות, העברת ארגומנטים לפונקציות, טעינת ערכים, ועוד, יתבצעו בעזרת המחסנית. אין אוגרים כמו שיש ב-x86. המכונות הווירטואליות של .NET ו-Java גם הן מבוססות-מחסנית.

אם נרצה להכפיל שני מספרים ב-VM מבוסס-מחסנית, סדר הפעולות יהיה כזה:

1. דחיפה של מספר #1 למחסנית.
2. דחיפה של מספר #2 למחסנית.
3. הוראת הכפלה, שבעצם תבצע את תת ההוראות הללו:
   - שליפה של מספר #2 מהמחסנית.
   - שליפה של מספר #1 מהמחסנית.
   - חיבור של שני המספרים.
   - דחיפה של הסכום למחסנית.

הערך שבראש המחסנית מכונה TOS, והאלמנטים שמתחת לראש מכונים TOS1, TOS2, TOS3 וכן הלאה. ניתן גם לייצג מיקום מסוים בצורת אינדקס כך: TOS[-i].

המודול הסטנדרטי dis מאפשר לנו לבצע disassembling לאובייקט קוד, או לאובייקט שאפשר להשיג ממנו אובייקט קוד, כמו פונקציה או מודול. בנוסף, dis מכיל שלל מידע על הוראות הבייטקוד, כמו: אילו הוראות מקבלות ארגומנטים, מילונים המאפשרים לנו להמיר הוראת בייטקוד מסוימת לערך המספרי המייצג אותה ולהפך, ועוד. הדוקומנטציה של dis מכילה את ה-reference הרשמי של קוד הביניים: https://docs.python.org/2/library/dis.html, ושם תוכלו לקרוא תיאור של כל הוראה. אציין כי המידע שבדוקומנטציה קצת שונה בכל גרסת פייתון על מנת לשקף את השינויים בהוראות וכו'. הנה דוגמה ל-dis המשתמשת בפונקציה שהגדרנו קודם:

```
>>> import dis
>>> dis.dis(simple_func.func_code)
  3           0 LOAD_FAST                0 (int1)
              3 LOAD_FAST                1 (int2)
              6 BINARY_ADD
              7 LOAD_CONST               1 (5)
             10 BINARY_ADD
             11 STORE_FAST               2 (result)

  4          14 LOAD_CONST               2 ('The result is: {0}')
             17 LOAD_ATTR                0 (format)
             20 LOAD_FAST                2 (result)
             23 CALL_FUNCTION            1
             26 PRINT_ITEM
             27 PRINT_NEWLINE
             28 LOAD_CONST               0 (None)
             31 RETURN_VALUE
```

כמו שאתם רואים, קוד הביניים הזה הינו פשוט לקריאה והבנה גם בלי להבין יותר מדי על הנושא. משתמשים ב-LOAD_FAST על מנת לדחוף למחסנית את הארגומנטים int1 ו-int2, בסדר הזה, ואז ב-BINARY_ADD כדי לשלוף מהמחסנית את TOS ו-TOS1, לבצע את החיבור, ולדחוף את התוצאה בחזרה ל-TOS. מבצעים עוד חיבור, הפעם טוענים רק את המספר 5 למחסנית ומבצעים חיבור משום שהתוצאה הקודמת כבר נמצאת במחסנית, ולאחר מכאן שומרים את התוצאה במשתנה result. בהמשך הפונקציה מתבצע הפלט של התוצאה.

שימו לב לכך שהארגומנטים להוראות כמו LOAD_FAST ו-STORE_FAST הם מספרים: מספרים אלו מתפקדים כאינדקסים למערך שכן מכיל את המידע של ההוראה. STORE_FAST 2 בעצם מתכוון לאלמנט השני במערך co_varnames, והוא, כפי שראינו, המשתנה result. נדע לאיזה מערך האינדקס בארגומנט מתכוון אליו לפי ההוראה עצמה: LOAD_CONST פועלת על co_consts, IMPORT_NAME פועלת על co_names, וכו'.

אציין כמה דברים לגבי הפורמט של ה-disassembly:

- המספרים שליד ההוראות מציינים את ה-offset של ההוראה בבייטקוד. לדוגמה, ההוראה הראשונה לוקחת שלושה בתים: אחד להוראה עצמה, ושניים לארגומנט.
- הארגומנט של הוראות בעלות פרמטר מצוין אחרי ההוראה עצמה, אחרי המרווח, ובסוגריים שליד הארגומנט מצוין למה בדיוק הארגומנט מתכוון. לדוגמה: LOAD_CONST 2 מציין את האלמנט באינדקס 2 ב-simple_func.func_code.co_consts, שערכו 'The result is: {0}', כפי שראינו קודם.
- המרחב בין שני בלוקי הקוד מציינים שורת קוד חדשה בקוד המקורי, והמספרים 3 ו-4 בשמאל הרחוק מייצגים את שורת הקוד הנוכחית בקוד המקורי.
- הוראה מתוייגת (כזו שלמשל קופצים אליה) תוקדם ב-"<<", נראה זאת אחר כך.

כפי שאנו רואים במספרי ה-offset, כל ארגומנט להוראה הוא בגודל שני בתים. אם הארגומנט גדול מדי לשני בתים, משתמשים בהוראה EXTENDED_ARG שתקדים את ההוראה הרלוונטית ותספק עוד שני בתים (כ-most significant), אך לא נתקל במקרה כזה כאן. הארגומנטים האלו שמורים בצורת little endian.

מעניין גם לדעת שבפייתון 3.6+ הוחלף הבייטקוד ב-16-bit wordcode, מה שאומר שגודל ההוראות עצמן (ללא הארגומנטים) הוא שני בתים במקום אחד.

אוסיף כי בהטעמה השנייה בפופולריותה, PyPy, המבוססת (כרגע) על פייתון 2, הרוב המוחלט של הדברים שכרגע הזכרתי תקפים גם כן: PyPy משתמשת באובייקטי קוד, ובאותו בייטקוד עם שינויים מינוריים, וניתן להשתמש במודול dis באותו אופן.

פתרון האתגר

הבה נחזור לאתגרנו. נתחיל בלקבל את ה-disassembly של הקוד המצוי ב-poc.py.pyc. נשתמש בקוד הבא למטרה זו:

```
import dis, marshal
with open("poc.py.pyc", "rb") as f:
    pyc_header = f.read(8)      # first 8 bytes comprise the pyc header
    code_obj = marshal.load(f)  # rest is marshalled code object

dis.dis(code_obj)
```

שימו לב: תצטרכו להסיט את הפלט משורת הפקודה לקובץ מסוים (בעזרת ">").

המודול הסטנדרטי marshal מאפשר לקרוא ולכתוב ערכי פייתון מסוימים—טיפוסים כגון list, tuple, int, וכמובן object code—בפורמט בינארי שיבטיח תאימות בין סביבות ריצה; בערך כמו המודולים pickle ו-shelve, רק שאין משתמשים בו למטרות כלליות בקוד: הוא נמצא בעיקר כדי לתמוך בקריאה וכתיבה של קוד פייתון מהודר. הפורמט הבינארי הזה יכול להשתנות בין גרסאות פייתון, למרות שזה לא קורה הרבה.

```
  2           0 LOAD_CONST               0 (-1)
              3 LOAD_CONST               1 (None)
              6 IMPORT_NAME              0 (sys)
              9 STORE_NAME               0 (sys)
             12 LOAD_CONST               0 (-1)
             15 LOAD_CONST               0 (-1)
             18 POP_TOP
             19 LOAD_CONST               1 (None)
             22 ROT_TWO
             23 ROT_TWO
             24 IMPORT_NAME              1 (random)
             27 NOP
             28 STORE_NAME               1 (random)

  4          31 LOAD_CONST               2 ('Flare-On ultra python obfuscater
2000')
             34 STORE_NAME               2 (__version__)
             37 ROT_TWO
             38 ROT_TWO
```

הדבר הראשון שקופץ לעין הוא ששמים את הערך 'Flare-On ultra python obfuscater 2000' במשתנה __version__.

זהו רמז מאוד עבה ל-obfuscation בקוד, ואכן, אם נסקור את הקוד ברפרוף נוכל להבחין במקטעי קוד זבל שמופיעים לאורך כל הקוד:

- ROT_TWO, ROT_TWO: מחליף את TOS ב-TOS1, ואז עוד פעם; אין כאן השפעה על הלוגיקה.
- ROT_THREE, ROT_THREE, ROT_THREE: אותו הסיפור כמו ROT_TWO, רק עם TOS, TOS1, ו-TOS2.
- LOAD_CONST, POP_TOP: דוחף קבוע למחסנית, ואז מיד מסיר אותו.
- NOP: לא מבצע שום פעולה.

הוראות הזבל הללו לא שם רק כדי לסבך את הקוד, אלא מבצעים עוד תפקיד ערמומי: הם אחראיים לכך שהקוד לא יוכל לעבור decompilation, וזאת ע"י ניצול הלוגיקה ה"נאיבית" וההשערות של הכלים המבצעים את הפעולה. המכונה הווירטואלית עצמה לא כזו נאיבית כמובן, ומאפשרת הרצה של הקוד.

מכאן אפשר להמשיך בשתי דרכים:

דרך אחת תהיה למחוק את קוד הזבל מה-disassembly שייצרנו, מתי שנתקל בו, או בצורה הרבה פחות מייסרת: חיפוש והחלפה בעזרת כמה ביטויים רגולריים מהירים בכל עורך טקסט צנוע, לדוגמה:

- .+?NOP\s*\r\n
- .+?LOAD_CONST.+?\r\n.+?POP_TOP.+?\r\n
- .+?ROT_TWO.+?\r\n.+?ROT_TWO.+?\r\n
- .+?ROT_THREE.+?\r\n.+?ROT_THREE.+?\r\n.+?ROT_THREE.+?\r\n

בצורה זו נוכל לחצות את כמות השורות ב-disassembly. נעזר בדוקומנטציה של ההוראות (https://docs.python.org/2/library/dis.html) כדי להבין את משמעותן (רובן דיי ישירות), בשילוב עם ידע בסיסי בפייתון, ותוך זמן קצר נצליח להבין מה הקוד עושה; הרי ה-disassembly הרבה יותר קריא מדבר כמו x86 assembly.

הבעיה היחידה שאולי תתקלו בה היא בשורות הללו:

```
495 LOAD_CONST              16 (<code object <genexpr> at 02219B60, file "poc.py",
line 80>)
498 MAKE_FUNCTION            0
```

אנחנו רואים כאן שיוצרים פונקציה מתוך קבוע באינדקס 16 שהוא אובייקט קוד, ודוחפים אותה למחסנית.

איך נחקור אובייקט קוד שנמצא בתוך אובייקט קוד? בדרך דומה מאוד לאיך שעשינו זאת לקובץ, רק שהפעם נעביר ל-dis.dis את הקבוע:

```
>>> import marshal, dis
>>> pyc = open("poc.py.pyc", "rb")
>>> pyc_header = pyc.read(8)
>>> co = marshal.load(pyc)
>>> dis.dis(co.co_consts[16])
 80           0 LOAD_FAST                0 (.0)
        >>    3 FOR_ITER                27 (to 33)
              6 STORE_FAST               1 (x)
              9 LOAD_GLOBAL              0 (chr)
             12 LOAD_GLOBAL              1 (ord)
             15 LOAD_FAST                1 (x)
             18 CALL_FUNCTION            1
             21 LOAD_CONST               0 (66)
             24 BINARY_XOR
             25 CALL_FUNCTION            1
             28 YIELD_VALUE
             29 POP_TOP
             30 JUMP_ABSOLUTE            3
        >>   33 LOAD_CONST               1 (None)
             36 RETURN_VALUE
```

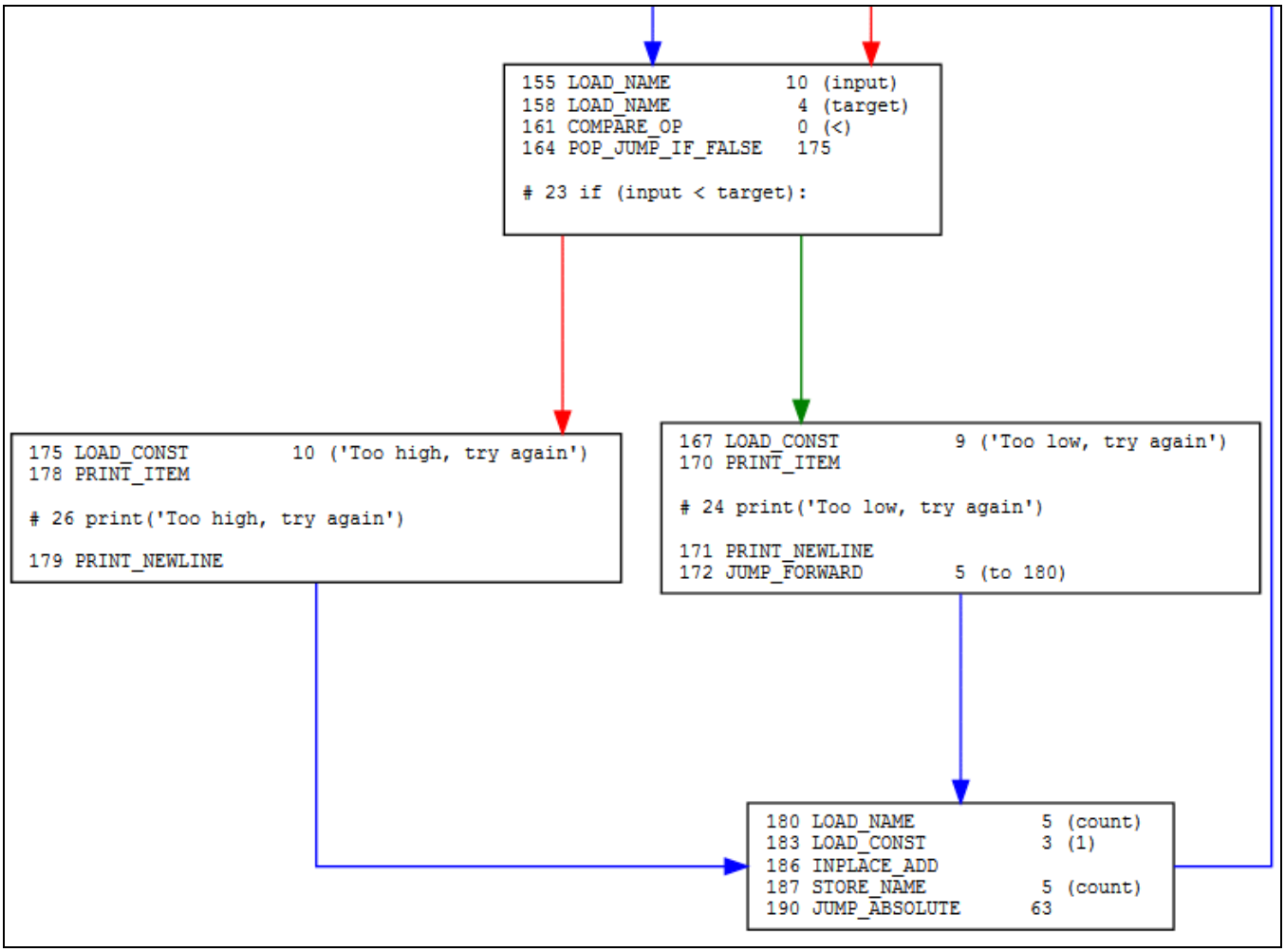
דרך שנייה להמשיך תהיה להסיר את קוד הזבל מהקוד עצמו על מנת שנוכל לבצע decompilation. פה נכנס לתמונה המודול bytecode_graph מאת צוות FLARE, אותם חברה שיצרו את האתגרים.

אפשר להוריד את המודול דרך pip (pip install bytecode-graph), או מכאן:

https://github.com/fireeye/flare-bytecode_graph.

מודול זה מאפשר לנו לערוך הוראות באובייקט קוד בקלות, ומטפל בדברים נחוצים כמו שינוי יעדי קפיצה אחרי הסרת הוראות וכו'.

בנוסף לכך, מודול זה מאפשר להציג גרף בקרת זרימה לבייטקוד בעזרת התוכנה GraphViz, כמתואר בתמונה שלמטה, אך לא ארחיב על כך.



נשתמש ב-bytecode_graph, marshal, ו-dis.opmap—מילון עם שמות הוראות כמפתחות וערכם המספרי כערכים—כדי ליצור סקריפט שמסיר את הוראות הזבל:

```
#!/usr/bin/env python2

import marshal
import bytecode_graph
from dis import opmap

def nop_junk(bcg):
    '''
    Turns junk code to NOPs.
    '''
    # resolve mnemonics to values
    nop_val = opmap["NOP"]
    rot_two_val = opmap["ROT_TWO"]
    load_const_val = opmap["LOAD_CONST"]
```

```
    pop_top_val = opmap["POP_TOP"]
    three_rot_three = tuple([opmap["ROT_THREE"] for _ in range(3)])

    # iterate instructions
    for node in bcg.nodes():
        if node.next is None:
            break

        if node.opcode == rot_two_val and node.next.opcode == rot_two_val:
            node.opcode = node.next.opcode = nop_val
        elif node.opcode == load_const_val and node.next.opcode == pop_top_val:
            node.opcode = node.next.opcode = nop_val
        elif node.next.next is not None and \
                (node.opcode, node.next.opcode, node.next.next.opcode) ==
three_rot_three:
            node.opcode = node.next.opcode = node.next.next.opcode = nop_val

def remove_nops(bcg):
    '''
    Removes NOPs from bytecode
    '''
    for node in bcg.nodes():
        if node.opcode == opmap["NOP"]:
            bcg.delete_node(node)

def main():
    with open("poc.py.pyc", "rb") as pyc_file:
        pyc_header = pyc_file.read(8)
        co = marshal.load(pyc_file)

        bcg = bytecode_graph.BytecodeGraph(co)
        nop_junk(bcg)
        remove_nops(bcg)
        new_co = bcg.get_code()

        with open("poc-deobf.py.pyc", "wb") as pyc_deobf:
            pyc_deobf.write(pyc_header)
            marshal.dump(new_co, pyc_deobf)

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

הקוד דיי פשוט: אנחנו משיגים את אובייקט הקוד מה-pyc, עוברים על כל הוראה בבייטקוד (nodes) והופכים את הוראות הזבל ל-NOPים, אחר כך אנו מסירים את כל ה-NOP-ים מהבייטקוד, ולבסוף שומרים את אובייקט הקוד החדש לקובץ poc-deobf.py.pyc.

כעת נוכל להשתמש בכלי כמו uncompyle6 כדי להשיג decompilation.

אפשר להתקין את המודול דרך pip (pip install uncomyple6). נריץ את הפקודה הבאה:

```
uncompyle6 poc-deobf.py.pyc > poc.py
```

ולשמחתנו הרבה נקבל סקריפט פייתון ללא בעיות:

```
import sys
import random
__version__ = 'Flare-On ultra python obfuscater 2000'
target = random.randint(1, 101)
count = 1
error_input = ''
while True:
    print '(Guesses: %d) Pick a number between 1 and 100:' % count,
    input = sys.stdin.readline()
    try:
        input = int(input, 0)
    except:
        error_input = input
        print 'Invalid input: %s' % error_input
        continue

    if target == input:
        break
    if input < target:
        print 'Too low, try again'
    else:
        print 'Too high, try again'
    count += 1

if target == input:
    win_msg = 'Wahoo, you guessed it with %d guesses\n' % count
    sys.stdout.write(win_msg)
if count == 1:
    print 'Status: super guesser %d' % count
    sys.exit(1)
if count > 25:
    print 'Status: took too long %d' % count
    sys.exit(1)
else:
    print 'Status: %d guesses' % count
if error_input != '':
    tmp = ''.join((chr(ord(x) ^ 66) for x in error_input)).encode('hex')
    if tmp != '312a232f272e27313162322e372548':
        sys.exit(0)
    stuffs = [67, 139, 119, 165, 232, 86, 207, 61, 79, 67, 45, 58, 230, 190, 181,
74, 65, 148, 71, 243, 246, 67, 142, 60, 61, 92, 58, 115, 240, 226, 171]
    import hashlib
    stuffer = hashlib.md5(win_msg + tmp).digest()
    for x in range(len(stuffs)):
        print chr(stuffs[x] ^ ord(stuffer[x % len(stuffer)])),

    print
```

להלן סקירה של קוד האתגר:

כמות הניחושים צריכה להיות יותר מאחד ופחות מ-25. אנחנו צריכים להזין ערך לא מספרי מסוים כדי למלא את error_input. על התווים של error_input מבוצע XOR עם המספר 66, הופכים את הבתים למחרוזת הקס, והתוצאה נבדקת עם המחרוזת הבאה:

312a232f272e27313162322e372548

אם נהפוך את האלגוריתם הזה בצורה הזו:

```
>>> import binascii
>>> ''.join((chr(ord(x) ^ 66) for x in
binascii.unhexlify('312a232f272e27313162322e372548')))
'shameless plug\n'
```

נקבל כי אנחנו צריכים להזין "shameless plug" בזמן הניחושים.

יוצרים האש MD5 מהשרשור של win_msg ו-tmp, מה שאומר שצריכים לנחש את המספר הרנדומלי במספר ספציפי של ניחושים, משום ש-win_msg משתנה בהתאם למספר הניחושים. אחר כך משתמשים באותו האש כמפתח בהצפנת XOR ל-ciphertext (stuffs), ומדפיסים מחרוזת שלכאורה היא סיסמת האתגר. נשתמש ב-brute force קצרצר על מנת להגיע למספר הניחושים הדרוש:

```
import sys, string, hashlib

def is_printable_string(stringy):
    return all(c in string.printable for c in stringy)

tmp = "312a232f272e27313162322e372548"
stuffs = (67, 139, 119, 165, 232, 86, 207, 61, 79, 67, 45, 58, 230, 190, 181, 74,
65, 148, 71, 243, 246, 67, 142, 60, 61, 92, 58, 115, 240, 226, 171)

for i in range(2,25):
    stuffer = hashlib.md5(b"Wahoo, you guessed it with {0} guesses\n{1}".format(i,
tmp)).digest()
    plaintext = ""
    for x in range(len(stuffs)):
        plaintext += chr(stuffs[x] ^ ord(stuffer[x % len(stuffer)]))
    if (is_printable_string(plaintext)): # plaintext is probably printable
        print("Correct guess number: {0}\nPassword: {1}".format(i, plaintext))
```

והפלט הוא:

```
Correct guess number: 11
Password: 1mp0rt3d_pygu3ss3r@flare-on.com
```

וזהו זה! השגנו את הסיסמה המיוחלת.

דרך נוספת לפתרון

אמנם הגענו לסוף האתגר, אך ארצה להציג אף עוד דרך בה נוכל לבחור על מנת לפתור את האתגר. הסיבה לכך היא שאחת ממטרות המאמר היא הרחבת אופקים, ויכולת גישה לבעיות במספר דרכים תורמת רבות בהנדסה הפוכה. בנוסף, שיטה זו אינה תלויה בכלים מצד שלישי, אלא משתמשת רק במה שפייתון כבר מספקת לנו.

באתגר הזה השתמשו ב-MD5 ו-XOR על מנת להגיע לטקסט הגלוי. היה לנו פשוט מאוד לרשום סקריפט שמבצע את אותה פעולה על מנת להגיע לפתרון. אך מה אם היו משתמשים בפונקציית האש שאתם לא מצליחים לזהות? או אם קוד ה-decryption היה מסורבל ומסועף? בהתחשב בכך שעם ניתוח בעזרת dis נוכל להגיע למסקנה שכמות הניחושים אמורה להיות בין 2 ל-24, וגם למסקנה שאנו אמורים להזין "shameless plug" בעת לולאת הניחושים, נוכל פשוט לשחק את המשחק (עם כמות נדיבה של רמאות), ולהגיע לפתרון בכך שנבחון את הפלט המתקבל בכל כמות ניחושים שבטווח.

במחשבה ראשונה, זה לא נשמע כמו רעיון מזהיר: בהחלט לא נרצה לנסות לנחש מספר רנדומלי בטווח 1–100 בשני ניחושים, וגם המחשבה שבמקרה הכי גרוע נצטרך לעשות זאת עבור 23 מספרים לא מעודדת במיוחד. הדבר יקח המון זמן. פה הרמאות נכנסת למשחק: מה אם נדפיס את המספר שהוגרל? ובכן, אז כל העניין יתקצר משמעותית!

נבצע זאת בעזרת שינוי הבייטקוד עצמו. נתחיל בעבודת בלשנות קצרה.

חשוב: אשתמש בקובץ ה-pyc המקורי כאן, אשר כולל את קוד הזבל. הסרתי מטקסט ה-disassembly את קוד הזבל על מנת לחסוך במקום ובשביל בהירות.

בתחילת הקוד שמים את המספר המוגרל במשתנה target:

```
 6           39 LOAD_NAME                1 (random)
             46 LOAD_ATTR                3 (randint)
             49 LOAD_CONST               3 (1)
             52 LOAD_CONST               4 (101)
             55 CALL_FUNCTION            2
             58 STORE_NAME               4 (target)-
```

יעד מצוין להדפסת המספר המוגרל (target) הוא בהודעת השגיאה שנקבל כשנזין "shameless plug" (או כל ערך לא מספרי):

```
17          174 LOAD_CONST               8 ('Invalid input: %s')
            178 LOAD_NAME                6 (error_input)
            181 BINARY_MODULO
            182 PRINT_ITEM
            183 PRINT_NEWLINE
```

זה יעד נוח משום שההדפסה הזו מקבלת כארגומנט משתנה להצגה, וגם משום שההדפסה האחרת בלולאת הניחושים אשר מקבלת ארגומנט:

```
'(Guesses: %d) Pick a number between 1 and 100:' % count
```

מספקת לנו מידע שימושי והוא כמות הניחושים שביצענו.

השגנו את המידע הנחוץ לנו בשני קטעי ה-disassembly שלמעלה. עכשיו אנו צריכים קצת רקע לגבי המודול types.

כידוע לנו, בפייתון כמעט הכל הוא אובייקט: פונקציות, מודולים, גנרטורים, מספרים שלמים, וכו'. המודול הסטנדרטי types מכיל חלק מהטיפוסים של אותם אובייקטים. פונקציות מובנות כמו int ו-list הן בסה"כ אלטרנטיבה נוחה לבנאים (constructors) types.IntType() ו-types.ListType(). אמחיש זאת:

```
>>> import types
>>> list == types.ListType
True
>>> types.FloatType() == 0.0
True
>>> types.DictType() == {}
True
>>> types.TupleType([1,2,3]) == (1,2,3)
True
>>> type(5) == types.IntType
True
>>> type == types.TypeType
True
```

המודול הזה, כממשיך המסורת של המודולים הסטנדרטיים אשר סקרנו כאן, מאוד תלוי בגרסת הפייתון שבשימוש: לפעמים מורידים ממנו טיפוסים, לפעמים מוסיפים. למשל, בפייתון 3 והלאה הסירו מהמודול מחלקות/טיפוסים שכבר יש דרך נוחה לגשת אליהם, כמו types.ListType ו-types.IntType, וזאת משום שאין בהם צורך. מה שמעניין אותנו הוא שהמודול מכיל בנאים ליצירת טיפוסים פנימיים אשר אין להם קיצורים/שמות נוספים. הטיפוס הרלוונטי לנו הוא types.CodeType, והוא נשאר במודול בכל גרסת פייתון המופיעה בדוקומנטציה בזמן הכתיבה.

הבנאי types.CodeType() מאפשר לנו ליצור אובייקט קוד בדרך תכנותית. הוא מקבל כארגומנטים את כל מה שמרכיב אוביקט קוד: פיסת קוד, סדרת קבועים, סדרת שמות משתנים, גודל מחסנית, ועוד, ומחזירה אובייקט קוד חדש. החתימה של הבנאי היא כזו:

```
CodeType(argcount, nlocals, stacksize, flags, code, consts,
         names, varnames, filename, name, firstlineno,
         lnotab, freevars=None, cellvars=None)
```

בפייתון 3 חתימת הפרמטרים של types.CodeType() השתנתה מעט בכך שנוסף עוד פרמטר, kwonlyargcount, במקום השני בחתימה (אחרי argcount).

נשתמש בבנאי זה על מנת ליצור אוביקט קוד חדש שיכיל את שינויינו. אחרי ההקדמה זו, נוכל לערוך את התוכנה בעזרת פייתון:

```
import struct
import marshal
from types import CodeType
from dis import opname

with open("poc.py.pyc", "rb") as f:
    pyc_header = f.read(8)
    co = marshal.load(f) # get code object

    new_co_consts = list(co.co_consts)
    # Append our new constant to co_consts
    new_co_consts.append("Psst, I've got what you need: %s")
    new_co_consts = tuple(new_co_consts)

    # Just to make sure we are working on the correct pyc
    # Notice that we use dis.opname, the opposite of dis.opmap
    assert opname[ord(co.co_code[174])] == "LOAD_CONST"
    assert opname[ord(co.co_code[178])] == "LOAD_NAME"
    # Get modifiable bytecode
    new_co_code = bytearray(co.co_code)
    # Generate a correct index argument for LOAD_CONST. little endian
    msg_index_arg = struct.pack("<H", len(new_co_consts) - 1)
    # Change LOAD_CONST's argument to our message
    new_co_code[174 + 1: 174 + 3] = msg_index_arg
    # Change LOAD_NAME 6 (error_input) to LOAD_NAME 4 (target)
    new_co_code[178 + 1: 178 + 3] = struct.pack("<H", 4)
    new_co_code = bytes(new_co_code)

    # Create a new code object with our modified data
    new_co = CodeType(co.co_argcount,
                co.co_nlocals, co.co_stacksize, co.co_flags,
                new_co_code, new_co_consts, # Change occurs here
                co.co_names, co.co_varnames, co.co_filename,
                co.co_name, co.co_firstlineno, co.co_lnotab,
                co.co_freevars, co.co_cellvars)

    with open("poc-deobf2.py.pyc", "wb") as de_f:
        de_f.write(pyc_header)
        marshal.dump(new_co, de_f)
```

הקוד מוסיף לאובייקט הקוד עוד קבוע: "Psst, I've got what you need: %s". אחר כך הוא משתמש ב-offsets של ההוראות שהצגתי קודם, 174 ו-178, על מנת לשנות את הארגומנט שלהן, אשר נמצא בשני הבתים החופפים. ולבסוף, יוצר אוביקט קוד חדש ושומר אותו לקובץ.

כעת, בעזרת התוכנה המעודכנת, נוכל לשחק בקלות יתרה:

```
C:\Windows\system32\cmd.exe

C:\Users\Admin\Downloads\flare-on\6\solution>py -2 poc-deobf2.py.pyc
(Guesses: 1) Pick a number between 1 and 100:shameless plug
 Psst, I've got what you need: 87
(Guesses: 1) Pick a number between 1 and 100:87
Wahoo, you guessed it with 1 guesses
Status: super guesser 1
```

ننסה את כל טווח כמויות הניחושים:

```
C:\Windows\system32\cmd.exe

C:\Users\Admin\Downloads\flare-on\6\solution>py -2 poc-deobf2.py.pyc
(Guesses: 1) Pick a number between 1 and 100:shameless plug
 Psst, I've got what you need: 61
(Guesses: 1) Pick a number between 1 and 100:0
 Too low, try again
(Guesses: 2) Pick a number between 1 and 100:61
Wahoo, you guessed it with 2 guesses
Status: 2 guesses
{ ∈ l ◄   R ` y B : ? ‖ W h ‼ æ y ± \ G β G ! x 0 % ( ä A 4 ⊢
```

נבחין כי ה-decryption נכשל, ולכן התוכנה מציגה מחרוזת לא הגיונית. נמשיך בתהליך עד שנגיע לכמות הדרושה:

```
C:\Windows\system32\cmd.exe
C:\Users\Admin\Downloads\flare-on\6\solution>py -2 poc-deobf2.py.pyc
(Guesses: 1) Pick a number between 1 and 100:shameless plug
 Psst, I've got what you need: 25
(Guesses: 1) Pick a number between 1 and 100:0
 Too low, try again
(Guesses: 2) Pick a number between 1 and 100:0
 Too low, try again
(Guesses: 3) Pick a number between 1 and 100:0
 Too low, try again
(Guesses: 4) Pick a number between 1 and 100:0
 Too low, try again
(Guesses: 5) Pick a number between 1 and 100:0
 Too low, try again
(Guesses: 6) Pick a number between 1 and 100:0
 Too low, try again
(Guesses: 7) Pick a number between 1 and 100:0
 Too low, try again
(Guesses: 8) Pick a number between 1 and 100:0
 Too low, try again
(Guesses: 9) Pick a number between 1 and 100:0
 Too low, try again
(Guesses: 10) Pick a number between 1 and 100:0
 Too low, try again
(Guesses: 11) Pick a number between 1 and 100:25
Wahoo, you guessed it with 11 guesses
Status: 11 guesses
1 m p 0 r t 3 d _ p y g u 3 s s 3 r @ f l a r e - o n . c o m

C:\Users\Admin\Downloads\flare-on\6\solution>
```

הידד! 11 ניחושים.

כמה הערות נוספות בנוגע לפתרון:

ממש לא היינו חייבים לשנות את מחרוזת השגיאה. דבר זה רק האריך את זמן הפתרון. המטרה של כך הייתה להדגים איך אפשר להזריק ל-pyc מידע משלנו. אם נרצה להיות פרקטיים, במקרה זה נשנה רק את הארגומנט של מחרוזת השגיאה מהקלט למספר המוגרל.

כדי לפתור את האתגר בצורה אף עוד יותר מהירה, היינו יכולים להשתמש בעורך הקס כדי לבצע את השינוי: קודם כל היינו יוצרים חתימה של הבייטקוד מסביב להוראה שאנחנו רוצים לשנות, וזאת בעזרת שימוש ב-dis.opmap, ואז מבצעים חיפוש והחלפה של הארגומנט בעורך. בדרך הזו אני פתרתי את האתגר.

יש לציין שביצענו כאן פאטצ'ינג מאוד "נוח", שהותאם לקוד של התוכנה עצמה על מנת לא לשנות אותה בצורה משמעותית, וזאת כדי לחסוך בדברים כמו הוספה והסרה של הוראות—מה שכמובן מעלה את רמת המורכבות. לפי הזן של פייתון: "Simple is better than complex".

סיכום

במאמר זה למדנו שלל מידע על python internals ועל שיטות שיעזרו לנו להנדס לאחור קבצי פייתון מהודרים. הכרנו את הכלים unpy2exe ו-uncompyle6, המודולים dis, marshal, types, ו-bytecode_graph, ונחשפנו לקצת מקוד הביניים של CPython. אני מקווה כי במאמר זה הצלחתי להעביר יותר מאשר פתירת אתגר מסוים, אלא גם ידע שיאפשר לכם לגשת ולפתור בעיות דומות בהצלחה.

על המחבר

מחבר המאמר הינו לירן פאר, ליצירת קשר ניתן לפנות ל:

l.peer@protonmail.com

או בערוץ ה-IRC: #reversing בשרת NiX.

מקורות נוספים לעיון

- סקירה על התמודדות עם בייטקוד פייתון מעורפל וגם הסבר על איך בדיוק קוד זבל יכול לשבור decompilation, מ-FireEye:

https://www.fireeye.com/blog/threat-research/2016/05/deobfuscating_python.html

- סקירות נרחבת על code objects:

https://www.quora.com/What-is-a-code-object-in-Python

https://tech.blog.aknin.name/2010/07/03/pythons-innards-code-objects/

- מכונות ווירטואליות מבוססות מחסנית נגד כאלו מבוססות אוגרים, וה-Dalvik VM (קצת פחות רלוונטי ל-CPython):

http://www.codeproject.com/Articles/461052/Stack-based-vs-Register-based-Virtual-Machine-Arch

- הדוקומנטציה של המודול marshal:

https://docs.python.org/2/library/marshal.html

- סקירה על הפורמט הבינארי של marshal:

http://demoseen.com/blog/2010-02-20_Python_Marshal_Format.html

- הסבר נרחב על אופן השימוש ב-types.CodeType():

http://stackoverflow.com/questions/16064409/how-to-create-a-code-object-in-python