Digital Whisper
Technologic Papers

חלק ג' - Practical Python Internals

חיפוש באגים, וקוד שמשנה את עצמו

מאת אלי קסקי

הקדמה

זהו המאמר השלישי בסדרת מאמרים שבה אני מציג אספקטים שונים במימוש של CPython. במאמרים הקודמים, לצורך הליך הלמידה ביצענו שינויים בקוד המקור של CPython והוספנו פונקציות חדשות ומגניבות, ובכך למעשה יצרנו גרסאות Python משלנו. זו לא חוכמה גדולה לעשות דברים מגניבים בדרך הזו. הרבה יותר מרשים לעשות דברים על גרסאות Python רשמיות!

במאמר זה נשתמש בידע שצברנו במאמר הקודם, בעיקר על איך אובייקטים ב-Python מיוצגים בזיכרון, כדי לכתוב קוד שמשנה את ה-Bytecode של עצמו בזמן ריצה:

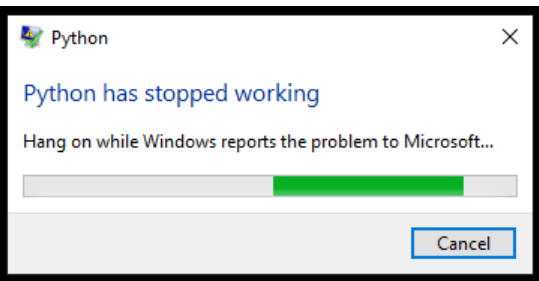
```
def foo(a, b):
    return 2 * a + b

print("before: foo(1, 2) =", foo(1, 2))
patch_function_constant(foo, 2, 3)
print("after:  foo(1, 2) =", foo(1, 2))
```

```
C:\Users\Eli\Desktop>python patch_const.py
before: foo(1, 2) = 4
after:  foo(1, 2) = 5
```

וגם נעבור על קוד המקור של אובייקטים ב-CPython במטרה למצוא התנהגויות מעניינות ובאגים פוטנציאליים:

```
>>> class HashCollision:
...  def __init__(self, s):
...   self.s = s
...  def __hash__(self):
...   return 1337
...  def __eq__(self, other):
...   self.s.add(self.s.pop() * 1)
...   return False
...
>>> s=set()
>>> x=HashCollision(s)
>>> s.add(1337)
>>> s & {x, 1337}
```



מעבר על הקוד של CPython

חמוש בידע מהמאמר הקודם על איך אובייקטים עובדים, החלטתי להסתכל על מימושים של סוגי הטיפוסים ברירת המחדל ב-Python, במטרה למצוא מקרי קצה או באגים פוטנציאליים. סידרתי אותם לפי רמת קושי משוערת: tuple, list, set, dictionary. האובייקטים של tuple ו-list די פשוטים, ומרפרוף עליהם לא מצאתי משהו מעניין במיוחד.

כשהגעתי לטיפוס set ועברתי על מספר פונקציות בסיסיות (בקובץ setobject.c), תפסה את עיני שורה מעניינת בפונקציית הוספת איבר ל-set. נסו למצוא אותה כאן:

```
107  static int
108  set_add_entry(PySetObject *so, PyObject *key, Py_hash_t hash)
109  {
110    restart:
111
112      mask = so->mask;
113      i = (size_t)hash & mask;
114      freeslot = NULL;
115      perturb = hash;
116
117      while (1) {
118          entry = &so->table[i];
119          probes = (i + LINEAR_PROBES <= mask) ? LINEAR_PROBES: 0;
120          do {
121              if (entry->hash == 0 && entry->key == NULL)
122                  goto found_unused_or_dummy;
123              if (entry->hash == hash) {
124                  PyObject *startkey = entry->key;
125                  assert(startkey != dummy);
126                  if (startkey == key)
127                      goto found_active;
128                  if (PyUnicode_CheckExact(startkey)
129                      && PyUnicode_CheckExact(key)
130                      && _PyUnicode_EQ(startkey, key))
131                      goto found_active;
132                  table = so->table;
133                  Py_INCREF(startkey);
134                  cmp = PyObject_RichCompareBool(startkey, key, Py_EQ);
135                  Py_DECREF(startkey);
136                  if (cmp > 0)
137                      goto found_active;
138                  if (cmp < 0)
139                      goto comparison_error;
140                  if (table != so->table || entry->key != startkey)
141                      goto restart;
142                  mask = so->mask;
143              }
144              else if (entry->hash == -1) {
145                  assert (entry->key == dummy);
146                  freeslot = entry;
147              }
148              entry++;
149          } while (probes--);
150          perturb >>= PERTURB_SHIFT;
151          i = (i * 5 + 1 + perturb) & mask;
152      }
```

פונקציה זו מקבלת אובייקט set, אובייקט "מפתח" שצריך להכניס ל-set, ואת ה-hash של אובייקט המפתח. הפונקציה בודקת האם המפתח כבר קיים ב-set, ואם לא - אז מוסיפה אותו. היא מבצעת זאת על ידי חישוב אינדקס מתוך ה-hash, והשוואת האובייקט לאיברים שקיימים במערך פנימי בסביבת האינדקס הזה. ניתן לקרוא עוד על אלגוריתם זה בתיעוד שבקובץ אבל לא בזה נתעמק.

השורה שתפסה לי את העין היא שורה 140, ושורה 141 שאחריה. לאחר שנמצא אובייקט עם hash זהה לאובייקט שאנחנו רוצים להכניס (שורה 123), ויצא שהאובייקטים שונים (שורה 126) כלומר יש התנגשות ב-hash, ולא מדובר באובייקט מסוג מחרוזת, אז יש קריאה לפונקציה PyObject_RichCompareBool. אם ערך החזרה של הפונקציה הוא 0, כלומר השוואה מחזירה תשובה שלילית, אז כשמגיעים לשורה 140 בודקים האם ה-set השתנה. אם הוא אכן השתנה אז חוזרים להתחלה ועושים את כל התהליך מחדש. שאלתי את עצמי למה שלאחר השוואה כל ה-set ישתנה? ולמה זה גורם כך שמתחילים הכל מהתחלה? מה יקרה אם תמיד ה-set ישתנה ונישאר לנצח תקועים בלולאה של "goto restart"? השאלות האלה הובילו אותי לחקור קצת על השוואות בין אובייקטים ב-CPython.

איך מתבצעת השוואת אובייקטים ב-CPython

לפעמים Python צריך לבדוק האם שני אובייקטים שווים. למשל בבדיקה האם אובייקט נמצא ברשימה, מאחורי הקלעים מתבצע מעבר על הרשימה איבר-איבר, ומתבצעת השוואה של האובייקט לאיבר הנוכחי. ב-CPython קיימת אופטימיזציה על השוואות בין אובייקטים, והיא עובדת כך:

1. אם שני האובייקטים מצביעים לאותה כתובת - אז הם שווים.
2. אם ערכי ה-hash של שני האובייקטים שונים - אז הם שונים.
3. השוואת האובייקטים עם האופרטור == והחזרת התוצאה.

אפשרות 1 היא מהירה מאוד - בסך הכל השוואה של מצביעים. אפשרות 2 גם מהירה מאוד - באופן דומה, השוואה של מספר. Fun Fact - נכון להיום, בברירת המחדל, ה-hash של אובייקט מוגדר להיות הכתובת שלו - מוסטת ימינה 4 ביטים:

```
>>> class MyClass:
...  pass
...
>>> x = MyClass()
>>> hex(id(x))
'0x243a0530890'
>>> hex(hash(x))
'0x243a053089'
>>> hash(x) == id(x) >> 4
True
>>>
```

לעומת זאת אפשרות 3 יכולה להיות איטית מאוד, ולכן היא המוצא האחרון. מצאתי סרטון שמסביר על הביצועים של שיטת השוואה זו, ומאמר שמציג אותה בצורה ברורה.

שיטת ההשוואה הזו מניחה שה-hash של אובייקט נשאר קבוע ולא משתנה אף פעם. האם זה אומר שאם נשבור את ההנחה הזאת ונשנה את ה-hash מדי פעם אז יהיה אפשר למשל להכניס איבר ל-set פעמיים? התשובה לכך היא כן!

```
class DynamicHash:
  def __init__(self):
  self.i = 0

def __hash__(self):
  self.i += 1
  return self.i

x = DynamicHash()
s = set()
s.add(x)
s.add(x)
```

```
>>> class DynamicHash:
...  def __init__(self):
...   self.i = 0
...
...  def __hash__(self):
...   self.i += 1
...   return self.i
...
>>> x = DynamicHash()
>>> s = set()
>>> s.add(x)
>>> s.add(x)
>>> s
{<__main__.DynamicHash object at 0x00000243A0531290>,
 <__main__.DynamicHash object at 0x00000243A0531290>}

>>> len(s)
2
>>>
```

המחלקה מוגדרת כך שאף פעם אין שוויון ב-hash כשמשווים בין אובייקטים. כשמכניסים את האובייקט ל-set בפעם הראשונה, מחושב לו אינדקס מתוך ה-hash הראשון של האובייקט, ובמימוש הפנימי של set האובייקט נשמר באינדקס הזה במערך. כשמכניסים את האובייקט ל-set בפעם השניה, מחושב אינדקס אחר מתוך ה-hash השני של אותו האובייקט. האינדקס החדש פנוי במערך ולכן האובייקט נשמר שוב במערך באינדקס שונה מזה שחושב בפעם הראשונה.

נראה שאכן אפשר לעבוד על Python ולעשות דברים שלא התכוונו שנעשה.

בחזרה להתנהגות הבעייתית ב-set

אם נחזור בחזרה לשורה שתפסה לי את העין, אולי עכשיו קצת יותר ברור למה ש-set ישתנה אחרי הקריאה לפונקציה PyObject_RichCompareBool. פונקציה זו עלולה בסופו של דבר להגיע לאפשרות 3 בהשוואה ולקרוא למימוש של __eq__ בקוד ה-Python שהמשתמש כתב. המשתמש יכול בעצם לעשות בפונקציה הזו מה שהוא רוצה, ובין היתר - לשנות את ה-set עצמו.

התנאי הראשון ב-if שבשורה הבעייתית הוא שהמערך עצמו השתנה - כלומר ה-set גדל או קטן מספיק כדי להקצות מחדש את המערך. התנאי השני הוא שהאיבר שבמיקום של ה-hash המתאים השתנה. לצורך הבדיקה אני הלכתי על האפשרות השניה - לשנות את האובייקט עם ה-hash המתאים. אחרי קצת ניסוי וטעיה כתבתי קוד שגורם לכך:

```
class HashCollision:
 def __init__(self, s):
  self.s = s
 def __hash__(self):
  return 1337
 def __eq__(self, other):
  print("comparing")
  self.s.add(self.s.pop() * 1)
  return False

s=set()
x=HashCollision(s)
s.add(1337)
s.add(x)
```

בקוד זה יש set שמוכנס אליו הקבוע 1337. ערך ה-hash של מספר קבוע הוא המספר עצמו. מוגדרת מחלקה שה-hash של כל אובייקט שלה הוא 1337. אנחנו יוצרים אובייקט x מהמחלקה ומכניסים אותו גם ל-set. מכיוון שה-hash של x זהה ל-hash של האיבר הקיים ב-set שאנחנו משווים אליו, אנחנו עוברים את התנאי בשורה 123 (התנגשות ב-hash).

מכיוון ש-x שונה ממש מהקבוע 1337 שנמצא כבר ב-set, אנחנו עוברים את התנאי בשורה 126. מכיוון ש-x לא מסוג מחרוזת, יש קריאה לפונקציה __eq__ עם הקבוע 1337. בפונקציה זו אנחנו מוציאים את הקבוע הזה מה-set, ומכניסים אותו מחדש ל-set. הכפל ב-1 גורם לכך שיווצר אובייקט חדש למספר 1337 (אחרת, אותו המקום בזיכרון ימוחזר ונישאר עם מצביע זהה). לבסוף הפונקציה __eq__ מחזירה False כדי לציין שהאובייקטים שונים, זאת כדי להגיע לשורה 140 הבעייתית. הפעם, התנאי השני יתקיים ונגיע לשורה 141, מה שיגרום לכל הפונקציה לרוץ מחדש, ולכל התהליך הזה לרוץ מההתחלה.

לכאורה קטע הקוד שיצרנו הוא תמים - הכנסה של איבר ל-set, הוצאה והכנסה של איבר אחר. הדבר ההגיוני לצפות לו הוא שגם כשיש התנגשות ב-hash, אחרי שההשוואה של האובייקט מחזירה תוצאה שלילית (איברים שונים), האיבר x פשוט יכנס ל-set והתהליך יסתיים. בפועל כשמריצים את הקוד אכן ה-Interpreter נכנס ללולאה אינסופית:

```
Python 3.11 (64-bit)
>>> class HashCollision:
...  def __init__(self, s):
...   self.s = s
...  def __hash__(self):
...   return 1337
...  def __eq__(self, other):
...   print("comparing")
...   self.s.add(self.s.pop() * 1)
...   return False
```

```
>>> s=set()
>>> x=HashCollision(s)
>>> s.add(1337)
>>> s.add(x)
comparing
comparing
comparing
comparing
comparing
comparing
```

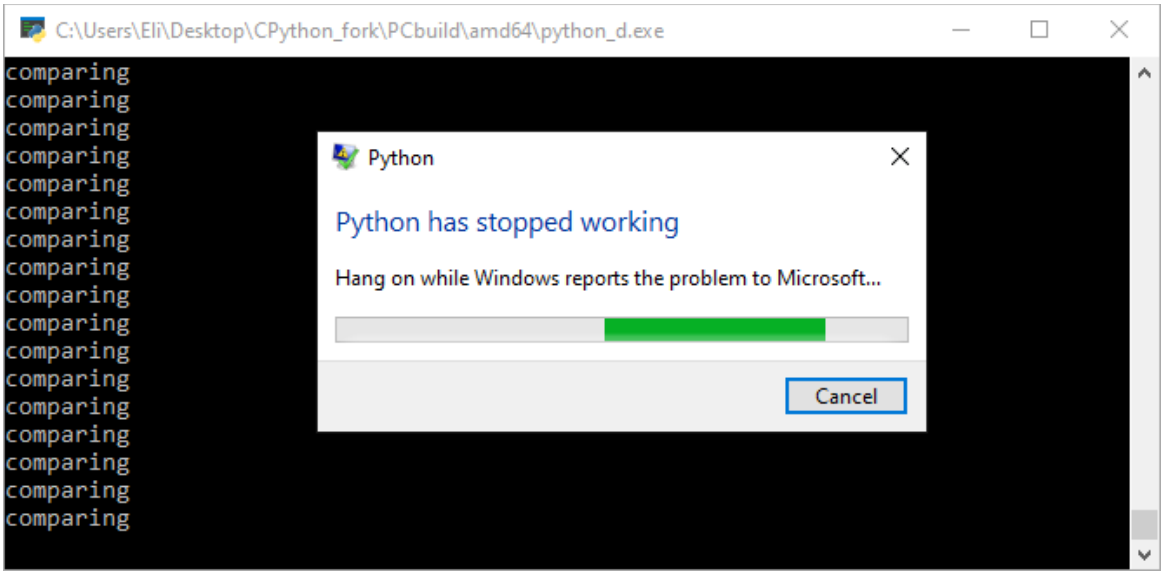
אני לא יודע אם להגדיר את התופעה הזו כבאג, כי אני כן חושב שאם היה שינוי ב-set בזמן שאנחנו מנסים להכניס אליו איבר, לפעמים הגיוני להתחיל חיפוש מחדש. אולי היה אפשר להגביל את מספר ה-restart-ים לכמות מסוימת, או להעלות Exception כש-set משתנה בזמן הכנסה של איבר, או פתרון אחר כלשהו. אם הייתי מפתח של CPython והיו אומרים לי לתקן את זה, לא בטוח שהייתי יודע מה הפתרון העדיף.

עוד התנהגויות של set

בשלב זה חשבתי לעצמי שאולי מה שמצאתי לא באמת מעניין. אז המשכתי להסתכל על קוד של set וגיליתי את אותו התנאי בדיוק בפונקציה אחרת, set_lookkey, שמטרתה לבדוק האם איבר מסוים נמצא ב-set.

```
60  static setentry *
61  set_lookkey(PySetObject *so, PyObject *key, Py_hash_t hash)
62  {
63      while (1) {
64          entry = &so->table[i];
65          probes = (i + LINEAR_PROBES <= mask) ? LINEAR_PROBES: 0;
66          do {
67              if (entry->hash == 0 && entry->key == NULL)
68                  return entry;
69              if (entry->hash == hash) {
70                  PyObject *startkey = entry->key;
71                  assert(startkey != dummy);
72                  if (startkey == key)
73                      return entry;
74                  if (PyUnicode_CheckExact(startkey)
75                      && PyUnicode_CheckExact(key)
76                      && _PyUnicode_EQ(startkey, key))
77                      return entry;
78                  table = so->table;
79                  Py_INCREF(startkey);
80                  cmp = PyObject_RichCompareBool(startkey, key, Py_EQ);
81                  Py_DECREF(startkey);
82                  if (cmp < 0)
83                      return NULL;
84                  if (table != so->table || entry->key != startkey)
85                      return set_lookkey(so, key, hash);
86                  if (cmp > 0)
87                      return entry;
88                  mask = so->mask;
89              }
90              entry++;
91          } while (probes--);
92          perturb >>= PERTURB_SHIFT;
93          i = (i * 5 + 1 + perturb) & mask;
94      }
95  }
```

השורה הבעייתית היא 84, ו-85 שאחריה. בפונקציה זו המצב גרוע יותר מהפונקציה הקודמת, כי הפעם במקום לולאה אינסופית יש קריאה רקורסיבית! זה פוטנציאל ל-Stack Overflow וקריסות, אז אולי זה יותר מעניין? כשמשנים את השורה האחרונה בקוד שלנו מ-s.add(x) ל-x in s, מקבלים:



כשמדבגים רואים שהתהליך אכן קורס בגלל Stack Overflow. אבל, וזה אבל גדול, זה קורה רק ב-Python בגרסת Debug, שאין בה אופטימיזציות. כשמריצים את אותו הקוד בגרסת Python רשמית (שבגרסת Release), מגלים שהתהליך אמנם גם מדפיס אינסוף "comparing", אך אינו קורס. בפועל, גרסאות ה-Python שאנחנו משתמשים בהן ביום יום קומפלו עם המון אופטימיזציות, ואחת מהן היא אופטימיזציית רקורסיית זנב: כשיש פונקציה רקורסיבית שהפעולה האחרונה בה היא הקריאה הרקורסיבית לעצמה, והיא לא מצריכה לשמור את המשתנים המקומיים שבה, אז במקום ליצור frame חדש על המחסנית, "ממחזרים" את ה-frame הנוכחי של הפונקציה ולמעשה קופצים אל תחילתה. במקרה שלנו מתקיימים התנאים לאופטימיזציה זו ולכן התהליך באמת לא קורס אלא רק נכנס לרקורסיה אינסופית, שמתנהגת כמו לולאה אינסופית.

יש לציין שהפונקציה set_lookkey היא די בסיסית ונקראת לבסוף מהרבה פונקציות אחרות, בין היתר set_issuperset, set_intersection, set_isdisjoint, set_difference, ו-set_issubset. לכן ניתן להכנס לרקורסיה האינסופית שראינו בהרבה דרכים שונות. למשל עבור set_issubset אם נחליף את השורה האחרונה בקוד בשורה {x}.issubset(s), שגורמת לחיפוש של x בתוך s, הדבר יגרום כמו קודם לרקורסיה אינסופית. או למשל אם נחליף את השורה האחרונה בשורה {x, 1337}.intersection(s), אז ב-Python בגרסא 3.7 התוכנית פשוט תקרוס (נסו בעצמכם!), וזה כבר יותר מעניין לדעתי.

התנהגות דומה ב-dictionary

הקוד של dictionary מאוד דומה לקוד של set וגם שם קיימת לולאה אינסופית פוטנציאלית אם המילון השתנה בזמן ההשוואה בין אובייקטים שבו. הקוד המתאים:

```
class HashCollision:
 def __init__(self, d):
  self.d = d
 def __hash__(self):
  return 1337
 def __eq__(self, other):
  print("comparing")
  del self.d[1337]
  self.d[1337] = 1
  return False

d = {1337: 1}
x=HashCollision(d)
x in d
# alternatively:
# d[x] = 1
```

וכשמריצים:

```
Python 3.11 (64-bit)
>>> class HashCollision:
...  def __init__(self, d):
...   self.d = d
...  def __hash__(self):
...   return 1337
...  def __eq__(self, other):
...   print("comparing")
...   del self.d[1337]
...   self.d[1337] = 1
...   return False
...
>>> d = {1337: 1}
>>> x=HashCollision(d)
>>> x in d
comparing
comparing
comparing
comparing
comparing
comparing
```

לכאורה גם פה היינו מצפים שכשיש התנגשות ב-hash, ההשוואה של האובייקט תחזיר תוצאה שלילית וכל הבדיקה תסתיים מיד. בפועל נכנסים ללולאה אינסופית.

בנוסף, באופן דומה להכנסת אותו אובייקט פעמיים ל-set, ניתן לבצע את אותו הדבר גם ב-dictionary, עם הקוד:

```
class DynamicHash:
 def __init__(self):
  self.i = 0

 def __hash__(self):
  self.i += 1
  return self.i

x = DynamicHash()
d = dict()
d[x] = 1
d[x] = 2
```

כשמריצים:

```
Python 3.11 (64-bit)
>>> class DynamicHash:
...  def __init__(self):
...   self.i = 0
...
...  def __hash__(self):
...   self.i += 1
...   return self.i
...
>>> x = DynamicHash()
>>> d = dict()
>>> d[x] = 1
>>> d[x] = 2
>>> d
{<__main__.DynamicHash object at 0x0000022208B5BA90>: 1,
 <__main__.DynamicHash object at 0x0000022208B5BA90>: 2}
```

ולפעמים ה-dictionary יזהה שהאובייקט כבר נמצא בו - כשתהיה התנגשות באינדקס של המערך הפנימי שמחושב מערך ה-hash:

```
Python 3.11 (64-bit)
>>> for _ in range(10):
...  print(x in d)
...
False
False
False
False
False
False
True
True
False
False
>>>
```

גילויים שונים נוספים

במהלך החיפוש אחר באגים ותהליך הלמידה שלי, יצא לי להסתכל על הרבה חלקים בקוד של CPython וללמוד כל מיני אנקדוטות מעניינות שלא קשורות לכלום ☺. הנה חלק מהן:

ידוע שערך ה-hash של מספר הוא ערך המספר עצמו. ניתן לראות בקוד שערכי hash מועברים לפונקציות וחוזרים מפונקציות, ובגלל שבשפת C הערך 1- בדרך כלל מייצג שגיאה כלשהי, הוא לא נחשב כערך hash חוקי. לכן הגדירו שיתקיים hash(-1) = -2.

```
Python 3.11 (64-bit)
>>> for i in range(-5, 5):
...  print(i, "-->", hash(i))
...
-5 --> -5
-4 --> -4
-3 --> -3
-2 --> -2
-1 --> -2
0 --> 0
1 --> 1
2 --> 2
3 --> 3
4 --> 4
>>>
```

עוד נקודה נחמדה מהקוד של list, בפונקציה list_dealloc שמשחררת רשימה מהזיכרון, מגלים שהאיברים ברשימה משוחררים מהסוף להתחלה:

```
353  static void
354  list_dealloc(PyObject *self)
355  {
356      PyListObject *op = (PyListObject *)self;
357      Py_ssize_t i;
358      PyObject_GC_UnTrack(op);
359      Py_TRASHCAN_BEGIN(op, list_dealloc)
360      if (op->ob_item != NULL) {
361          /* Do it backwards, for Christian Tismer.
362             There's a simple test case where somehow this reduces
363             thrashing when a *very* large list is created and
364             immediately deleted. */
365          i = Py_SIZE(op);
366          while (--i >= 0) {
367              Py_XDECREF(op->ob_item[i]);
368          }
369          PyMem_Free(op->ob_item);
370      }
```

ובקוד Python:

```
Python 3.11 (64-bit)
>>> class MyClass:
...  def __init__(self, name):
...   self.name = name
...  def __del__(self):
...   print("deleting", self.name)
...
>>> l = [MyClass("First"), MyClass("Second"), MyClass("Third")]
>>> del l
deleting Third
deleting Second
deleting First
>>>
```

דוגמא נוספת היא אופן חישוב גודל המערך שמוקצה ל-set בהתאם לכמות האיברים בו. כדי למנוע יותר מדי התנגשויות hash, יש צורך לבצע Load Balancing מסוים. נחמד לראות את החישוב המדויק של מתי ובכמה להגדיל את ה-set. הנה סוף הפונקציה set_add_entry שמוסיפה איבר ל-set:

```
175    found_unused:
176      so->fill++;
177      so->used++;
178      entry->key = key;
179      entry->hash = hash;
180      if ((size_t)so->fill*5 < mask*3)
181          return 0;
182      return set_table_resize(so, so->used>50000 ? so->used*2 : so->used*4);
```

ניתן לראות שאם המערך קטן מ-50,000 תאים אז מגדילים אותו פי שניים, ואחרת פי ארבע. ויותר מעניין שאם כמות האיברים בפועל ב-set גדולה מ-60% מגודל המערך שמוקצה לו, אז מגדילים את המערך. כותרת ה-commit האחרון שנגע בשורה הזו היא:

```
Reduce load factor (from 66% to 60%) to improve effectiveness of linear probing.
```

די נחמד לראות את חישובי האופטימיזציות האלה, ולהבין לעומק את השיקולים בבחירת ה-Load Factor. באופן דומה ל-set, בתיעוד של dictionary יש אלגוריתם ברמה של מאמר אקדמי על ה-Load Factor הרצוי ואופן ההתמודדות עם התנגשויות ב-hash.

ניתן גם להבין יותר לעומק איך עובדות פעולות שלא מוגדרות היטב - למשל הפונקציה set.pop - איך נבחר האיבר ששולפים מתוך הסט? לפי התיעוד הרשמי מדובר באיבר אקראי, אבל בטח אנחנו יכולים למצוא תיאור יותר טוב מזה. ממעבר על קוד הפונקציה set_pop ניתן לראות שקיים באובייקט set ערך מספרי בשם finger שמציין את האינדקס של האיבר הבא שיש לשלוף במערך הפנימי במימוש של set. נוכל לחשב אותו בעצמנו וכך לחזות את סדר האיברים שישלפו עם קריאות ל-set.pop.

לצורך הדוגמא נסתכל על Python בגרסא 3.11. ערך ה-finger מאותחל להיות 0 ביצירת ה-set, וגדל באחד בכל פעם שנשלף איבר עם pop. כל מה שנצטרך לעשות הוא לדעת איזה איברים נמצאים באיזה אינדקסים במערך של set. האינדקס של אובייקט הוא פשוט ה-hash של האובייקט, מודולו גודל המערך. אם למשל נאתחל set בגודל של 6 איברים, אז יוקצה בשבילו מערך בגודל 32. נוכל לחזות את סדר האיברים כך:

```
elements = [42, 123, True, "digital", "whisper", object()]
s = set(elements)
pop_indexes = [hash(element) % 32 for element in elements]
```

כשמריצים את הקוד רואים שאכן הסדר מתאים לאינדקסים שחישבנו:

```
Python 3.11 (64-bit)
>>> elements = [42, 123, True, "digital", "whisper", object()]
>>> s = set(elements)
>>> pop_indexes = [hash(element) % 32 for element in elements]
>>> pop_indexes
[10, 27, 1, 5, 15, 4]
>>> s.pop()
True
>>> s.pop()
<object object at 0x000001BBA0B64640>
>>> s.pop()
'digital'
>>> s.pop()
42
>>> s.pop()
'whisper'
>>> s.pop()
123
>>>
```

בדוגמא זו האינדקס הקטן ביותר הוא 1, והוא מתאים לאיבר השלישי שהכנסנו - שהוא True, לכן הוא ישלף ראשון. האינדקס הקטן ביותר הבא ברשימה הוא 4, והוא מתאים לאיבר האחרון - שהוא האובייקט, ולכן הוא נשלף שני, וכך הלאה.

מפאת חוסר זמן לקראת פרסום המאמר הייתי צריך להפסיק כאן את החיפוש אחר באגים פוטנציאלים. אני חושב שבהחלט יש מקום להמשיך מנקודה זו ולמצוא התנהגויות בעייתיות ועוד הרבה דברים מעניינים נוספים. ובמעבר חד לנושא הבא של מאמר זה...

קוד שמשנה את עצמו - רקע

שיטה נפוצה של malware-ים להקשות על חוקרי אבטחה להבין מה תוכנית מסוימת עושה - היא שימוש בקוד שמשנה את עצמו בזמן ריצה (Self-modifying code). בשיטה זו חוקר שמסתכל על הקוד בצורה סטטית לא יכול לראות את הקוד ה"אמיתי" שרץ, אלא רק את הקוד שמפענח את הקוד ה"אמיתי" וקורא לו. כדי למצוא את הקוד שניסו להחביא צריך להריץ דינמית את הקובץ, לעבור את הקטע שמפענח את הקוד ה"אמיתי", ואז ניתן לבחון אותו. קוד שמשנה את עצמו מופיע מדי פעם גם ב-CTF-ים. דוגמא לקוד כזה בשפת C היא:

```
int main(int argc, char **argv) {
  char* dest = mmap(0, 0x100, (PROT_READ | PROT_WRITE | PROT_EXEC),
    (MAP_PRIVATE | MAP_ANONYMOUS), -1, 0);
  if ( dest == -1 ) {
    exit(1);
  }
  memcpy(dest, &bytes_blob, 0x100);
  for (int i = 0; i < 0x100; i++)
    dest[i] ^= 0x20;
  (dest)();
  munmap(dest, 0x100);
  return 0;
}
```

בקטע זה, מתבצע מיפוי לזיכרון של איזור עם הרשאות קריאה כתיבה והרצה, מועתקים אליו הבתים ה"מוצפנים", ומבצעים עליהם xor עם הערך הקבוע 0x20. תוצאת החישוב היא קטע קוד "אמיתי", שלאחר מכן מריצים על ידי קריאה (call) לאיזור הזיכרון. בסוף הפונקציה מתבצע ניקוי הזיכרון. חוקר שמסתכל על קוד זה, ועל הבתים ה"מוצפנים" שב-bytes_blob, לא יכול לראות בצורה סטטית מה הקוד שניסו להחביא, כי לא ניתן לפרש את אותם בתים כקוד תקני. כמובן ששיטה זו לא מונעת לחלוטין מהחוקר להבין את הקוד אלא רק מקשה עליו, כי החוקר צריך לדבג את התהליך או לפענח בעצמו את קטע הקוד, כלומר להשקיע יותר מאמצים כדי להבין מה הקוד שרץ בכלל.

אמנם בדוגמא זו החביאו את קטע הקוד בשלמותו, אך זה לא חייב להיות כך. יכול להיות שהבתים ה"מוצפנים" יהיו פונקציה עם קוד אסמבלי תקין לחלוטין שמבצע לוגיקה "לגיטימית" מסוימת. ובתרחיש מסוים הקוד ידרוס חלק מהבתים בפונקציה כך שתתבצע לוגיקה נסתרת, אחרת לגמרי. תרחיש זה קשה אפילו יותר לזיהוי.

בנוסף בדוגמא זו ה"הצפנה" היא xor עם ערך קבוע. ניתן היה להשתמש בהצפנה רצינית יותר כמו למשל AES עם מפתח שהוא hash של קלט מסוים שמגיע מהמשתמש, או הזמן הנוכחי במערכת, או משהו דומה, מה שיקשה על חוקרים אפילו עוד יותר.

קוד Python שמשנה את עצמו

נניח שהיינו רוצים ליצור משהו דומה לקוד שמשנה את עצמו, אבל ב-Python. למשל ליצור סקריפט Python שמבקש סיסמא מהמשתמש, ואם הסיסמא נכונה אז הסקריפט מבצע איזושהי פעולה סודית. אנחנו לא רוצים שיהיה קל להבין מה הפעולה הסודית שמתבצעת, ולכן במקום לפרסם את קוד המקור שלנו בקובץ py., אנחנו מקמפלים את הקוד לקובץ pyc. שמכיל רק את ה-Bytecode המתאים לקוד שלנו, ומפיצים אותו בלבד. אבל כמובן שחוקר שיודע לקרוא Bytecode יוכל עדיין להבין מה הפונקציונליות הסודית שרצינו להחביא.

במאמר הקודם ראינו שבסופו של דבר, קוד Python מתקמפל לפקודות Bytecode בסיסיות יחסית. אם נתייחס אליהן כמו אל פקודות אסמבלי נוכל באופן דומה לשיטה ב-C, ליצור רצף פקודות Bytecode שמשנה את עצמו. ואז אם יבוא חוקר וינסה להבין מתוך קובץ ה-pyc. המקומפל שלנו מה הקוד שלנו עושה, יהיו לו חיים קשים.

נתחיל בניסוי פשוט - נכתוב פונקציה פשוטה יחסית וננסה לשנות אותה ברמת ה-Bytecode בזמן ריצה. למשל פונקציה שמקבלת מספר ומחברת אותו עם הקבוע 1:

```
def foo(x):
    return x + 1
```

לצורך דוגמה זו נעבוד על Python בגרסא 3.7. כדי להדפיס את פקודות ה-Bytecode שהפונקציה מתקמפלת אליהן נשתמש בספריה dis, ובנוסף נדפיס את תוצאת הריצה של הפונקציה על הקבוע 5:

```
import dis
dis.dis(foo)
print("foo(5) =", foo(5))
```

להלן פלט הקוד:

```
C:\Users\Eli\Desktop>C:\Python\Python37\python.exe simple.py
  4           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (1)
              4 BINARY_ADD
              6 RETURN_VALUE
foo(5) = 6
```

עד כה לא מרגש במיוחד. כדי להבין איך ניתן לשנות את הפקודות האלה נצטרך ללמוד קצת על האובייקט שמייצג פונקציה ב-Python.

אובייקט של פונקציה

כזכור כל דבר ב-Python הוא אובייקט. בפרט, גם פונקציה היא אובייקט. אובייקט מסוג function. לאובייקט זה יש מספר שדות, והשדה החשוב ביותר הוא שדה ה-__code__. שדה זה הוא אובייקט מסוג code שמכיל נתונים על הפונקציה שהכרחיים להרצה שלה. למשל שם הפונקציה, מספר הארגומנטים שהיא מקבלת, המשתנים והקבועים שהיא משתמשת בהם ועוד. אבל הכי חשוב - את קוד ה-Bytecode שלה - שנמצא בשדה בשם co_code שהוא מסוג bytes:

```
Python 3.7 (64-bit)
>>> foo.__code__.co_code
b'|\x00d\x01\x17\x00S\x00'
>>>
```

כל שורת Bytecode שראינו בפלט של dis מתאימה למספר בתים בשדה co_code. ניתן למצוא את המיפוי המדויק בין פקודת Bytecode לבין ה-opcode (בית אחד) שמתאים לה על ידי הסתכלות בקוד המקור של CPython, בפרט בקובץ opcode_ids.h. המיפוי הזה בין פקודת Bytecode לערך הבית שמתאים לה, משתנה בין גרסא לגרסא של Python. למזלנו יש דרך קלה לגשת למיפוי זה, ונתקלנו בה גם במאמר הקודם - שימוש במילון opmap שבספריה dis - שמחזיק בדיוק את המיפוי הזה:

```
Python 3.7 (64-bit)
>>> dis.opmap
{'POP_TOP': 1, 'ROT_TWO': 2, 'ROT_THREE': 3, 'DUP_TOP': 4, 'DUP_TOP_TWO': 5, 'NOP': 9, 'UNARY_PO
SITIVE': 10, 'UNARY_NEGATIVE': 11, 'UNARY_NOT': 12, 'UNARY_INVERT': 15, 'BINARY_MATRIX_MULTIPLY'
: 16, 'INPLACE_MATRIX_MULTIPLY': 17, 'BINARY_POWER': 19, 'BINARY_MULTIPLY': 20, 'BINARY_MODULO':
 22, 'BINARY_ADD': 23, 'BINARY_SUBTRACT': 24, 'BINARY_SUBSCR': 25, 'BINARY_FLOOR_DIVIDE': 26, 'B
INARY_TRUE_DIVIDE': 27, 'INPLACE_FLOOR_DIVIDE': 28, 'INPLACE_TRUE_DIVIDE': 29, 'GET_AITER': 50,
'GET_ANEXT': 51, 'BEFORE_ASYNC_WITH': 52, 'INPLACE_ADD': 55, 'INPLACE_SUBTRACT': 56, 'INPLACE_MU
LTIPLY': 57, 'INPLACE_MODULO': 59, 'STORE_SUBSCR': 60, 'DELETE_SUBSCR': 61, 'BINARY_LSHIFT': 62,
 'BINARY_RSHIFT': 63, 'BINARY_AND': 64, 'BINARY_XOR': 65, 'BINARY_OR': 66, 'INPLACE_POWER': 67,
'GET_ITER': 68, 'GET_YIELD_FROM_ITER': 69, 'PRINT_EXPR': 70, 'LOAD_BUILD_CLASS': 71, 'YIELD_FROM
': 72, 'GET_AWAITABLE': 73, 'INPLACE_LSHIFT': 75, 'INPLACE_RSHIFT': 76, 'INPLACE_AND': 77, 'INPL
ACE_XOR': 78, 'INPLACE_OR': 79, 'BREAK_LOOP': 80, 'WITH_CLEANUP_START': 81, 'WITH_CLEANUP_FINISH
': 82, 'RETURN_VALUE': 83, 'IMPORT_STAR': 84, 'SETUP_ANNOTATIONS': 85, 'YIELD_VALUE': 86, 'POP_B
LOCK': 87, 'END_FINALLY': 88, 'POP_EXCEPT': 89, 'STORE_NAME': 90, 'DELETE_NAME': 91, 'UNPACK_SEQ
```

[חלק מהמילון dis.opmap]

לצורך הדוגמא אנחנו נרצה לשנות את הפקודה BINARY_ADD ל-BINARY_SUBTRACT, כדי שבמקום שהפונקציה foo תחבר מספר עם הקבוע 1, היא תחסיר ממספר את הקבוע 1.

לכאורה הדרך פשוטה מאוד - ניתן בקלות ליצור אובייקט bytes עם ה-Bytecode החדש:

```
bytecode = foo.__code__.co_code
old_instruction = dis.opmap["BINARY_ADD"].to_bytes(1, "little")
new_instruction = dis.opmap["BINARY_SUBTRACT"].to_bytes(1, "little")
new_bytecode = bytecode.replace(old_instruction, new_instruction)
print("before:", bytecode)
print("after :", new_bytecode)
```

הפלט הוא:

```
C:\Windows\System32\cmd.exe
C:\Users\Eli\Desktop>C:\Python\Python37\python.exe simple.py
before: b'|\x00d\x01\x17\x00S\x00'
after : b'|\x00d\x01\x18\x00S\x00'
```

אך כשמנסים לכתוב את ה-Bytecode החדש בחזרה לתוך co_code:

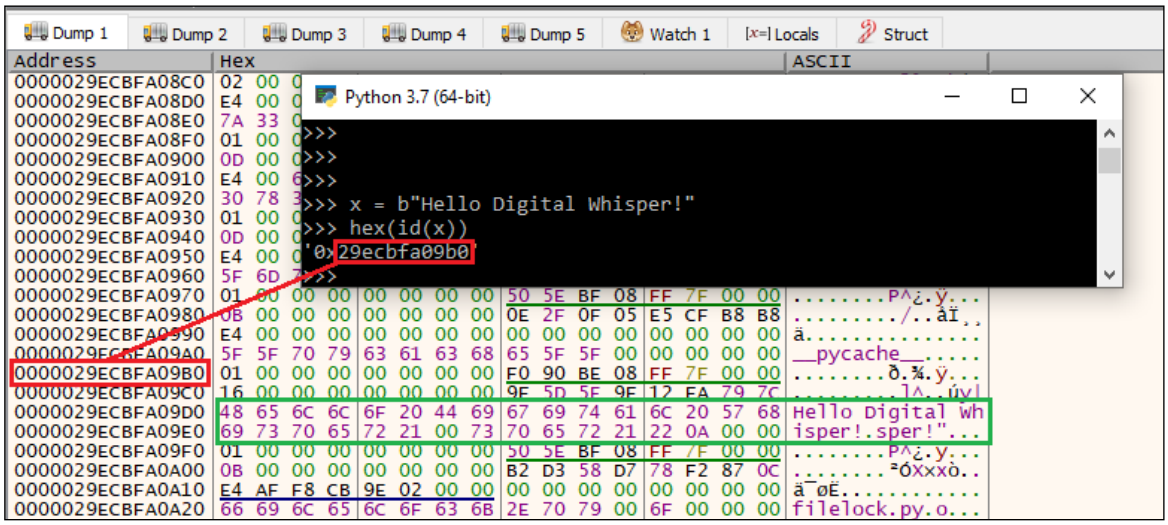
```
foo.__code__.co_code = new_bytecode
```

נתקלים בשגיאה:

```
C:\Users\Eli\Desktop>C:\Python\Python37\python.exe simple.py
before: b'|\x00d\x01\x17\x00S\x00'
after : b'|\x00d\x01\x18\x00S\x00'
Traceback (most recent call last):
  File "simple.py", line 21, in <module>
    foo.__code__.co_code = new_bytecode
AttributeError: readonly attribute
```

לא ניתן לשנות את co_code כי הוא לא ניתן לכתיבה. בנוסף, מכיוון שמדובר באובייקט bytes שהוא immutable ב-Python, גם לא ניתן לכתוב אליו ולשנות בו בתים ספציפיים. נראה שהכותבים של Python לא רצו לאפשר למשתמש לשחק עם מה שקורה מאחורי הקלעים כשהקוד רץ. זו גישה הגיונית שכן מדובר בקוד שנמצא כבר מתחת לשכבת האבסטרקציה של Python ורץ ב-C. שינויים לא אחראיים בו יכולים לגרום לשגיאות זיכרון, קריסות ועוד.

למזלנו, אנחנו חמושים בידע מהמאמר הקודם על איך סוגי אובייקטים ממומשים! קל לבדוק ולראות שאובייקט bytes שומר בזיכרון את מערך הבתים עצמם ב-offset של 0x20 בתים מתחילתו:



לכן אפשר להיות ילדים גדולים ולכתוב ישירות לזיכרון כל ערך שנרצה.

שינויים לאובייקט קוד

בספריה המובנית ctypes קיימת פונקציה בשם memmove שמאפשרת לעשות את זה בדיוק. פונקציה זו מקבלת כתובת יעד, כתובת מקור, וכמות בתים. היא קוראת את כמות הבתים המבוקשת מכתובת המקור, וכותבת אותם לכתובת היעד. כדי להשיג את כתובת הבסיס של האובייקט ניתן להשתמש בפונקציה id כפי שעשינו במאמר הקודם, ולהוסיף לו 0x20 כדי להגיע למערך הבתים עצמו. אבל למעשה אפילו לא צריך לעשות את זה, כי הפונקציה memmove במקרה יודעת לעבוד גם עם אובייקטים מסוג bytes ישירות:

```
from ctypes import memmove
memmove(bytecode, new_bytecode, len(bytecode))
```

לאחר שורות אלה נדפיס שוב את ה-Bytecode של הפונקציה ואת תוצאת ההרצה שלה עם הקבוע 5:

```
C:\Users\Eli\Desktop>C:\Python\Python37\python.exe simple.py
  4           0 LOAD_FAST                0 (x)
              2 LOAD_CONST               1 (1)
              4 BINARY_SUBTRACT
              6 RETURN_VALUE
foo(5) = 4
```

הצלחנו!

הקוד המלא הוא:

```
import dis
def foo(x):
    return x + 1

dis.dis(foo)
print("foo(5) =", foo(5))
bytecode = foo.__code__.co_code
old_instruction = dis.opmap["BINARY_ADD"].to_bytes(1, "little")
new_instruction = dis.opmap["BINARY_SUBTRACT"].to_bytes(1, "little")
new_bytecode = bytecode.replace(old_instruction, new_instruction)
print("before:", bytecode)
print("after :", new_bytecode)

from ctypes import memmove
assert len(bytecode) == len(new_bytecode)
memmove(bytecode, new_bytecode, len(bytecode))

dis.dis(foo)
print("foo(5) =", foo(5))
```

נסו להריץ בעצמכם! ב-Python בגרסא 3.7.

שינויים לאובייקט קוד בגרסאות 3.11 והלאה

ננסה לבצע את אותו השינוי גם ב-Python בגרסא 3.11. בגרסא זו פקודות ה-Bytecode קצת שונות מגרסא 3.7. בפרט הפקודות BINARY_ADD, BINARY_SUBTRACT ועוד, אוחדו לפקודה אחת בשם BINARY_OP:

```
>>> def foo(x):
...  return x + 1
...
>>> import dis
>>> dis.dis(foo)
  1           0 RESUME                   0

  2           2 LOAD_FAST                0 (x)
              4 LOAD_CONST               1 (1)
              6 BINARY_OP                0 (+)
             10 RETURN_VALUE
>>> foo.__code__.co_code
b'\x97\x00|\x00d\x01z\x00\x00\x00S\x00'
```

לפקודה זו מוצמד פרמטר oparg שמציין את סוג הפעולה הבינארית המבוקשת - חיבור, חיסור, כפל, xor וכו'. הפרמטר הזה הוא בית אחד שנמצא מיד לאחר ה-opcode של הפקודה BINARY_ADD ברצף הבתים שמייצגים את ה-Bytecode. בניגוד ל-opcode, לא קיים בספריה dis מיפוי בין שם הפעולה לערך ה-oparg המתאים לה, ולכן נשתמש בערך כקבוע. הערך המתאים לחיבור הוא 0, והערך המתאים לחיסור הוא 10. ניתן למצוא זאת בקובץ opcode.h או על ידי בחינה של הבתים של ה-Bytecode והסתכלות על הבית שנמצא מיד לאחר ה-opcode של הפקודה BINARY_OP. הקוד המתאים הוא:

כשמריצים אותו התוצאה היא:

```
C:\Users\Eli\Desktop>python simple311.py
  3           0 RESUME                   0

  4           2 LOAD_FAST                0 (x)
              4 LOAD_CONST               1 (1)
              6 BINARY_OP                0 (+)
             10 RETURN_VALUE
foo(5) = 6
before     : b'\x97\x00|\x00d\x01z\x00\x00\x00S\x00'
after      : b'\x97\x00|\x00d\x01z\n\x00\x00S\x00'
new co_code: b'\x97\x00|\x00d\x01z\n\x00\x00S\x00'
  3           0 RESUME                   0

  4           2 LOAD_FAST                0 (x)
              4 LOAD_CONST               1 (1)
              6 BINARY_OP               10 (-)
             10 RETURN_VALUE
foo(5) = 6
```

נראה שהבתים אכן נכתבים כמו שצריך, ה-Bytecode אכן משתנה, אך הלוגיקה של הפונקציה לא משתנה. למה זה לא עובד הפעם? זו שאלה ששאלתי את עצמי גם, והיא הובילה אותי לנושא הבא שלדעתי מגיעה לו כותרת בפני עצמו.

בגרסא 3.11 של Python התווספה אופטימיזציה ברמת ה-Bytecode שנקראת Specialization. בגדול, לקחו כמה פקודות Bytecode והוסיפו להן ב-postfix את המילה "_ADAPTIVE". למשל מהפקודה BINARY_OP יצרו פקודה חדשה בשם BINARY_OP_ADAPTIVE. פקודות נוספות שנכללות באופטימיזציה זו הן LOAD_ATTR, LOAD_GLOBAL, CALL ועוד.

ומה מיוחד בפקודות ה"אדפטיביות" החדשות האלו? הפקודה האדפטיבית תבצע בדיוק את אותה הפעולה כמו הפקודה הלא-אדפטיבית המתאימה לה, אך גם תשמור סטטיסטיקה על סוג האובייקטים שהיא רצה עליהם. לאחר מכן, בתהליך שנקרא Specialization, הפקודות האדפטיביות משתנות בזמן ריצה לפקודות יעילות יותר ו"מותאמות אישית" לקוד שרץ באותו רגע.

ניקח כדוגמא את פקודת פעולת החיבור הבינארית BINARY_OP. כאשר הפקודה הזו רצה, בזמן הריצה מאחורי הקלעים מתבצעות הרבה פעולות Overhead שקשורות בבדיקות Object Oriented של Python על הארגומנטים של הפקודה, בדיקות Operator Overloading וכו', עד שלבסוף מגיעים לפונקציה המתאימה ברמת ה-C שאחראית על פעולת החיבור המתאימה לאותם ארגומנטים. אם פקודה זו תרוץ מספר פעמים על ארגומנטים שהם מספרים שלמים לדוגמא, אז בתהליך ה-Specialization הפקודה תשתנה ותהפוך להיות BINARY_OP_ADD_INT. זוהי פקודה יעילה יותר מהפקודה הגנרית.

כשהפקודה הזו רצה, מתבצעת בדיקה יחסית "קלה" של ה-type של שני הפרמטרים שבראש המחסנית, ואם שניהם מסוג של מספר שלם, אז ה-Interpreter מדלג על כל ה-Overhead של הפקודה הגנרית וקורא ישירות לפונקציה המתאימה ב-C. אך אם ה-type של אחד מהארגומנטים הוא לא מספר שלם, אז מתבצעת "נפילה" חזרה למקרה הגנרי ורצה הלוגיקה של הפקודה הגנרית.

אם הפקודה המותאמת אישית "נכשלת" מספר פעמים, היא משתנה בחזרה לפקודה אדפטיבית גנרית. ניתן ללמוד על האופטימיזציה הזו יותר בסרטון שיחת השחרור של גרסא 3.11 שבה מציגים פיצ'רים עיקריים, אך כנראה שההסבר הכי טוב לאופטימיזציה הזו הוא דוגמא אמיתית.

דוגמא אמיתית

כדי להדפיס את הגרסא האדפטיבית של ה-Bytecode, צריך לקרוא לפונקציה dis.dis עם פרמטר adaptive=True (שבברירת מחדל מוגדר להיות False). כך ניתן לראות בפועל את השינוי שקורה ברמת ה-Bytecode, למשל בקטע הקוד הבא:

```
import dis

def add(a, b):
    return a + b

dis.dis(add, adaptive=True)
for _ in range(10):
    add(1, 2)

dis.dis(add, adaptive=True)
for _ in range(80):
    add("a", "b")

dis.dis(add, adaptive=True)
for _ in range(10):
    add("a", "b")

dis.dis(add, adaptive=True)
```

בקטע קוד זה ישנה פונקציה שמקבלת שני פרמטרים, מריצה עליהם את אופרטור החיבור, ומחזירה את התוצאה. כשמריצים את הקוד, זה הפלט שמתקבל, כשכל קריאה ל-dis.dis מודגשת בצבע אחר:

```
3           0 RESUME                   0

4           2 LOAD_FAST                0 (a)
            4 LOAD_FAST                1 (b)
            6 BINARY_OP                0 (+)
           10 RETURN_VALUE
3           0 RESUME_QUICK             0

4           2 LOAD_FAST__LOAD_FAST     0 (a)
            4 LOAD_FAST                1 (b)
            6 BINARY_OP_ADD_INT        0 (+)
           10 RETURN_VALUE
3           0 RESUME_QUICK             0

4           2 LOAD_FAST__LOAD_FAST     0 (a)
            4 LOAD_FAST                1 (b)
            6 BINARY_OP_ADAPTIVE       0 (+)
           10 RETURN_VALUE
3           0 RESUME_QUICK             0

4           2 LOAD_FAST__LOAD_FAST     0 (a)
            4 LOAD_FAST                1 (b)
            6 BINARY_OP_ADD_UNICODE     0 (+)
           10 RETURN_VALUE
```

באדום - ה-Bytecode שמתאים לפונקציה בתחילת העולם, לפני שהיא נקראת. הפקודה BINARY_OP (+) גנרית וללא אופטימיזציה כלשהי.

בירוק - ה-Bytecode לאחר 10 קריאות לפונקציה עם פרמטרים שהם מספרים שלמים. ניתן לראות שהתבצעה אופטימיזציה של הפקודה (+) BINARY_OP הגנרית לפקודת BINARY_OP_ADD_INT המותאמת למספרים שלמים. (וגם התבצעה אופטימיזציה של הפקודה LOAD_FAST).

בכחול - ה-Bytecode לאחר 80 קריאות לפונקציה עם פרמטרים שהם מחרוזות. בכל קריאה כזאת, האופטימיזציה למספרים שלמים שהתבצעה קודם, עם פקודת ה-BINARY_OP_ADD_INT, נכשלת, ומתבצעת "נפילה" לביצוע פעולת חיבור כמו בפקודת חיבור גנרית. לכן לאחר מספר מסוים של כשלונות כאלה, האופטימיזציה מתבטלת ובמקומה מתבצעת חזרה לפקודה גנרית של (+) BINARY_OP_ADAPTIVE.

בסגול - ה-Bytecode לאחר עוד 10 קריאות לפונקציה עם פרמטרים שהם מחרוזות. מתבצעת אופטימיזציה מחדש של הפקודה הגנרית (+) BINARY_OP_ADAPTIVE לפקודה שמותאמת למחרוזות - (+) BINARY_OP_ADD_UNICODE.

כמות הפעמים שקטע קוד צריך לרוץ לפני שהוא עובר אופטימיזציה, מבטל אופטימיזציה, ויוצר אופטימיזציה חדשה, משתנה בהתאם לגרסאת ה-Python שמריצים. לא אכנס לחישוב המדויק במסגרת המאמר. המספרים שהצגתי בדוגמא זו (10 קריאות ראשונות, 80 קריאות שנכשלות, ו-10 קריאות נוספות) הם המספרים שעבדו לי לצורך הדוגמא.

בחזרה לקוד שמשנה את עצמו

כל המבוא הזה על אופטימיזציות של קוד אדפטיבי נועד בין היתר להסביר למה החל מגרסא 3.11 צריך להסתכל על השדה co_code_adaptive_ ולא על co_code כשאנחנו רוצים לדרוס את ה-Bytecode.

באובייקט code בגרסא 3.11, הבתים שמכילים את הקוד האדפטיבי נמצאים בסוף האובייקט עצמו, ולא באובייקט מסוג bytes שקיים אליו מצביע. ניתן לראות זאת על ידי הסתכלות בזיכרון, או על ידי הסתכלות על הגדרת אובייקט code בקובץ code.h.

לכן בניגוד לשדה co_code, כשניגשים לשדה co_code_adaptive_, מקבלים רק עותק של הבתים שמכילים את ה-Bytecode האדפטיבי, שכאמור נמצא במקום אחר בזיכרון. אנחנו רוצים לדרוס את הבתים עצמם ולא את העותק שלהם. לכן ראשית צריך למצוא את ה-offset של הבתים האלה מתחילת אובייקט ה-code, ואז לדרוס אותם.

הקוד לזה נראה כך:

```
import dis

def foo(x):
    return x + 1

def find_offset_to_adaptive_bytecode(code_object):
    # copy entire code object into bytes variable
    size = 0x100
    code_object_bytes = b"\x00" * size
    memmove(code_object_bytes, id(code_object), size)
    # search _co_code_adaptive index in the bytes variable
    return code_object_bytes.index(code_object._co_code_adaptive)

dis.dis(foo, adaptive=True)
print("foo(5) =", foo(5))

OP_ADD = b"\x00"
OP_SUB = b"\x0A"
old_instruction = dis.opmap["BINARY_OP"].to_bytes() + OP_ADD
new_instruction = dis.opmap["BINARY_OP"].to_bytes() + OP_SUB

bytecode = foo.__code__._co_code_adaptive
new_bytecode = bytecode.replace(old_instruction, new_instruction)
print("before                :", bytecode)
print("after                 :", new_bytecode)

from ctypes import memmove
assert len(bytecode) == len(new_bytecode)
offset = find_offset_to_adaptive_bytecode(foo.__code__)
memmove(id(foo.__code__) + offset, new_bytecode, len(bytecode))

print("new _co_code_adaptive:", foo.__code__._co_code_adaptive)
dis.dis(foo, adaptive=True)
print("foo(5) =", foo(5))
```

כשמריצים אותו מקבלים את התוצאה:

```
C:\Windows\System32\cmd.exe
C:\Users\Eli\Desktop>python simple311.py
  3           0 RESUME                   0

  4           2 LOAD_FAST                0 (x)
              4 LOAD_CONST               1 (1)
              6 BINARY_OP                0 (+)
             10 RETURN_VALUE
foo(5) = 6
before                : b'\x97\x00|\x00d\x01z\x00\x00\x00S\x00'
after                 : b'\x97\x00|\x00d\x01z\n\x00\x00S\x00'
new _co_code_adaptive: b'\x97\x00|\x00d\x01z\n\x00\x00S\x00'
  3           0 RESUME                   0

  4           2 LOAD_FAST                0 (x)
              4 LOAD_CONST               1 (1)
              6 BINARY_OP               10 (-)
             10 RETURN_VALUE
foo(5) = 4

C:\Users\Eli\Desktop>
```

נראה שהפעם זה עובד, מגניב!

שדות נוספים באובייקט code

כאמור באובייקט code, מעבר לקוד ה-Bytecode של הפונקציה קיימים שדות נוספים. ביניהם נמצאים בין היתר השדה co_consts שמכיל את הערכים הקבועים שמשתמשים בהם בקוד, והשדה co_varnames שמכיל את שמות הפרמטרים לפונקציה והמשתנים שמוגדרים בה. שני שדות אלה הם מסוג tuple. פקודת Bytecode שמשתמשת באחד השדות האלה, מכילה oparg שערכו הוא האינדקס של האיבר המתאים ב-tuple. למשל בפונקציה הבאה וקוד ה-Bytecode המתאים לה:

```
Python 3.11 (64-bit)
>>> def foo(a, b):
...  return 2 * a + b
...
>>> dis.dis(foo)
  1           0 RESUME                   0

  2           2 LOAD_CONST               1 (2)
              4 LOAD_FAST                0 (a)
              6 BINARY_OP                5 (*)
             10 LOAD_FAST                1 (b)
             12 BINARY_OP                0 (+)
             16 RETURN_VALUE
>>>
```

ערכי השדות האלה הם:

```
Python 3.11 (64-bit)
>>> foo.__code__.co_consts
(None, 2)
>>> foo.__code__.co_varnames
('a', 'b')
>>>
```

ה-tuple של co_consts מכיל את הקבועים None (שתמיד נמצא בו) והערך 2. הפקודה LOAD_CONST, שדוחפת קבוע למחסנית, מכילה oparg שערכו 1 - זהו האינדקס של הקבוע 2 ב-co_consts.

ה-tuple של co_varnames מכיל את השמות a ו-b, שמות הפרמטרים שהפונקציה מקבלת. אם בפונקציה היו מוגדרים משתנים מקומיים, גם השמות שלהם היו נמצאים ב-tuple הזה. הפקודה LOAD_FAST הראשונה, שטוענת משתנה ודוחפת אותו למחסנית, מכילה oparg שערכו 0 - זהו האינדקס של השם a ב-tuple של co_varnames.

כבר ראינו קודם שה-oparg של פקודה נמצא בסמוך ל-opcode של הפקודה עצמה במערך הבתים של ה-Bytecode. אנחנו יכולים לשנות אותו כך שהפקודה תתייחס למשתנה אחר או קבוע אחר. למשל אם נחליף את ה-oparg בפקודה "LOAD_FAST 0" (שמתאים למשתנה a) בערך 1 (שמתאים למשתנה b), הדבר יהיה שקול לכך שהפונקציה תחזיר את הביטוי "2 * b + a".

הקוד לכך הוא:

```
import dis

def foo(a, b):
    return 2 * a + b

def find_offset_to_adaptive_bytecode(code_object):
    # copy entire code object into bytes variable
    size = 0x100
    code_object_bytes = b"\x00" * size
    memmove(code_object_bytes, id(code_object), size)
    # search _co_code_adaptive index in the bytes variable
    return code_object_bytes.index(code_object._co_code_adaptive)

print("before: foo(1, 2) =", foo(1, 2))

old_instruction = dis.opmap["LOAD_FAST"].to_bytes() + b"\x00" # index of a
new_instruction = dis.opmap["LOAD_FAST"].to_bytes() + b"\x01" # index of b
new_bytecode = foo.__code__._co_code_adaptive.replace(old_instruction, new_instruction)
from ctypes import memmove
offset = find_offset_to_adaptive_bytecode(foo.__code__)
memmove(id(foo.__code__) + offset, new_bytecode, len(new_bytecode))

print("after:  foo(1, 2) =", foo(1, 2))
```

והתוצאה:

```
Command Prompt
C:\Users\Eli\Desktop>python code_object.py
before: foo(1, 2) = 4
after:  foo(1, 2) = 6
```

אם נוציא את כל לוגיקת החלפת ה-Bytecode לפונקציה, שגם תבצע זאת בהתאם לגרסאת ה-Python הנוכחית, הקוד יראה יפה יותר:

```
import dis
def patch_function(func, old_instruction, new_instruction):
    import sys
    from ctypes import memmove
    assert len(old_instruction) == len(new_instruction)
    code = func.__code__

    if sys.version_info >= (3, 11):
        new_bytecode = code._co_code_adaptive.replace(old_instruction, new_instruction)
        # find offset to _co_code_adaptive
        size = 0x100
        code_object_bytes = b"\x00" * size
        memmove(code_object_bytes, id(code), size)
        co_code_adaptive_offset = code_object_bytes.index(code._co_code_adaptive)

        memmove(id(code) + co_code_adaptive_offset, new_bytecode, len(new_bytecode))
    if sys.version_info >= (3, 0):
        new_bytecode = code.co_code.replace(old_instruction, new_instruction)
        memmove(code.co_code, new_bytecode, len(new_bytecode))
    else:
        raise Exception("unsupported python version for patching function (for now...)")

def foo(a, b):
    return 2 * a + b

old_instruction = dis.opmap["LOAD_FAST"].to_bytes(1, byteorder="little") + b"\x00" # index of a
new_instruction = dis.opmap["LOAD_FAST"].to_bytes(1, byteorder="little") + b"\x01" # index of b

print("before: foo(1, 2) =", foo(1, 2))
patch_function(foo, old_instruction, new_instruction)
print("after:  foo(1, 2) =", foo(1, 2))
```

וגם ירוץ על שתי גרסאות ה-Python שמעניינות אותנו - 3.7 וגם 3.11:

```
Command Prompt
C:\Users\Eli\Desktop>C:\Python\Python37\python.exe code_object.py
before: foo(1, 2) = 4
after:  foo(1, 2) = 6

C:\Users\Eli\Desktop>C:\Python\Python311\python.exe code_object.py
before: foo(1, 2) = 4
after:  foo(1, 2) = 6
```

ניתן גם להחליף את הפקודה עצמה, למשל אם נרצה לשנות את "LOAD_CONST 1" ל-"LOAD_FAST 0", זה יהיה שקול לביטוי "return a *a + b". הקוד המתאים הוא:

```
old_instruction = dis.opmap["LOAD_CONST"].to_bytes(1, byteorder="little") + b"\x01" #
constant 2
new_instruction = dis.opmap["LOAD_FAST"].to_bytes(1, byteorder="little") + b"\x00" #
variable a

print("before: foo(1, 2) =", foo(1, 2))
patch_function(foo, old_instruction, new_instruction)
print("after:  foo(1, 2) =", foo(1, 2))
```

והתוצאה שלו:

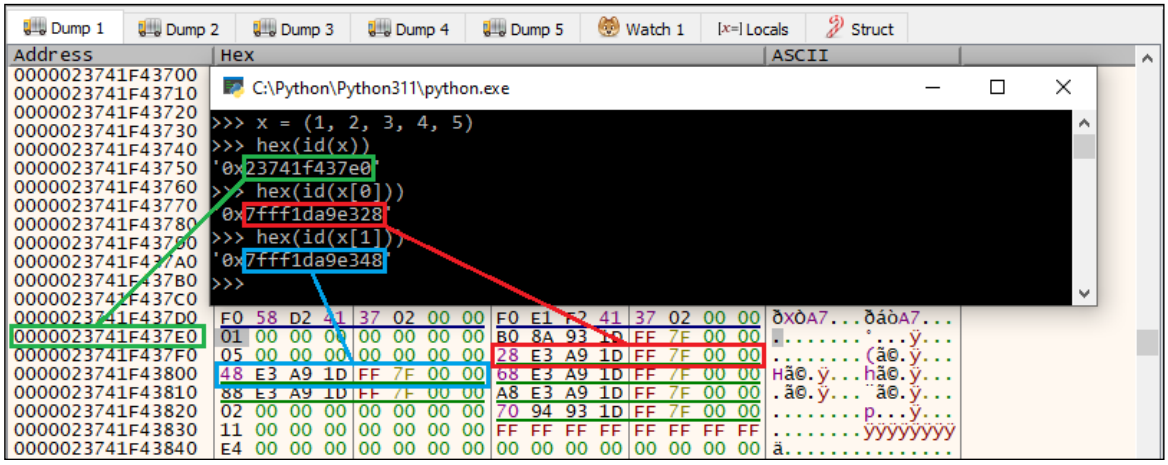
```
Command Prompt
C:\Users\Eli\Desktop>C:\Python\Python311\python.exe code_object.py
before: foo(1, 2) = 4
after:  foo(1, 2) = 3
```

שינויים ל-tuple-ים באובייקט code

עד כה ערכנו את שדה ה-Bytecode עצמו, אך מה אם היינו רוצים לערוך את שדות ה-co_consts וה-co_varnames? נניח שבדוגמא שלנו היינו רוצים להשתמש בקבוע 3 במקום בקבוע 2. בשביל לעשות את זה צריך לשנות את ה-tuple של co_consts. כידוע tuple הוא אובייקט immutable ולא ניתן לשינוי, וכשהוא באובייקט ה-code הוא גם לא ניתן להחלפה. אני מניח שאתם מבינים כבר לאן אני חותר.

כמו שעשינו קודם, ניתן לכתוב לזיכרון ישירות ולערוך את אובייקט ה-tuple כרצוננו. ראינו במאמר הקודם איך להסתכל על אובייקטים בזיכרון. נוכל לגשת ל-tuple באינדקס המתאים ולדרוס את המצביע לקבוע שנמצא שם במצביע לקבוע אחר. יש לשים לב שמכיוון שמדובר בדריסה של מצביעים, עקרונית יש צורך לעדכן את שדה ה-ob_refcnt בהתאם. אך מכיוון שלצורך הדוגמא נעבוד עם מספרים שהם מסוג immortal, אין צורך להתייחס לזה.

בבדיקה קצרה נגלה ש-tuple מיוצג כמערך של מצביעים לאובייקטים שנמצאים בו, ותחילתו ב-offset של 0x18 בתים מתחילת אובייקט ה-tuple:



אז די פשוט לכתוב קוד שדורס את המצביע שאנחנו רוצים. הקוד שמחליף את הקבוע 2 בקבוע 3 בדוגמא שלנו הוא:

```
def patch_function_constant(func, old_value, new_value):
    from ctypes import memmove
    consts = index = func.__code__.co_consts
    index = consts.index(old_value)
    memmove(id(consts) + 0x18 + index * 8, id(new_value).to_bytes(8, "little"), 8)

def foo(a, b):
    return 2 * a + b

print("before: foo(1, 2) =", foo(1, 2))
patch_function_constant(foo, 2, 3)
print("after:  foo(1, 2) =", foo(1, 2))
```

קוד זה עובד גם בגרסא 3.7 וגם בגרסא 3.11.

הוא גורם לפונקציה להיות שקולה לביטוי "return 3 * a + b". כשמריצים אותו מקבלים כצפוי:

```
C:\Users\Eli\Desktop>python patch_const.py
before: foo(1, 2) = 4
after:  foo(1, 2) = 5
```

ניתן באותו אופן לערוך את ה-tuple של co_varnames, אם כי במקרה זה כן יש צורך להתייחס לשדה ה-ob_refcnt, כפי שהוצג במאמר הקודם.

אלה היו מספר דוגמאות שמראות הוכחת היתכנות של שינוי קוד בצורה דינמית בזמן ריצה. כמובן שניתן להכליל את כל מה שעשינו עד כה וליצור שינויים מורכבים הרבה יותר, כך שיהיה קשה מאוד לרברס ולהבין מה הקוד שלנו עושה מהסתכלות על ה-Bytecode שלו בלבד.

כל קטעי הקוד שהצגתי במאמר זה נמצאים גם ב-fork שעשיתי לפרויקט CPython הרשמי בקישור.

סיכום

בחלק הראשון של המאמר הסתכלנו על מימושים של פעולות על האובייקטים של set ו-dictionary במטרה למצוא בהם באגים, וכתבנו קטעי קוד שמנצלים אותם. על הדרך למדנו על האופטימיזציה בשיטת ההשוואה בין אובייקטים. מאוד נחמד לראות בעיניים שבסופו של דבר כל Python זה פשוט קוד C, ממומש בפועל עם פוינטרים לאובייקטים ולפונקציות, ממש כמו שאנחנו כבר מכירים. כמו שבתוכניות שכתובות ב-C יש באגים, גם ב-Python בסופו של דבר ניתן למצוא ולנצל אותם באותו אופן.

דרך מציאת הבאגים שפעלתי לפיה כללה מעבר מהיר על הקוד וחיפוש קטעי קוד בעייתיים פוטנציאלית, כתיבת קוד Python שמגיע ל-flow הבעייתי, דיבוג ובחינת ההתנהגות. לדעתי זו יכולה להיות נקודת פתיחה טובה למציאת באגים יותר רציניים מאלו שהצגתי, ואולי אפילו חולשות. אני משוכנע שאם נסתכל מספיק, נוכל למצוא עוד התנהגויות בעייתיות ב-CPython.

בחלק השני של המאמר התמקדנו באובייקט code ואיך ניתן לשנות אותו כך שהפונקציונליות של הקוד תשתנה כרצוננו בזמן ריצה, וכל זאת למרות ההגבלה על כתיבה לאובייקט זה. יכולנו, כמו במאמרים הקודמים, לשנות את התכונות של האובייקט ולגרום לו להיות ניתן לכתיבה, לקמפל את CPython, וליצור גרסאת Python משלנו שבה יהיה אפשר ליצור את הקוד שמשנה את עצמו מבלי לדרוס את הזיכרון בדרך שעשינו. אבל אז מה שאנחנו מנסים לעשות יעבוד רק על גרסאת ה-Python שיצרנו ולא על גרסאות רשמיות, שזה פחות מגניב בעיניי.

אני אהבתי לראות איך אפשר להביא לידי ביטוי את הידע שצברנו במאמר הקודם, על איך האובייקטים ב-CPython עובדים ונראים בזיכרון, ולעשות איתו דברים מעניינים בפועל. השתמשנו בידע הזה כדי למצוא בדיוק באיזה מקום בזיכרון צריך לדרוס בתים כך שנקבל את הפונקציונליות שאנחנו רוצים. על הדרך גם למדנו על אופטימיזציית ה-Specialization של פקודות אדפטיביות בזמן ריצה ברמת ה-Bytecode.

בסופו של דבר הצלחנו לגרום לפונקציות להשתנות בזמן ריצה, למרות שהכותבים של Python התאמצו למנוע מאיתנו לעשות זאת.

ואת כל הדברים היפים האלה עשינו על גרסאות רשמיות של Python!

על המחבר

אני אלי קסקי, בן 30, אוהב לכתוב ולפתור אתגרי CTF, במיוחד בקטגוריות Reverse Engineering וקריפטוגרפיה, ואוהב לעשות דברים מגניבים עם קוד.

לפניות בכל נושא מוזמנים ליצור איתי קשר ב-elikaski94@gmail.com או ב-LinkedIn.