digital whisper
Technologic Papers

The Superpowers of Reverse Engineering

מאת xorpd

הקדמה

לפני שנים רבות כתבתי תוכנית קטנה (ב-Visual Basic 6) שסורקת ומורידה מידע מאתרי אינטרנט. למי שלא זכה להשתמש ב-Visual Basic, מדובר בסביבת פיתוח ל-Windows שמאפשרת לתכנת בקלות תוכניות עם ממשק משתמש גרפי. כשנה לאחר מכן עברתי למחשב חדש, אבל שכחתי לגבות את הקוד של התוכנה שכתבתי. כשרציתי לבצע כמה שינויים קטנים בתוכנית שלי גיליתי לצערי שכל מה שנותר לי מהתוכנה הוא קובץ הרצה מקומפל (קובץ EXE).

המקרה הזה הוביל אותי לשאלה, האם בהינתן קובץ הרצה מקומפל יש דרך לחזור חזרה אל הקוד המקורי? או אולי אם נדרוש קצת פחות, האם בהינתן קובץ הרצה מקומפל אפשר לגלות מה יש בתוכו וכיצד הוא עובד?

ביליתי הרבה זמן בלחפש את התשובה לשאלה הזו באינטרנט. קיוויתי שהתשובה לשאלה תהיה בדמות כלי פשוט שאפשר להוריד מהאינטרנט, שיחזיר לי את הקוד האבוד בצורה אוטומטית.

שאלתי אנשים בפורומים באינטרנט (זה מה שהיינו עושים לפני שגוגל היה פופולארי), אך בהתחלה התשובות היו עמומות. היו שאמרו לי שלשחזר את הקוד האבוד זוהי משימה בלתי אפשרית, מאחר והרבה מידע נעלם בתהליך הקומפילציה. לדוגמה: ההערות, שמות המשתנים והפונקציות בקוד לרוב נמחקים ע"י הקומפיילר ולא נמצאים בקובץ ההרצה הסופי.

לא הייתי מרוצה מהתשובה הזו, אז המשכתי לחפש. גיליתי שיש כלים שמאפשרים לפתוח קבצי הרצה ולראות את התוכן שלהם. לדוגמה, Hex Editor היא תוכנה המאפשרת לראות תוכן של כל קובץ במחשב, בבתים (בית הוא ערך בגודל 8 ביטים, והוא יחידת האחסון הבסיסית במחשב).

תוך שימוש ב-Hex Editor אפשר לראות לדוגמה את המחרוזות שמוטבעות בתוך קובץ הרצה, ואפילו לשנות אותן. אבל פרט למחרוזות, היו עוד הרבה מאוד בתים בתוך קובץ ההרצה של התוכנית שכתבתי שלצערי עוד לא יכולתי להבין.

```
00040e28 A8 67 44 74 2D C5 67 60 .gDt-.g`
00040e30 D7 D8 37 6D 59 3F 92 B9 ..7mY?..
00040e38 E0 D6 3C 14 27 55 07 E3 ..<.'U..
00040e40 05 31 6F 1E EC 85 78 A0 .1o...x.
00040e48 D6 98 11 26 35 09 7C FC ...&5.|.
00040e50 AD 71 D9 40 11 EC 47 94 .q.@..G.
00040e58 4B DA D3 89 FC A3 E2 2E K.......
00040e60 D8 05 E4 50 81 AA 0F F5 ...P....
00040e68 06 AE 46 CF 33 EA 81 E5 ..F.3...
00040e70 62 B9 4A 40 69 D4 78 27 b.J@i.x'
00040e78 B1 A7 7A DB 70 C9 38 F8 ..z.p.8.
00040e80 86 A9 03 69 BF 4F AC 82 ...i.O..
00040e88 5C 3A 4A EC 62 92 C7 51 \:J.b..Q
00040e90 C9 D3 D8 25 BE 10 3A D3 ...%..:.
00040e98 93 2B EF 83 2E 0B 18 12 .+......
00040ea0 92 96 86 C3 2B 0D F2 68 ....+..h
00040ea8 DF D2 17 14 7A 26 0E ED ....z&..
00040eb0 43 24 6F B5 56 CF 5D 55 C$o.V.]U
00040eb8 A4 38 01 22 4B A7 FB 20 .8."K..
00040ec0 DD C1 C0 E4 53 80 59 1A ....S.Y.
00040ec8 D0 2E 45 7B 82 BF 9C 99 ..E{....
00040ed0 38 64 95 06 A1 F4 7C 56 8d....|V
00040ed8 EB 2E E4 6E CB BB D6 0A ...n....
00040ee0 7F 78 E2 56 F3 8F F3 AA .x.V....
00040ee8 8B 13 97 C4 14 FF 89 C1 ........
```

[תמונה של hex editor]

אחרי מחקר נוסף התברר לי אודות התפקידים של שאר הבתים בקובץ. חלק מהבתים מספרים על המבנה של קובץ ההרצה למערכת ההפעלה. לדוגמה, קבצי EXE ב-Windows מכילים מבנה שנקרא PE, שמאפשר למערכת ההפעלה לדעת איך לטעון את קובץ ההרצה לזכרון כשהמשתמש מפעיל אותו. חלק אחר מהבתים מייצג קוד אסמבלי: זהו קוד מאוד בסיסי שהמעבד יכול להבין.

יש כלי שנקרא Disassembler שמאפשר לתרגם את הערך המספרי של פקודות האסמבלי לטקסט שקל לבני אדם לקרוא. לדוגמה, במעבד ממשפחת x64, אוסף הבתים:

```
48 83 C0 03 50
```

מייצגים את האסמבלי:

```
48 83 C0 03          add  rax, 3
50                   push rax
```

ובעברית: הוסף לאוגר rax את המספר 3, ולאחר מכן דחוף את הערך של האוגר rax לתוך המחסנית.

למעשה, כאשר כתבתי את התוכנית שלי ב-Visual Basic ולחצתי על לחצן הקומפילציה, הקומפיילר הפך את כל הקוד שכתבתי ב-Visual Basic להרבה מאוד פקודות אסמבלי, ולבסוף ארז את כל קוד האסמבלי שנוצר לתוך קובץ הרצה.

לכן כדי להבין איך קובץ הרצה עובד, אפשר להכניס אותו לתוך Disassembler ולהסתכל על פקודות האסמבלי שמתקבלות. עם קצת ידע על פקודות אסמבלי, עקשנות וזמן פנוי, אפשר להצליח להבין את הקוד שנמצא בתוך קובץ הרצה. לתהליך הזה קוראים הנדסה לאחור של קובץ ההרצה.

```
                    sub_4007A8      proc near               ; CODE XREF
41 54                               push    r12
41 55                               push    r13
41 55                               push    r13
49 89 FC                            mov     r12, rdi
49 89 F5                            mov     r13, rsi

                    loc_4007B4:                             ; CODE XREF
48 BF C0 10 60 00+                  mov     rdi, offset a02x ; "%02X"
49 0F B6 34 24                      movzx   rsi, byte ptr [r12]
31 C0                               xor     eax, eax
E8 C6 FC FF FF                      call    _printf
49 FF C4                            inc     r12
49 FF CD                            dec     r13
75 E2                               jnz     short loc_4007B4
41 5D                               pop     r13
41 5D                               pop     r13
41 5C                               pop     r12
C3                                  retn
                    sub_4007A8      endp
```

[תמונה של Disassembler מודרני (IDA)]

הנדסה לאחור היא יכולת מאוד שימושית. מתכנתים לעיתים משתמשים בהנדסה לאחור על מנת למצוא באגים קשים בתוכנות שלהם. חוקרי אבטחה משתמשים בהנדסה לאחור על מנת להבין איך מערכות פועלות. לדוגמה: חוקרי וירוסים של מחשבים מבצעים הנדסה לאחור של וירוסים כדי לגלות כיצד וירוס עובד, ואיך ניתן לנטרל אותו.

ReversingHero

איך כדאי ללמוד הנדסה לאחור? לדעתי הדרך הטובה ביותר היא על ידי עשיה. מוצאים קובץ הרצה קטן ומעניין, פותחים אותו ב-Disassembler ומתחילים לנסות להבין איך הוא עובד. הזמנים קצת השתנו מאז שהתחלתי ללמוד הנדסה לאחור. עדיין אפשר למצוא הרבה קבצי הרצה מעניינים, אבל רובם כבר לא קטנים.

כדי לאפשר גם למתחילים לקבל חוויה טובה של הנדסה לאחור בניתי ערכה ללימוד עצמי שנקראת ReversingHero. זהו קובץ הרצה שכולל בתוכו 15 שלבים של אתגרי הנדסה לאחור, אפשר להוריד את הקובץ מהאתר https://www.reversinghero.com. הקובץ נועד לרוץ על מחשבי לינוקס עם מעבד ממשפחת x64.

מה כדאי לדעת לפני שמתחילים לפתור את ReversingHero? דרוש ידע בסיסי בתכנות ב-C ואסמבלי. בנוסף, לשלבים המתקדמים יותר מומלץ לדעת להשתמש בשפת סקריפט כמו פייתון.

במאמר הזה נספר כיצד לפתור את שני השלבים הראשונים של ReversingHero. מהלך הפתרון כולו קורה בסביבת לינוקס. במהלך הפתרון אשתמש ב-Disassembler שנקרא Interactive Disassembler, או בקצרה IDA של חברת hex rays. ניתן להוריד גרסה חינמית של IDA באתר הרשמי:

https://www.hex-rays.com/products/ida/support/download_freeware.shtml

שלב ראשון של ReversingHero

נתחיל בלהוריד את קובץ ההרצה:

```
$ wget https://www.reversinghero.com/reversinghero
```

(במקום להשתמש בשורת הפקודה ניתן להוריד את הקובץ ישירות מהקישור באתר של ReversingHero).

מתקבל קובץ שנקרא reversinghero:

```
$ ls -la reversinghero
-rw-r--r-- 1 real real 595896 Oct 30 07:58 reversinghero
```

נסתכל על תוכן הקובץ בבתים באמצעות הכלי xxd:

```
$ xxd reversinghero | less

00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 3e00 0100 0000 7005 4000 0000 0000  ..>.....p.@.....
00000020: 4000 0000 0000 0000 f812 0900 0000 0000  @...............
00000030: 0000 0000 4000 3800 0600 4000 1300 1200  ....@.8...@.....
00000040: 0600 0000 0400 0000 4000 0000 0000 0000  ........@.......
00000050: 4000 4000 0000 0000 4000 4000 0000 0000  @.@.....@.@.....
...
```

על פי הבתים הראשונים בקובץ אפשר לזהות שמדובר בקובץ ELF. זהו קובץ הרצה לינוקס-י. ניתן לקובץ הרשאות הרצה:

```
$ chmod a+x reversinghero
$ ls -la reversinghero
-rwxr-xr-x 1 real real 595896 Oct 30 07:58 reversinghero
```

ונריץ את הקובץ:

```
$ ./reversinghero
@ 1/p1
@ 1/x1
] +
```

נראה שנוצרה תיקיה חדשה בשם "1", ובתוכה נוצרו שני קבצים חדשים p1 ו-x1.

```
$ cd
$ ls -la
total 572
drwx------ 2 real real   4096 Nov 21 17:17 .
```

```
drwxr-xr-x 3 real real   4096 Nov 21 17:17 ..
-rw-r--r-- 1 real real 566160 Nov 21 17:17 p1
-rw-r--r-- 1 real real   6640 Nov 21 17:17 x1
```

הקובץ p1 הוא די גדול (בערך 560KB), והקובץ x1 הוא קטן יחסית (בערך 6KB). מבדיקה נראה ששני הקבצים הם קבצי הרצה לינוקסיים מסוג ELF. הקובץ x1 הוא השלב הראשון של ReversingHero, והקובץ p1 הוא השער ("portal") אל השלבים הבאים. ניתן לשני הקבצים הרשאות הרצה באמצעות הפקודה:

```
chmod a+x ./*
```

נתחיל מהקובץ הקטן, x1. כאשר מריצים את x1 נראה שהוא ממתין לקלט. ננסה קלט כלשהו ונראה מה קורה:

```
$ ./x1
? asdfasdf
! -
```

הקלט שהכנסנו בדוגמה הוא "asdfasdf", נראה שקיבלנו בחזרה את הפלט "-" והתוכנית הסתיימה.

ננסה לראות אם יש מחרוזות מיוחדות בתוך הקובץ x1 באמצעות הכלי strings. זהו כלי שבדרך כלל מגיע עם רוב ההפצות של לינוקס, ומאפשר למצוא רצפים של תווים דפיסים בתוך קבצים:

```
$ strings x1
/lib64/ld-linux-x86-64.so.2
libc.so.6
printf
[...]
A^A]A\
&EI1
ATAUAUI
A]A]A\
ATAUAUH
A]A]A\
d(-_-)b//d(+_+)b\\d(-_-)b
! +
ReversingHero
www.xorpd.net
%02X
_DYNAMIC
_GLOBAL_OFFSET_TABLE_
_edata
[...]
```

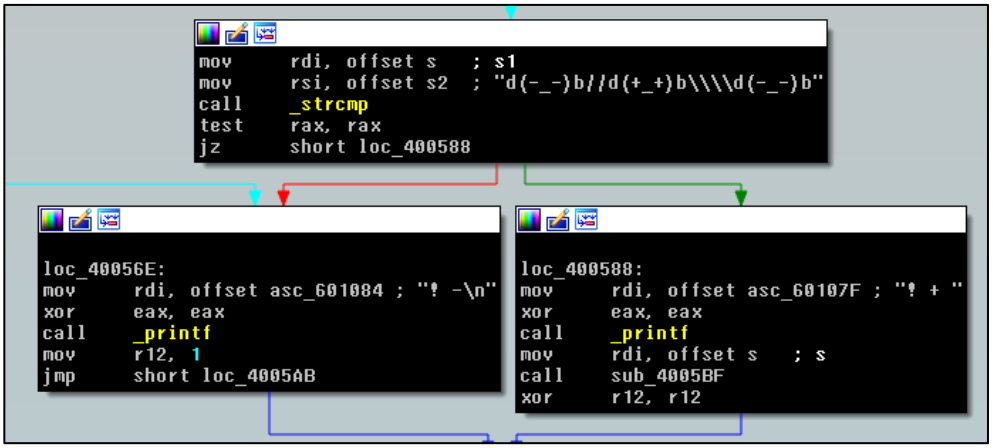
חלק מהפלט של strings הושמט. ממבט במחרוזות אפשר לזהות כמה מחרוזות מעניינות. אחת מהן היא המחרוזת:

```
d(-_-)b//d(+_+)b\\d(-_-)b
```

שנראית כמו ציור של מספר אנשים מאזינים למוזיקה. אולי המחרוזת הזו היא הפתרון? ננסה להכניס אותה בתור קלט לקובץ x1:

```
$ ./x1
? d(-_-)b//d(+_+)b\\d(-_-)b
! +
3E437BBA43971D612049DE8AD54FDEF068931E8C6D26F63D83742F932E740B6D
```

ואכן נראה שקיבלנו פלט שונה מהפלט שהתקבל קודם! זה כנראה סימן טוב. קיבלנו "+", ואחריו מחרוזת ארוכה של תווים בבסיס 16. הסתמכנו כאן על ניחוש טוב והרבה מזל. מה אם לא היינו חושבים על לבדוק מחרוזות? נפתח את x1 ב-IDA ונתבונן בקוד:



ב-IDA אפשר לראות קריאה לפונקציה strcmp. זוהי פונקציה שמשווה בין שתי מחרוזות. אפשר לראות שהארגומנט השני של strcmp (נכנס באמצעות האוגר rsi) היא המחרוזת "d(-_-)b//d(+_+)b\\d(-_-)b". סביר להניח שהארגומנט הראשון (נכנס באמצעות האוגר rdi) קשור בקלט שלנו.

בנוסף, אפשר לראות שבלוק ההשוואה מתפצל לשתי אפשרויות:

- צד שמאל (החץ האדום) מייצג את המקרה שבו אין שוויון בין המחרוזות, זהו המקרה שמייצג כשלון, ואפשר לראות שאכן מודפס התו "-" למסך במקרה זה, כמו שקרה לנו כשהכנסנו את הקלט "asdfasdf" בנסיון הראשון.
- צד ימין (החץ הירוק) מייצג את המקרה שבו יש שוויון בין המחרוזות. זה כנראה המקרה שמייצג הצלחה. במקרה זה מודפס התו "+" למסך, ואחריו יש קריאה לפונקציה נוספת, ככל הנראה הפונקציה שמדפיסה את המחרוזת של התווים בבסיס 16.

ניקח את המחרוזת של התווים בבסיס 16 שהתקבלה ונכניס אותה בתור קלט לקובץ p1, על מנת לעבור לשלב הבא:

```
$ ./p1
> 3E437BBA43971D612049DE8AD54FDEF068931E8C6D26F63D83742F932E740B6D
@ 2/p2
@ 2/x2
] +
```

נראה שנוצרה תיקיה חדשה בשם 2, ובתוכה יש שני קבצים: x2 ו-p2:

```
$ ls -la
total 540
drwx------ 2 real real   4096 Nov 21 17:52 .
drwxr-xr-x 4 real real   4096 Nov 21 17:53 ..
-rw-r--r-- 1 real real 536240 Nov 21 17:52 p2
-rw-r--r-- 1 real real   6648 Nov 21 17:52 x2
```

בדומה לשלב הראשון, כאן הקובץ x2 הוא האתגר של שלב 2, ו-p2 הוא השער למעבר לשלבים הבאים.

שלב שני של ReversingHero

נתחיל בלתת הרשאות הרצה ל-x2 ו-p2:

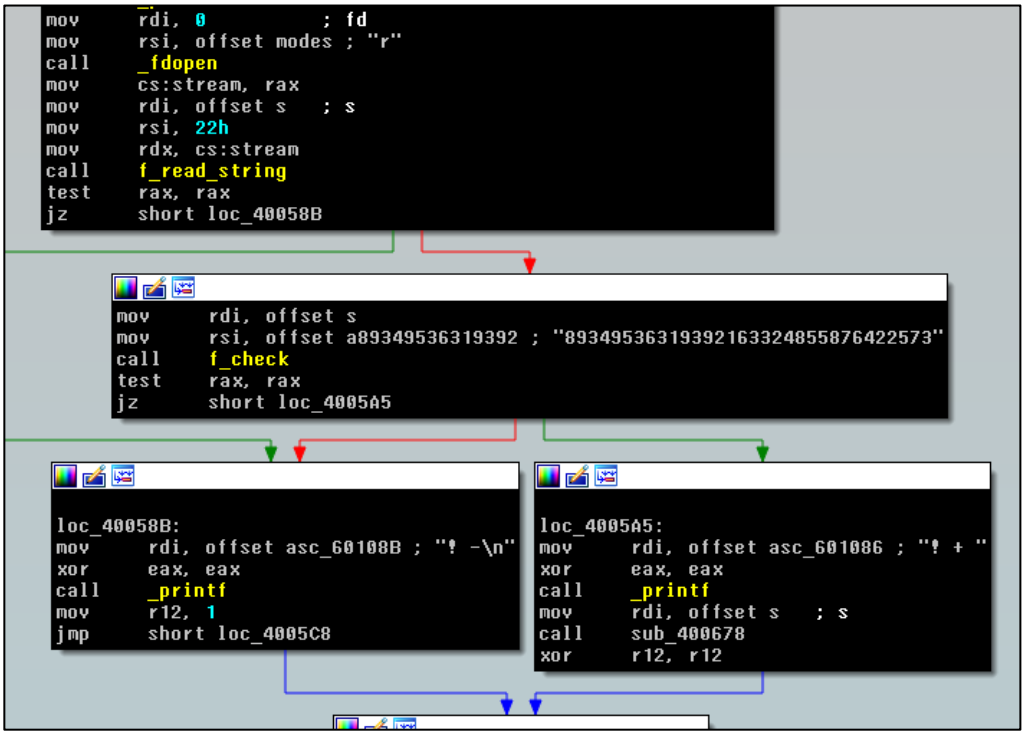
```
$ chmod a+x ./*
```

ונריץ את x2:

```
$ ./x2
? qwerqwer
! -

$ ./x2
? 12341234
! -
```

ניסינו להכניס שתי מחרוזות שונות ("qwerqwer" ו-"12341234"), נראה שאף אחד מהן לא עבדה: בשני המקרים קיבלנו כפלט את המחרוזת "-". נפתח את x2 ב-IDA וננסה להבין איך הוא עובד. כך נראית הפונקציה הראשית של x2 (נקראת start):



[בתמונה: כל הפונקציות ששמן מתחיל ב-"f_" הן פונקציות שאנחנו בחרנו להן שם]

הפונקציה מתחילה בקריאה לפונקציה f_read_string. זהו שם שבחרתי לפונקציה על סמך מספר סימנים:

- מיד לפני הפונקציה פותחים את stdin (צינור הקלט מהמשתמש) באמצעות הפונקציה fdopen לקריאה.
- נראה שהפונקציה מקבלת את הגלובלי s בתור ארגומנט, ולאחר מכן מבצעים איתו פעולות אחרות. בסיכוי טוב s קשור לקלט שלנו.

ככל הנראה הפונקציה f_read_string קוראת מחרוזת מהמשתמש, ושומרת אותו בתוך הגלובאלי s.

לאחר מכן קוראים לפונקציה f_check עם שני ארגומנטים: הערך s שנקרא מהמשתמש, ומחרוזת של ספרות:

```
89349536319392163324855876422573
```

נראה שהפונקציה f_check גם מקבלת את ההחלטה האם עברנו את השלב או לא: אפשר לראות שהבלוק שבו קוראים ל-f_check מסתיים באלטרנטיבה של שתי אפשרויות:

- אם האוגר rax מכיל ערך ששונה מ-0 נגיע שמאלה (חץ אדום) מוביל לכשלון. אפשר לראות שהבלוק השמאלי מדפיס את התו "-" למסך.
- אם האוגר rax מכיל את הערך 0 נגיע ימינה (חץ ירוק) מוביל להצלחה. הבלוק הימני מדפיס את התו "+" למסך.

זו הסיבה שבחרתי את השם f_check לפונקציה. אפשר להניח שהיא "בודקת" משהו על הקלט שלנו.

אם נצליח לגרום ל-f_check להחזיר 0 באוגר rax נוכל לפתור את השלב. לשם כך נבדוק איך הפונקציה f_check עובדת.

הפונקציה f_check

```
; rdi -- user_input
; rsi -- digits_str

f_check proc near
push    r12
mov     r12, rsi

; rdi -- user_input
mov     rsi, offset g_buffer
call    f_calc


mov     rsi, offset g_buffer ; s2
mov     rdi, r12             ; digits_str
mov     rdx, 20h             ; n
call    _memcmp
pop     r12
retn
f_check endp
```

[בתמונה: הפונקציה f_check]

נראה ש-f_check בנויה משתי קריאות: קריאה לפונקציה f_calc ולאחר מכן קריאה לפונקציה memcmp. עדיין לא ידוע לנו מה f_calc עושה, אז נתחיל בלהבין את החלק השני: הקריאה ל-memcmp.

הפונקציה memcmp היא פונקציה מפורסמת. נוכל למצוא הסבר עליה בדפי ה-man של לינוקס:

```
$ man memcmp
NAME
    memcmp - compare memory areas

SYNOPSIS
    #include <string.h>

    int memcmp(const void *s1, const void *s2, size_t n);

DESCRIPTION
    The memcmp() function compares the first n bytes (each interpreted
as unsigned char) of the memory areas s1 and s2.
```

בעברית: memcmp היא פונקציה שמשווה בין חלקי זכרון. היא מקבלת שני מצביעים לזכרון (s1 ו-s2) ומשווה אותם לאורך n בתים. במידה ויש שיוויון, memcmp תחזיר rax=0. אחרת, יוחזר ב-rax מספר שאינו 0.

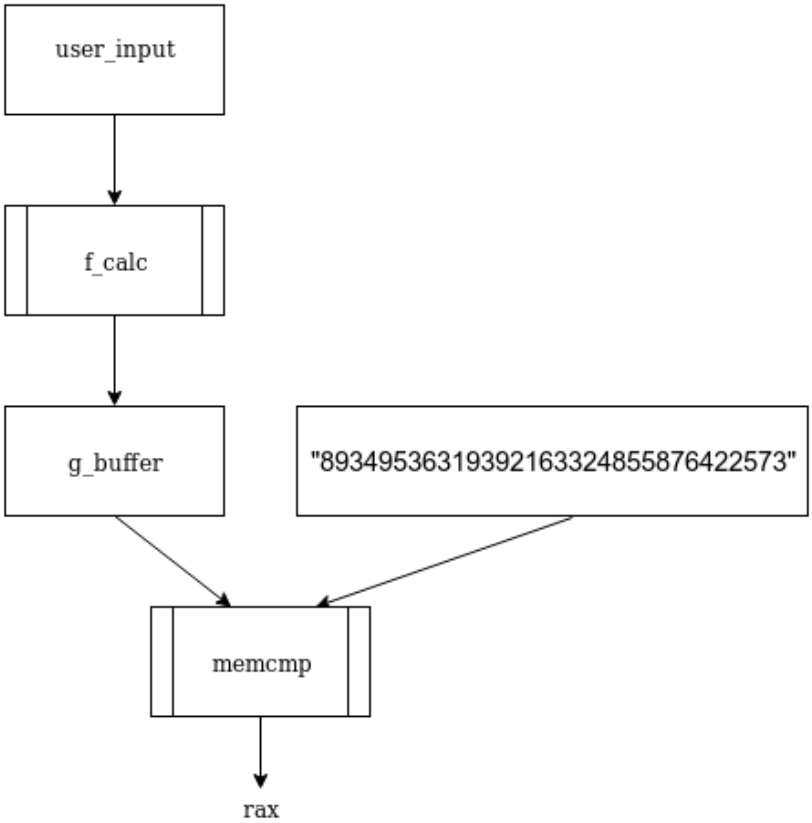
כדי לפתור את שלב 2, אנחנו נרצה לקבל rax=0, כדי שערך החזרה של הפונקציה f_check כולה יהיה rax=0. לכן אנחנו רוצים לוודא ששני חלקי הזכרון ש-memcmp משווה אכן יהיו שווים.

אז מה memcmp משווה? נראה שההשוואה נעשית בין שני buffer-ים באורך 0x20=32 בתים. ה-buffer הראשון הוא g_buffer, זהו buffer גלובלי כלשהו ששמור בזכרון. כרגע עוד לא ברור מה יש בתוכו. ה-buffer השני מוצבע ע"י האוגר r12. במקור זהו הארגומנט השני של הפונקציה f_check: מחרוזת הספרות הארוכה. בהערות קראנו למחרוזת הזו digits_str. כלומר - נראה ש-memcmp משוואה את g_buffer למחרוזת הספרות הארוכה. אבל מה יש בתוך g_buffer?

הקריאה הקודמת ל-f_calc מקבלת את g_buffer בתור הארגומנט השני (דרך האוגר rsi). הארגומנט הראשון מועבר דרך rdi, והוא מחרוזת הקלט שניתנה ע"י המשתמש. מעניין לשים לב שהארגומנט הראשון נכנס בצורה "שקטה" לתוך rdi: f_calc מגיע כל הדרך מלמעלה, עוד מהפונקציה הראשונה שראינו, start.

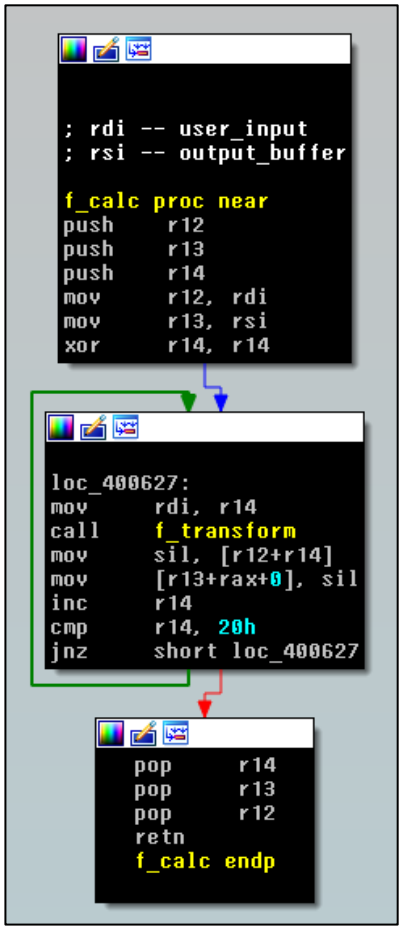
עד כה לא ראינו אף קטע קוד שמאתחל את g_buffer. לכן אפשר להניח שהפונקציה שבונה את התוכן של g_buffer היא הפונקציה f_calc. ככל הנראה f_calc מקבלת את הקלט של המשתמש, ובאמצעותו בונה את התוכן של g_buffer.

נתאר את התהליך שמתבצע בפונקציה f_check בדיאגרמה:



נותר להבין איך הפונקציה f_calc עובדת. אם נצליח למצוא קלט כך שהפונקציה f_calc תוציא את מחרוזת התווים הארוכה כפלט, נצליח לפתור את השלב.

הפונקציה f_calc



הפונקציה f_calc מקבלת שני ארגומנטים: הקלט מהמשתמש באוגר rdi, ו-buffer לפלט באוגר rsi.

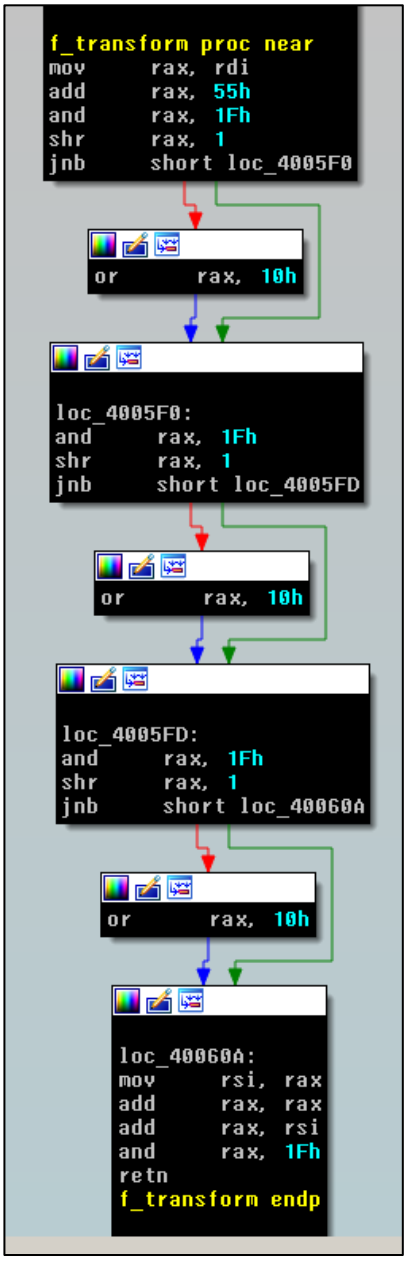
נראה שהאוגר r14 מאותחל ל-0 באמצעות הפקודה xor, ולאחר מכן מתחילה לולאה. האוגר r14 משמש כאיטרטור בלולאה: הוא מתחיל בערך 0, ובכל איטרציה של הלולאה r14 גדל ב-1 (באמצעות הפקודה inc). יציאה מהלולאה מתרחשת רק כאשר r14 שווה לערך 0x20 = 32.

בכל איטרציה של הלולאה קוראים לפונקציה f_transform (שם שאנחנו בחרנו לפונקציה) עם הארגומנט r14. התוצאה מוחזרת ב-rax. לאחר מכן קוראים בית אחד מתוך האינדקס r14 במחרוזת הקלט של המשתמש. את הבית שנקרא כותבים למקום rax ב-buffer הפלט. ככה זה אמור להראות בקוד C:

```
for (r14 =0; r14 <0x20; ++r14) {
    output_buffer[f_transform(r14)] = user_input[r14];
}
```

נראה שהפלט של הפונקציה f_calc הוא שינוי סדר הבתים במחרוזת הקלט מהמשתמש. בבירור לא יכולים להיווצר תווים חדשים שלא נמצאים במחרוזת הקלט מהמשתמש. בתור מסקנה, מחרוזת הקלט שלנו צריכה להכיל אך ורק ספרות. בנוסף, נראה שאנחנו צריכים לפחות 32 ספרות במחרוזת הקלט שלנו. נותר לבדוק איך הפונקציה f_transform עובדת.

הפונקציה f_transform



ידוע לנו ש-f_transform מקבלת מבחוץ את האיטרטור r14 (שהוא מספר בין 0 ל-0x20, לא כולל), ומחזירה מספר שאמור להיות אינדקס לתוך g_buffer, שהוא buffer באורך 0x20. כלומר, אנחנו מצפים מ-f_transform לקבל מספר בין 0 ל-0x20 (לא כולל), ולהחזיר תוצאה שהיא מספר בין 0 ל-0x20 (לא כולל).

נקרא לארגומנט של f_transform בשם x. הבלוק הראשון מתחיל בלחבר 0x55 ל-x (באמצעות הפקודה add).

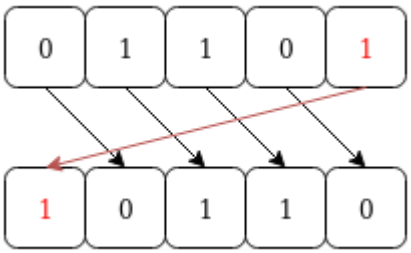
לאחר מכן מבצעים מספר פעולות לוגיות בביטים: תחילה מבצעים and עם המסכה 0x1f, ומזיזים ביט אחד ימינה (באמצעות הפקודה shr). כאשר מזיזים ביט אחד ימינה, מאבדים את הביט התחתון ביותר.

הפקודה הבאה, jnb, בודקת האם הביט שאבד היה 0 או 1.

- אם הביט התחתון היה 1, מבצעים or עם המסכה 0x10.
- אם הביט התחתון היא 0, מדלגים.

ננסה להבין את המשמעות מאחורי הפעולות הלוגיות בביטים. ראשית, נשים לב שכל הפעולות קורות בתוך מגרש בגודל 5 ביטים. זאת אפשר לדעת לפי המסכה 0x1f = 0b11111. הכמות של 5 ביטים גם מסתדרת עם גודל ה-buffer-ים בפונקציות החיצוניות: g_buffer, לדוגמה, הוא באורך 0x20 = 32 בתים. 5 ביטים בדיוק מספיקים לייצג אינדקס לתוך g_buffer, כי 2 בחזקת 5 שווה בדיוק ל-32.

נחזור לפעולה הלוגית של ההזזה ימינה בביט אחד: המשמעות של פעולת or עם המסכה 0x10 היא הדלקת הביט החמישי (הגבוה ביותר בעולם של 5 ביטים). אם הביט התחתון היה 1, מדליקים את הביט החמישי. אם כך, אפשר לדמיין את המהלך כולו בתור סיבוב של ביט אחד ימינה בעולם של 5 ביטים!



[בתמונה: סיבוב אחד ימינה בעולם של 5 ביטים]

נראה שאותו רצף פעולות: and, shr , jn, or קורה עוד פעמיים נוספות. כלומר, סה"כ מבצעים סיבוב של 3 ביטים ימינה (בעולם של 5 ביטים).

אפשר לסמן את הערך שהתקבל עד כה:

```
y = ((x + 0x55) & 0x1f) >>> 3
```

לאחר מכן, מבצעים את שלושת הפקודות:

```
mov rsi, rax   ; y
add rax, rax   ; 2*y
add rax, rsi   ; 2*y + y = 3*y
```

שומרים את הערך שהתקבל y לתוך האוגר rsi, מחברים את y לעצמו, ומוסיפים את rsi. התוצאה שנקבל בסך הכל היא 3*y (שלוש פעמים y). לבסוף מבצעים פעולת and עם המסכה 0x1f, כדי לוודא שהערך שמתקבל נשאר בין 0 ל-x200 (לא כולל).

כלומר, הפעולה הכוללת של f_transform היא:

```
y = ((x + 0x55) & 0x1f) >>> 3
return (3*y) & 0x1f
```

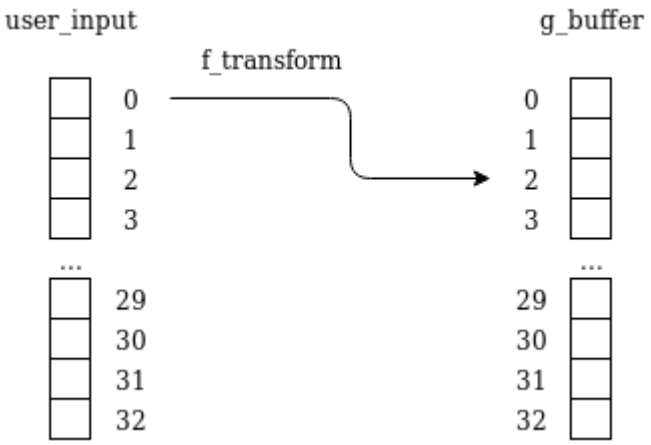
בניית הפתרון

נזכר שההשוואה האחרונה שחשובה לנו על מנת לפתור את שלב 2 היא ההשוואה האחרונה בתוך הפונקציה f_check: השוואה בין g_buffer לבין מחרוזת הספרות הארוכה המוטבעת בתוך קובץ ההרצה.

נראה שיש לנו שליטה מסויימת על הבניה של g_buffer: הבתים בו מועתקים מתוך הבתים של הקלט של המשתמש, רק בסדר קצת אחר. הסדר נקבע ע"י הפונקציה f_transform באופן הבא: לכל אינדקס i בין 0 ל-32 (לא כולל):

```
output_buffer[f_transform(i)] = g_buffer[i];
```

נותר לוודא האם f_transform באמת מאפשרת לנו להגיע לכל האינדקסים של output_buffer (אינדקסים מ-0 עד 20x0 = 32, לא כולל).



[בתמונה: דוגמה לאיך בתים מתוך user_input מסודרים מחדש בתור בתים ב-g_buffer]

הפונקציה f_transform פועלת על קלט מתחום מאוד קטן, לכן קל לייצר טבלה של כל האפשרויות לקלט ופלט. נעשה זאת באמצעות סקריפט פייתון:

```
def rotate_right(x, i):
    return (x >> i) | (x << (5-i)) & 0x1f

def f_transform(x):
    y = rotate_right(((x + 0x55) & 0x1f), 3)
    return (3*y) & 0x1f

print([f_transform(i) for i in range(0x20)])
```

הפונקציה הראשונה, rotate_right, מבצעת את הסיבוב הביטי ימינה בעולם של חמישה ביטים. הפונקציה f_transform היא סימולציה בפייתון של הפונקציה f_transform מהבינארי. לבסוף אנחנו מדפיסים את f_transform עבור כל i בתחום 0 עד 20x0, לא כולל.

הפלט שמתקבל הוא:

```
[2, 14, 26, 9, 21, 1, 13, 25, 5, 17, 29, 0, 12, 24, 4, 16, 28, 8, 20, 3,
15, 27, 7, 19, 31, 11, 23, 6, 18, 30, 10, 22]
```

מה המשמעות של הפלט?

- הבית במקום 0 של user_input יועתק לתוך בית מספר 2 של g_buffer.
- הבית במקום 1 של user_input יועתק לתוך בית מספר 14 של g_buffer.

וכך הלאה.

תופעה מעניינת שאפשר לראות בפלט היא שכל המספרים בין 0 ל-32 (לא כולל) מופיעים בדיוק פעם אחת - אף מספר לא חוזר פעמיים. במילים אחרות, f_transform היא פרמוטציה!

הידע שצברנו עד כה צריך להספיק לנו כדי להרכיב קלט שיפתור את שלב 2. נתבונן במחרוזת הספרות שמשווית ל-g_buffer:

```
89349536319392163324855876422573
```

- אנחנו רוצים ש-g_buffer במקום ה-0 יהיה הספרה '8'. לכן הבית במקום 11 של הקלט צריך להיות הספרה '8'.
- אנחנו רוצים ש-g_buffer במקום ה-1 יהיה הספרה '9'. לכן הבית במקום 5 של הקלט צריך להיות הספרה '9'.
- אנחנו רוצים ש-g_buffer במקום ה-2 יהיה הספרה '3'. לכן הבית במקום 0 של הקלט צריך להיות הספרה '3'.

וכן הלאה. כך נוכל להרכיב את הקלט שיפתור את השלב. נבצע את ההרכבה של הקלט באמצעות סקריפט פייתון:

```
DIGITS = '89349536319392163324855876422573'

user_input = [None] * 0x20
for i in range(0x20):
    user_input[i] = DIGITS[f_transform(i)]

print(''.join(user_input))
```

הסקריפט מרכיב את user_input כפי שתיארנו למעלה. הפלט שמתקבל:

```
31415926535897932384626433832795
```

ננסה להכניס את המחרוזת הזו בתור קלט לתוכנית x2:

```
$ ./x2
? 31415926535897932384626433832795
! + 8899660DBD0537F4B5148EECBB7481E6D6A6FF2C96FD97C0A62A52DA34497544
```

ואכן, נראה שהצלחנו וקיבלנו מפתח לשלב הבא!

סיכום

למדנו קצת על הנדסה לאחור: עם אילו בעיות הנדסה לאחור מאפשרת להתמודד, ומהם הכלים בהם משתמשים על מנת לבצע הנדסה לאחור. לאחר מכן פתרנו את שני השלבים הראשונים של ReversingHero, שהוא ערכה ללימוד עצמי של הנדסה לאחור.

עכשיו זה התור שלכם, נותרו עוד 13 שלבים ☺

לכל הערה, שאלה או רעיון, שלחו לי מייל לכתובת: xorpd@xorpd.net