מדריך GDB לקורס את"מ

תוכן עניינים

מה זה GDB?

תוכנת GDB (GNU Debugger) היא Debugger. התוכנה מבוססת על ארבעה דברים בסיסיים שהיא יודעת לעשות, ומספר רב של אפשרויות נוספות, המבוססות על ארבעת הדברים הבסיסיים. מהן אותן ארבעה דברים? בעת דיבוג של תוכנית, GDB נותן את האופציות הבאות[1]:

1. הרצת התוכנית.
2. עצירת התוכנית לפי תנאים מוגדרים מראש (breakpoint ,single-step וכו')
3. בדיקת השפעות התוכנית (תוכן הזיכרון, תוכן הרגיסטרים וכו') ברגע עצירתה (עצירה כמו בסעיף 2)
4. שינויים בתוכנית בזמן ריצה לצרכי דיבוג.

ב-GDB ניתן לדבג תוכניות במספר רב של שפות תכנות: Ada, Assembly, C, C++, D, Fortran, Go, Objective-C, OpenCL, Modula-2, Pascal וְ- Rust.

מה הקשר לקורס את"מ?

מבין כל שפות התכנות ש-GDB תומכת בהן, אנו פוגשים בקורס את C וכמובן את אסמבלי.
אנו נשתמש בתוכנה בעיקר לדיבוג של קוד אסמבלי (Fun Fact – גם הדיבאגר של תוכנת SASM לקוד ב-GAS מבוסס על GDB[2].
GDB מותקנת לכם על המכונות הווירטואליות שקיבלתם, השתמשו בה בחוכמה[3].
Fun Fact – GDB משתמשת בקריאת המערכת (מה זה קריאת מערכת? נלמד בקורס) מסוג ptrace (מה זו קריאת המערכת הזו? גם את זה נלמד בקורס).

---

[1] https://www.gnu.org/software/gdb
[2] https://dman95.github.io/SASM/english.html
[3] With great debugger comes great responsibility

איך להשתמש ב-GDB?

נתמקד במדריך הזה במספר אופציות בסיסיות לדיבוג קוד אסמבלי, שיעזרו לכם בקורס. למעוניינים, ניתן גם לקרוא על עוד אפשרויות שקיימות בדיבאגר במדריך המלא[4].

הפעלת הדיבאגר על תוכנית

על מנת להריץ קובץ ריצה (executable), עלינו להריץ בטרמינל את הפקודה הבאה[5]:

```
gdb <program>
```

כאשר program הוא ניתוב לקובץ הריצה שאנו מדבגים.

הרצת התוכנית

לאחר שהעברנו את הניתוב לתוכנית שלנו אל ה-GDB בעת הפעלתו, ניתן כעת להריץ אותה כמה פעמים שנרצה, ובכל ריצה להתמקד בנקודה אחרת שמעניינת אותנו.

ארגומנטים לריצה

לפני הרצת התוכנית, ניתן לקבוע עם אילו ארגומנטים היא תרוץ (ברירת המחדל היא ריצה ללא ארגומנטים כלל). על מנת ליישר קו - ארגומנטים של תוכנית הם command line arguments (אלו שנגישים לכם באמצעות argv בשפת c לדוגמה). קביעת הארגומנטים נעשית עם הפקודה הבאה[6]:

```
set args <args>
```

כאשר args הוא רצף הארגומנטים לתוכנית. ניתן להריץ את פקודה זו לפני כל הרצה מחדש של התוכנית בדיבאגר, כדי לבדוק ריצות עם ארגומנטים שונים. במידה ו-args ריק, הריצה הבאה תהיה ללא ארגומנטים כלל.

<u>הערה:</u> במקרים בהן לתוכנית אין שימוש ב-command line arguments, אין צורך להשתמש בפקודה זו כלל.

התחלת ריצה

כשנרצה להריץ את התוכנית, לאחר שקינפגנו ארגומנטים ושמנו breakpoints במידה ורצינו, נריץ את הפקודה[7]:

```
run
```

<u>הערה:</u> ניתן להריץ את התוכנית גם עם הקיצור לפקודה: r.
כאשר תוכנית התחילה לרוץ, היא אמורה להסתיים (בהצלחה, או שלא בהצלחה), אלא אם התערבנו באופן הריצה שלה (למשל, באמצעות הפקודות לעצירה שיוצגו בהמשך המדריך).

---

[4] https://sourceware.org/gdb/current/onlinedocs/gdb/, http://sourceware.org/gdb/current/onlinedocs/gdb.pdf
[5] https://sourceware.org/gdb/current/onlinedocs/gdb/Invoking-GDB.html#Invoking-GDB
[6] https://sourceware.org/gdb/current/onlinedocs/gdb/Arguments.html#Arguments
[7] https://sourceware.org/gdb/current/onlinedocs/gdb/Starting.html#Starting

עצירת הריצה

נדבר על שתי דרכים לבצע ריצה מבוקרת של קוד, מתוך מגוון האפשרויות ש-GDB מציע לנו[8].

הדרך הראשונה, היא להשתמש ב-breakpoint. כשמה כן היא, breakpoint היא נקודת עצירה. כשאנו בוחרים נקודה בתוכנית ושמים שם breakpoint, למעשה אנו רוצים שהתוכנית תעצור בנקודה זו. לרוב, העצירה תהיה במטרה לדבג.

הדרך השנייה, היא באמצעות stepping (נציג במדריך זה שימוש ב-single step). שימוש ב-stepping הוא הרצת הקוד "בדילוגים". כלומר ביצוע של מספר פעולות ולאחר מכן – עצירת הריצה.
כאמור, אנו נדגים כאן רק שימוש ב-single stepping, שאומר לדיבאגר לבצע פקודה אחת בלבד בתוכנית שאותה אנו מדבגים, ולאחריה לעצור.

בנוסף, נסביר איך להגיד לתוכנית להמשיך לרוץ (לאחר עצירה), מבלי להפסיק (אלא אם היא הגיעה ל-breakpoint).

Breakpoints

ישנן אפשרויות רבות לשימושים ב-breakpoints [9], רובן מתקדמות ולא ניגע בהן במדריך הנוכחי.
מה כן? נלמד לשים breakpoints בקוד[10]. זה נעשה באמצעות הפקודה הבאה:

```
break <label>
```

כאשר label (תזכורת: "כינוי" לכתובת מסוימת), הוא המקום בו אנו רוצים לעצור את הריצה.
<u>הערה:</u> ניתן לכתוב רק את הקיצור לפקודה: `b`.

Continuing and Stepping

בחלק זה, GDB נותן לנו מספר רב של אפשרויות להמשיך את הריצה לאחר עצירתה[11]. נתחיל עם הפקודה הבסיסית ביותר, שמאפשרת להמשיך להריץ את התוכנית עד לסיומה ללא עצירות, מלבד breakpoints או סיגנלים:

```
continue
```

<u>הערה:</u> ניתן לכתוב רק את הקיצור לפקודה: `c`.

כעת יש קטע מעט טריקי, אז חשוב לשים לב להבדל בין שתי הפקודות הבאות:

```
stepi <count>
step <count>
```

בשתי הפקודות ניתן לא להוסיף את count ונקבל את count הדיפולטי, שהוא 1.
<u>הערה:</u> לפקודת stepi ניתן לכתוב רק את הקיצור: `si`.
אז מה ההבדל ביניהן? נתחיל עם stepi, שמשמעותה step instruction – אנו מבקשים מהדיבאגר לבצע count פקודות אסמבלי בתוכנית המדובגת ואז לעצור שוב.
לעומתה, פקודת step מבצעת count <u>שורות קוד</u> ומיועדת לדיבוג ריצה של <u>קוד בשפה עילית.</u> אם תרצו לדבג קוד c, עליכם לקמפל אותו עם הדגל g- או ggdb- ואז תוכלו להשתמש בפקודת step כהלכה. כך גם עובדים דיבאגרים שפגשתם (או תפגשו) לשפות עיליות ב-IDE למיניהם.

---

[8] https://sourceware.org/gdb/current/onlinedocs/gdb/Stopping.html#Stopping
[9] https://sourceware.org/gdb/current/onlinedocs/gdb/Breakpoints.html#Breakpoints
[10] https://sourceware.org/gdb/current/onlinedocs/gdb/Set-Breaks.html#Set-Breaks
[11] https://sourceware.org/gdb/current/onlinedocs/gdb/Continuing-and-Stepping.html#Continuing-and-Stepping

שימו לב שלא להשתמש בפקודה זו אם אתם מדבגים קוד אסמבלי. אין לכם "שורות קוד" וה-gdb ימשיך את הריצה ללא עצירות.

ישנן עוד צמד פקודות שכדאי להכיר וההבדל ביניהן הוא כמו ההבדל בין step ו-stepi, ואלו הן:

```
nexti <count>
next <count>
```

וגם הן מבצעות פקודה/שורת קוד אחת (או יותר, לפי count) ועוצרים – אך במידה והפקודה היא קריאה לפונקציה, מתבצעת כל הפונקציה לפני העצירה.

למי שיש היכרות עם דיבאגרים אחרים, יכול להקביל במדויק את step ל-"step into" ואת next ל-"step over". יש גם פקודה דומה ל-"step out" והיא פקודת finish) ועוד מספר פקודות, כמו until, שאתם יכולים לקרוא עליהן בתיעוד של gdb.

בדיקת תוכן זיכרון ורגיסטרים

ישנן מגוון רחב של אפשרויות ש-GDB מאפשר להדפסת המידע הנגיש לתוכנית בעת עצירתה[12].
אנו נבחר להתמקד במדריך זה בכמה בסיסיות, לבדיקת תכולת זיכרון ולבדיקת תכולת הרגיסטרים.

תוכן רגיסטרים

על מנת לבדוק את תוכן הרגיסטרים של הקוד המדובג, בעת עצירתו, נשתמש בפקודה הבאה[13]:

```
info registers
```

שתדפיס לנו את תכולת כל הרגיסטרים (מלבד רגיסטרי הוקטור וה-floating, שאינם בחומר הקורס גם כך).

הצצה לזיכרון

הדרך הפשוטה ביותר לבחון את הזיכרון, היא להדפיס את התוכן של כתובת מסוימת[14]:

```
x <address>
```

שמדפיס את התוכן של הכתובת address לפי הקינפוג האחרון של הפקודה. איך מקנפגים אותה?

```
x/nfu <address>
```

כאשר לאחר ה-'/' יכולים להופיע אחד או יותר מהקינפוגים הבאים:

- n משמעותו כמה הדפסות רצופות לעשות. למשל, אם אנו מדפיסים מערך שמתחיל בכתובת 0x1000 וגודלו 5, נוכל לבצע את הפקודה x/5 0x1000 ונקבל את כל איברי המערך.
- f משמעותו הפורמט שבו יודפס תוכן הזיכרון[15]. ברירת המחדל היא הקסדצימלי (וניתן לחזור אליה עם 'x'), אך אפשר גם שלם דצימלי עם סימן ('d'), שלם דצימלי ללא סימן ('u'), שלם אוקטלי ('o'), בינארי ('t'), תו ('c') ועוד. למשל, אם אנו רוצים להדפיס את הערך שבכתובת 0x1000 בבינארי, נריץ x/b 0x1000.
- u משמעותו כמה בייטים "ליחידת הדפסה" (כלומר, בסופו של דבר אנו מדפיסים n*u בייטים מהזיכרון, במקבצים בגודל u). כאשר 'b' מסמל בייט בודד, 'h' מסמל 2 בייטים, 'w' מסמל ארבעה בייטים ו-'g' מסמל שמונה בייטים. למשל, אם נרצה להדפיס 8 בייטים החל בכתובת 0x1000, נעשה x/g 0x1000.

---

[12] https://sourceware.org/gdb/current/onlinedocs/gdb/Data.html#Data
[13] https://sourceware.org/gdb/current/onlinedocs/gdb/Registers.html#Registers
[14] https://sourceware.org/gdb/current/onlinedocs/gdb/Memory.html
[15] https://sourceware.org/gdb/current/onlinedocs/gdb/Output-Formats.html#Output-Formats

ניתן גם לשלב בין הפקודות. למשל, עבור הדפסה של 3 איברים במערך של long, בפורמט דצימלי עם סימן, שמתחיל ב-0x1000, נבצע את הפקודה x/3dg 0x1000.

דרך נוספת לבדיקת תוכן זיכרון, היא באמצעות הפקודה הבאה:

```
print <var>
```

פקודת print מיועדת בעיקר לשפות עיליות, ולא לאסמבלי, אך עדיין ניתן להשתמש בה כדי להדפיס labels בקוד שלנו אם נרצה. אם נבחר לעשות זאת, נצטרך לציין casting לפני המשתנה (התווית) שנרצה להדפיס, כי print נזקקת למידע הזה על מנת לבצע את ההדפסה (נזכור שבאסמבלי אין טיפוסים והבחירה האם תווית מסוימת היא int, או בכלל char, היא בידי המתכנת בלבד).
לכן, למשל, אם בקוד שלנו יש את התווית num ונרצה להדפיס את התוכן של num כ-int (כמספר שלם, בגודל 4 בייטים, עם סימן), נכתוב print (int)num.

מעקב אחר מחסנית הקריאות

אם עצרתם את התוכנית ואתם שואלים את עצמכם "How did I get here?", הפקודה הבאה היא בשבילכם[16]:

```
backtrace
```

<u>הערה:</u> ניתן גם לקצר ולכתוב bt (אין קשר לעדי ביטי).
בפשטות, פקודה זו תדפיס לכם את כל מחסנית הקריאות עד כה וכך תוכלו לדעת מאיפה באתם ולאן תשובו (בתקווה).

תצוגה של קוד האסמבלי במקביל לריצה

אם תרצו לעבוד עם תצוגה של קוד האסמבלי שלכם במקביל לריצה, ניתן לבצע:

```
layout asm
```

וכעת להמשיך עם אותן פקודות שבמדריך זה ובתיעוד GDB באופן כללי.
אם תרצו להוסיף במקביל לתצוגת קוד האסמבלי, גם תצוגה של ערכי הרגיסטרים, ניתן לבצע:

```
layout regs
```

יציאה מ-GDB

היציאה נעשית באמצעות הפקודה:

```
q
```

לרוב, GDB ישאל אתכם האם אתם בטוחים שאתם רוצים לצאת. נגיד לו y ונצא.

[16] https://sourceware.org/gdb/current/onlinedocs/gdb/Backtrace.html#Backtrace

דוגמאות

לסיום המדריך, נציג מספר דוגמאות לשימוש ב-gdb בעת דיבוג.

דוגמת הרצה פשוטה של קוד אסמבלי

נשתמש בקוד (חסר המשמעות) הבא:

```
.global _start

.section .data
x: .int 0x12345678
y: .int 0xdeadbeef

.section .text
_start:
 movl $x, %eax
 movl (y), %ebx
 movq (%eax), %rax
 movq (%ebx), %rbx
```

ונשים לב שאם ננסה להריץ אותו, נקבל שגיאה.

```
student@ubuntu18:~/Desktop$ as ex.asm -o ex.o
student@ubuntu18:~/Desktop$ ld ex.o -o ex.out
student@ubuntu18:~/Desktop$ ./ex.out
Segmentation fault (core dumped)
```

הידד, זו סיבה להשתמש בדיבאגר! הדבר הראשון שנעשה, יהיה להשתמש ב-objdump של לינוקס, עם הדגל D- (האות הגדולה חשובה) על מנת להדפיס גם את הכתובות אליהן ייטענו החלקים של קובץ הריצה (הקוד והזיכרון גם יחד):

```
student@ubuntu18:~/Desktop$ objdump -D ex.out

ex.out:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0:       b8 c4 00 60 00          mov    $0x6000c4,%eax
  4000b5:       8b 1c 25 c8 00 60 00    mov    0x6000c8,%ebx
  4000bc:       67 48 8b 00             mov    (%eax),%rax
  4000c0:       67 48 8b 1b             mov    (%ebx),%rbx

Disassembly of section .data:

00000000006000c4 <x>:
  6000c4:       78 56                   js     60011c <_end+0x4c>
  6000c6:       34 12                   xor    $0x12,%al

00000000006000c8 <y>:
  6000c8:       ef                      out    %eax,(%dx)
  6000c9:       be                      .byte 0xbe
  6000ca:       ad                      lods   %ds:(%rsi),%eax
  6000cb:       de                      .byte 0xde
```

אנחנו רואים למשל שהתווית x היא השם של הכתובת 0x6000c4 והתווית y היא השם של 0x6000c8. זה ישמש אותנו בהמשך, כשנרצה לבדוק את התוכן של התוויות האלו.

כעת נתחיל להשתמש ב-GDB:

```
student@ubuntu18:~/Desktop$ gdb ./ex.out
GNU gdb (Ubuntu 8.1-0ubuntu3.1) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./ex.out...(no debugging symbols found)...done.
(gdb)
```

לאחר הרצתו, הוא מחכה להתחיל לקבל פקודות. אם נריץ אותו ללא עצירה, נקבל את אותה השגיאה:

```
(gdb) r
Starting program: /home/student/Desktop/ex.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004000c0 in  _start ()
```

רק שהפעם אנו גם יודעים באיזו כתובת קרתה השגיאה. נבדוק מה יש באותה כתובת (בעזרת ה-objdump שעשינו קודם) ונראה שזו הכתובת של הפקודה האחרונה שביצענו. כבר יש לנו פקודה חשודה (ולכן יכולנו לשים label על אותה פקודה. נניח לקרוא לו dbg כדי להבין למה שמנו אותו, ואז להריץ את הקוד מחדש דרך האסמבלר והלינקר ולהתחיל את gdb שוב, כשאנחנו יודעים בדיוק איפה נרצה לעצור ולבדוק את הקוד), אך לשם ההדגמה נבצע כמה שלבים מקדימים, שיראו לנו שימוש בפקודות שראינו בתחילת המדריך.

תחילה, נשים breakpoints בנקודת ההתחלה, שמסומנת על ידי התווית _start ורק אז נריץ:

```
(gdb) b _start
Breakpoint 1 at 0x4000b0
(gdb) r
Starting program: /home/student/Desktop/ex.out

Breakpoint 1, 0x00000000004000b0 in _start ()
```

כעת, כשהתוכנית עצרה, נוכל להתקדם בזהירות, ולבצע פקודה אחת בלבד (ולאחריה, התוכנית עוצרת שוב):

```
(gdb) si
0x00000000004000b5 in  _start ()
```

כעת התוכנית עצרה בכתובת 0x4000b5, שזו אכן הכתובת של הפקודה השניה בקוד (חיזרו ל-objdump לוודא זאת). נרצה לבדוק שהפקודה בוצעה בהצלחה. איך נעשה זאת? נדפיס את תוכן הרגיסטרים ונבדוק מה מכיל רגיסטר rax.

```
(gdb) info registers
rax            0x6000c4 6291652
rbx            0x0      0
rcx            0x0      0
rdx            0x0      0
rsi            0x0      0
rdi            0x0      0
rbp            0x0      0x0
rsp            0x7fffffffe0d0   0x7fffffffe0d0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
r13            0x0      0
r14            0x0      0
r15            0x0      0
rip            0x4000b5 0x4000b5 <_start+5>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

נשים לב ל-RIP, שמכיל את הכתובת של הפקודה הבאה לביצוע. נשים לב גם -EFLAGS, ש-GDB מפרסר לנו ומספר אילו דגלים דולקים כרגע. גם RSP נתון לנו כאן, במידה ואנו מבצעים מעקב אחר המחסנית. לבסוף, נשים לב ל-RAX, שאכן מכיל את 0x6000c4 כפי שרצינו. הפקודה הראשונה אכן בוצעה בהצלחה.

בנוסף להדפסת הרגיסטרים, בואו נדגים את הדפסת תוכן הזיכרון:

```
(gdb) x 0x6000c4
0x6000c4:       0x12345678
(gdb) print (int)x
$1 = 305419896
```

אז מה אנו רואים? ביצוע של שתי פקודות.
בפקודה הראשונה ביקשנו להדפיס את התוכן של הכתובת 0x6000c4, שאנו יודעים (מה-objdump) שהיא הכתובת של התווית x. ההדפסה היא לפי ברירת מחדל בהגדרות, כלומר הדפסה בהקסדצימלי, 4 בייטים, של איבר אחד החל מכתובת הזיכרון הנתונה. לכן קיבלנו את 0x12345678 (שימו לב שההדפסה היא לאחר קריאת תוכן הזיכרון כ-little endian. בכתובת 0x6000c4 עצמה יש את הערך 0x78).
בפקודה השנייה השתמשנו ב-print, עם casting של התווית x להיות מהטיפוס int. ההדפסה היא אכן הערך הדצימלי של 0x12345678.

כעת נתקדם שתי פקודות, מאחר ואנו רוצים להגיע לפקודה הרביעית והחשודה:

```
(gdb) si 2
0x00000000004000c0 in  _start ()
```

הפקודה כוללת פניה לכתובת שרגיסטר rbx מחזיק. נבדוק מהי הכתובת ע"י הדפסת הרגיסטרים (חתכנו מהתמונה את רגיסטרי הסגמנט מאחר והם אינם רלוונטיים כאן).

```
(gdb) info registers
rax            0xdeadbeef12345678       -2401053092306725256
rbx            0xdeadbeef       3735928559
rcx            0x0      0
rdx            0x0      0
rsi            0x0      0
rdi            0x0      0
rbp            0x0      0x0
rsp            0x7fffffffe0d0   0x7fffffffe0d0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
r13            0x0      0
r14            0x0      0
r15            0x0      0
rip            0x4000c0 0x4000c0 <_start+16>
eflags         0x202    [ IF ]
```

ואכן רגיסטר rbx מכיל איזושהי כתובת שאינה בטווח שלנו ולכן נקבל segmentation fault כשננסה לגשת לכתובת זו.
עוד דבר מעניין הוא התוכן של rax, אך להבין מה קרה שם נשאר לכם כתרגיל.

דוגמת debugging של קוד אסמבלי עם מחסנית קריאות

ננסה לחקור את התנהגות הפונקציה foo הנתונה:

```
.global _start

.section .text

foo:
        movq    %rdi, %rax
        cmpq    $1, %rdi
        je      .L5
        pushq   %rbx
        movq    %rdi, %rbx
        leaq    -1(%rdi), %rdi
        call    foo
        addq    %rbx, %rax
        popq    %rbx
        ret
.L5:
        ret
_start:
        movl    $5, %edi
        call    foo
exit_prog:
        movq $60, %rax
        movq $0, %rdi
        syscall
```

```
example_stack.out:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <foo>:
  400078:       48 89 f8                mov    %rdi,%rax
  40007b:       48 83 ff 01             cmp    $0x1,%rdi
  40007f:       74 12                   je     400093 <foo+0x1b>
  400081:       53                      push   %rbx
  400082:       48 89 fb                mov    %rdi,%rbx
  400085:       48 8d 7f ff             lea    -0x1(%rdi),%rdi
  400089:       e8 ea ff ff ff          callq  400078 <foo>
  40008e:       48 01 d8                add    %rbx,%rax
  400091:       5b                      pop    %rbx
  400092:       c3                      retq
  400093:       c3                      retq

0000000000400094 <_start>:
  400094:       bf 05 00 00 00          mov    $0x5,%edi
  400099:       e8 da ff ff ff          callq  400078 <foo>

000000000040009e <exit_prog>:
  40009e:       48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  4000a5:       48 c7 c7 00 00 00 00    mov    $0x0,%rdi
  4000ac:       0f 05                   syscall
```

כפי שניתן לראות (ואם עברתם כבר את החומר של תרגול 4 ואתם יודעים את קונבנציית הקריאות של gcc), בתחילת הריצה, בפונקציה _start, קיימת קריאה ל-foo עם פרמטר בודד שערכו 5.

אם נרצה לחקור מה מחזירה foo מה נעשה? נבדוק את ערכו של rax (שוב, לפי הקונבנציה) בעת ההגעה ל-exit_prog.

נעשה זאת עם הצבת breakpoint ובדיקה של ערכי רגיסטרים ברגע העצירה:

```
(gdb) b exit_prog
Breakpoint 1 at 0x40009e
(gdb) r
Starting program:

Breakpoint 1, 0x000000000040009e in exit_prog ()
(gdb) info registers
rax            0xf      15
```

נגלה שערך rax הוא 15, כלומר foo(5) = 15 (וזה לא מפתיע, אם תבחנו את הקוד קצת בעצמכם).

אבל בדוגמה הזו אנחנו רוצים להסתכל על משהו חדש – על המחסנית.

קל להיווכח כי הפונקציה foo היא רקורסיבית. בשביל התרגיל וההתנסות, נרצה לחקור את התנהגות הקריאה הרקורסיבית ובפרט להסתכל על תוכן המחסנית בשלבים שונים של הריצה. נשים breakpoint על foo ונתחיל לבדוק את התנהגות התוכנית

```
(gdb) b foo
Breakpoint 1 at 0x400078
(gdb) r
```

ברגע העצירה הראשונה, נמצא את כתובת ראש המחסנית, המוצבע על ידי rsp

```
Breakpoint 1, 0x0000000000400078 in foo ()
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x0      0
rdx            0x0      0
rsi            0x0      0
rdi            0x5      5
rbp            0x0      0x0
rsp            0x7fffffffe098   0x7fffffffe098
```

ונבדוק את תוכן 8 ה-bytes החל מאותה הכתובת, בעזרת פקודת הדפסת תוכן כתובת, עם הדגלים g (להדפסת 8 bytes) ו-x (להדפסת ערך hexadecimal)

```
(gdb) x/xg 0x7fffffffe098
0x7fffffffe098: 0x000000000040009e
```

נגלה כי תוכן ראש המחסנית כעת הוא 0x40009e. האם זה הגיוני? שימו לב שזו בדיוק כתובת החזרה מהקריאה הראשונה ל-foo (ב-_start) ולכן זה בדיוק מה שציפינו שיהיה על המחסנית.

כעת נבצע המשך ריצה ונעצור בעת הכניסה בפעם השנייה אל foo. הפעם נדאג להסתכל על כתובת המחסנית המעודכנת (שימו לב שהמחסנית גדלה ב-0x10 בגלל שדחפנו אליה את rbx וביצענו עוד call – לכן כתובת המחסנית קטנה ב-0x10, מאחר והמחסנית גדלה כלפי מטה). בנוסף, הוספנו "3" להדפסה במטרה להדפיס 3 איברים בני 8 bytes ב-hexa החל מראש המחסנית החדש.

```
(gdb) x/3xg 0x7fffffffe088
0x7fffffffe088: 0x000000000040008e      0x0000000000000000
0x7fffffffe098: 0x000000000040009e
```

אנו רואים כעת בראש המחסנית את כתובת החזרה המתאימה לקריאה השנייה ל-foo (הקריאה הרקורסיבית הראשונה) ומעליה במחסנית את ערך rbx המקורי (מאותחל ל-0, אבל אפשר להתייחס לזה גם כערך זבל במקרה הזה) ואת ערך החזרה המתאים לקריאה הראשונה (בכתובת שכבר ראינו בבדיקת המחסנית הקודמת – המחסנית עוד לא התרוקנה ורק מתמלאת). נבדוק לאחר עוד 3 קריאות את המצב.

```
(gdb) x/9xg 0x7fffffffe058
0x7fffffffe058: 0x000000000040008e      0x0000000000000003
0x7fffffffe068: 0x000000000040008e      0x0000000000000004
0x7fffffffe078: 0x000000000040008e      0x0000000000000005
0x7fffffffe088: 0x000000000040008e      0x0000000000000000
0x7fffffffe098: 0x000000000040009e
```

בגלל שהקריאה רקורסיבית מאותו המקום בכל פעם – כתובת החזרה על המחסנית זהה. בנוסף, rbx שנשמר על המחסנית מחזיק את rdi של הקריאה הקודמת. לכן בפעם הראשונה שיש לו ערך משמעותי הוא 5, בפעם השניה 4 ובשלישית 3. התקדמנו כמה צעדים בריצה, עד לתווית L5. רגע לפני ביצוע ret ראשון וזה מצב הרגיסטרים (הרלוונטיים):

```
(gdb) info registers
rax            0x1      1
rbx            0x2      2
rcx            0x0      0
rdx            0x0      0
rsi            0x0      0
rdi            0x1      1
rbp            0x0      0x0
rsp            0x7fffffffe058   0x7fffffffe058
```

ואכן rax הוא 1, כפי שהיינו מצפים שיחזור כשאנחנו חוזרים לפי תנאי העצירה של רקורסיה זו.

כעת נבצע פקודה אחת (si) ואז נבדוק שוב את תוכן הרגיסטרים

```
(gdb) si
0x000000000040008e in foo ()
(gdb) info registers
rax            0x1      1
rbx            0x2      2
rcx            0x0      0
rdx            0x0      0
rsi            0x0      0
rdi            0x1      1
rbp            0x0      0x0
rsp            0x7fffffffe060   0x7fffffffe060
```

אכן המחסנית קטנה ב-0x8 (כי עשינו ret ולכן כתובת החזרה שהייתה בראשה – נשלפה). נעשה עוד 2 פקודות כדי לבצע גם את ה-pop שאחרי החזרה מהקריאה האחרונה ל-foo

```
(gdb) si
0x0000000000400091 in foo ()
(gdb) si
0x0000000000400092 in foo ()
(gdb) info registers
rax            0x3      3
rbx            0x3      3
rcx            0x0      0
rdx            0x0      0
rsi            0x0      0
rdi            0x1      1
rbp            0x0      0x0
rsp            0x7fffffffe068   0x7fffffffe068
```

ולא במפתיע – קיבלנו ש-rax ו-rbx התעדכנו בהתאם והמחסנית קטנה בעוד 0x8.

כעת נשים לב למשהו חשוב. אמנם המחסנית קטנה (כי ערכו של rsp גדל), אבל התוכן שלה לא נמחק. להלן

```
(gdb) x/9xg 0x7fffffffe058
0x7fffffffe058: 0x000000000040008e      0x0000000000000003
0x7fffffffe068: 0x000000000040008e      0x0000000000000004
0x7fffffffe078: 0x000000000040008e      0x0000000000000005
0x7fffffffe088: 0x000000000040008e      0x0000000000000000
0x7fffffffe098: 0x000000000040009e
(gdb) si
0x000000000040008e in foo ()
```

רגע לפני ביצוע ה-ret הבא, המחסנית נראית אותו הדבר (אבל ערך rsp שונה וזה משמעותי).

ניתן להמשיך ולדבג את המחסנית באופן דומה ובמקרים רבים – זו דרך מצוינת לשים לב לבעיות נפוצות בהקשרי המחסנית. למשל, סדר pop/push לא נכון, או גישה לאיברים במרחק לא מדויק מ-rbp, הוצאה של יותר מדי איברים מהמחסנית (שבסוף גורמת לביצוע ret על כתובת לא נכונה), ועוד מגוון רחב של טעויות נפוצות.

דבגו את הקוד בזהירות ותוכלו לעלות על כל הבעיות.

דוגמת debugging של קוד c ואסמבלי יחד

נשתמש בפונקציה foo מהדוגמה הקודמת, אך הפעם נוסיף לה קוד שפת c שמשתמש בה:

```
#include<stdio.h>

long foo(long n);

int main() {
        printf("%ld\n",foo(10));
        return 0;
}
```

```
.global foo

.section .text

foo:
        movq    %rdi, %rax
        cmpq    $1, %rdi
        je      .L5
        pushq   %rbx
        movq    %rdi, %rbx
        leaq    -1(%rdi), %rdi
        call    foo
        addq    %rbx, %rax
        popq    %rbx
        ret
.L5:
        ret
```

את הקוד הנ"ל קימפלו בעזרת gcc לקובץ ריצה אחיד (והוא רץ באופן תקין כי קוד האסמבלי נכתב לפי קונבנציות gcc.
כעת, נסתכל על חלק מה-objdump של קובץ הריצה. ה-objdump כולל הרבה תוספות של gcc מספריות המעטפת של C ואתם מוזמנים לחקור זאת בעצמכם, אך אנחנו נסתכל רק על החלק שכולל את main ו-foo:

```
00000000004004e7 <main>:
  4004e7:       55                      push   %rbp
  4004e8:       48 89 e5                mov    %rsp,%rbp
  4004eb:       bf 0a 00 00 00          mov    $0xa,%edi
  4004f0:       e8 1b 00 00 00          callq  400510 <foo>
  4004f5:       48 89 c6                mov    %rax,%rsi
  4004f8:       48 8d 3d b5 00 00 00    lea    0xb5(%rip),%rdi        # 4005b4 <_IO_stdin_used+0x4>
  4004ff:       b8 00 00 00 00          mov    $0x0,%eax
  400504:       e8 e7 fe ff ff          callq  4003f0 <printf@plt>
  400509:       b8 00 00 00 00          mov    $0x0,%eax
  40050e:       5d                      pop    %rbp
  40050f:       c3                      retq

0000000000400510 <foo>:
  400510:       48 89 f8                mov    %rdi,%rax
  400513:       48 83 ff 01             cmp    $0x1,%rdi
  400517:       74 12                   je     40052b <L5>
  400519:       53                      push   %rbx
  40051a:       48 89 fb                mov    %rdi,%rbx
  40051d:       48 8d 7f ff             lea    -0x1(%rdi),%rdi
  400521:       e8 ea ff ff ff          callq  400510 <foo>
  400526:       48 01 d8                add    %rbx,%rax
  400529:       5b                      pop    %rbx
  40052a:       c3                      retq

000000000040052b <L5>:
  40052b:       c3                      retq
  40052c:       0f 1f 40 00             nopl   0x0(%rax)
```

בפלט הנ"ל אפשר לראות איך שני קטעי הקוד השתלבו להם יחדיו (בעזרת הקומפיילר, האסמבלר והלינקר. על העבודה של השניים האחרונים אנחנו נלמד לעומק בקורס. על הקומפיילר תוכלו ללמוד בקורס קומפילציה).
<u>הערה:</u> בשביל לקבל פלט objdump כפי שראיתם לעיל, צריך להוסיף את הדגל no-pie- ל-gcc. אחרת נוצר קובץ שהוא PIE (Position Independent Executable), כלומר קובץ שלא ידוע לאן ייטען בזיכרון הריצה. יותר קשה לדבג קוד כזה (הכתובות בהן חלקי הקוד נמצאים לא ידועות לנו מראש) ולכן מאוד מומלץ בשלב debugging להשתמש בדגל זה.

כעת ניתן להמשיך ולדבג את הקוד בדיוק כפי שלמדנו במדריך הזה. אפשר לשים breakpoint בכל תווית שתרצו, בין אם היא במקור חלק מקוד C (למשל בתחילת main, שהיא פונקציה שנכתבה ב-C, אך השמות עוברות לתוויות גם בקובץ הריצה), או חלק מקוד האסמבלי (בתחילת foo כמו בדוגמה הקודמת). ניתן לבדוק היכן RIP נמצא (ובעזרת ה-objdump לדעת איפה הוא נמצא בקוד) ברגעים חשודים ולהדפיס את הרגיסטרים, או ערכי המחסנית, או סתם ערכים מהזיכרון. ברגע שהצלחתם להסתכל על הקוד המאוחד בקובץ ריצה עם כתובות ידועות (דגל no-pie-) אתם יכולים לדבג בכל דרך שרק תרצו.

הערות

המדריך נכתב על ידי בועז מואב לסמסטר אביב 2021 ועודכן עבור סמסטר חורף 2021-2022.

משהו חסר לכם במדריך? תוכלו לפנות לסגל הקורס בבקשה להוסיף לו עוד תכנים והמדריך ישודרג בהתאם לצרכי הסטודנטים והקורס.