ארגון ותכנות המחשב

תרגיל 2 - חלק רטוב

המתרגלת האחראית על התרגיל: טל נבל

שאלות על התרגיל – ב- **Piazza** בלבד.

הוראות הגשה:

- ההגשה בזוגות.
- על כל יום איחור או חלק ממנו, שאינו באישור מראש, יורדו 5 נקודות.
    - ניתן לאחר ב-3 ימים לכל היותר.
- הגשות באיחור יתבצעו דרך אתר הקורס.
- יש להגיש את התרגיל לפי ההוראות בסוף המסמך. אי עמידה בהוראות המפורטות תגרור הורדת נקודות יקרות.

חלק א – שגרות, קונבנציות, syscalls ומה שביניהן

מבוא

עבור חלק א' של התרגיל נדרש חומר הקורס עד הרצאה ותרגול 5 בלבד.

בתרגיל זה אתם תממשו קריאה של קלט טקסטואלי מקובץ וביצוע עליו מספר פעולות לוגיות פשוטות.
בתרגיל זה תממשו באסמבלי את הפונקציה שזו חתימתה:

```
unsigned short count_above(char separator, long limit);
```

פונקציה זו תקרא קלט מקובץ (ראו את השלבים הבאים), שיהיה מורכב ממספרים, שמופרדים ביניהם בעזרת התו שמתקבל כקלט בארגומנט `separator`. הפונקציה תצטרך להחזיר את כמות המספרים שהיו מעל הערך `limit` כאשר ההשוואה נעשית עם סימן. מימוש הפונקציה יהיה מורכב מכמה חלקים, שיוסברו בהמשך המסמך.

שלב ראשון – מציאת קובץ הקלט ופתיחתו

בשלב הראשון של `count_above` עליכם למצוא את קובץ הקלט.

- מאיפה מגיע קובץ הקלט?

אתם תצטרכו לקרוא לפונקציה הבאה:

```
char* get_path();
```

שמחזירה מצביע למחרוזת, שהיא ניתוב (path) של קובץ הקלט שממנו אתם צריכים לקרוא.
כעת תוכלו לפתוח את הקובץ ולהתחיל לקרוא 😊

שלב שני – קריאת קלט מקובץ

- איך נראה קובץ הקלט? הקובץ מתחיל במספר (הקובץ לא יהיה ריק). המספר יהיה כתוב ב-ASCII (ספרות ב-ASCII ואולי התו '-' שמסמן מספר שלילי, אם יש צורך) ואורכו יהיה לא ידוע, אך לא יותר מ-19 תווים.
  איך תדעו שנגמר המספר? אחת משתי האפשרויות הבאות:
  - תגיעו לתו המפריד שקיבלתם כפרמטר ל-count_above. תו זה לא יהיה '-', או אחת מעשר הספרות 0-9.
  - תגיעו ל-EOF.
- כעת צריך לתרגם. איך תתרגמו את המספר מ-ASCII לערך מספרי? אתם תצטרכו לקרוא לפונקציה הבאה:
  ```
  long atam_atol(char* num);
  ```
  הפונקציה תקבל null-terminated string של ספרות בלבד (באורך 20 לכל היותר, כולל ה-null terminator) ותחזיר את הערך המספרי. שימו לב שהמחרוזת חייבת להיות null-terminated (עם '\0' בסוף). הפונקציה `atam_atol` זמינה לכם בקובץ `get_path.o` (ראו הוראות הרצה בשלב הרביעי).
- כעת לכל מספר שתקראו, עליכם לבצע את השלב השלישי.

שלב שלישי – השוואה

- לכל מספר שקראתם (במהלך שלב 2), עליכם לבצע השוואה עם סימן. אם המספר שקראתם גדול (ממש) מה-`limit` שקיבלתם כארגומנט, עליכם לספור זאת.
- בסופה של הריצה (כשהגעתם ל-EOF), עליכם להחזיר כערך חזרה את כמות המספרים מהקלט, שהיו גדולים מה-`limit`.

שלב רביעי – טסטים

את hw2_part1.out יוצרים כך:

```
gcc basic_test.c get_path.o students_code.S -o hw2_part1.out
```

הערה חשובה:

אסור לשנות את שורת הקימפול הנ"ל, ובפרט אסור להוסיף את הדגל -fno-pie או -no-pie, מכיוון שאנחנו לא נשתמש בהם והקוד שלכם ייכשל בשלב היצירה. העובדה הזו גוזרת עליכם את ההגבלה הבאה – אסור להשתמש בשיטת מיעון אבסולוטית. בהמשך הקורס נלמד את הסיבה לכך שחוסר השימוש ב-no-pie גורר זאת.

שימו לב שהקובץ get_path.o מניח את קיומו של הקובץ in.txt בתיקייה. מותר לכם, לשם טסטים עצמיים, לשנות את in.txt ואת basic_test.c. בנוסף, מותר לכם לשנות גם את get_path.o (לייצר אחד משלכם), אך שימו לב שאתם שומרים על נכונות הקוד אם אתם עושים זאת.

הערות נוספות

רגע לפני הסוף, אנא קראו בעיון את ההערות, שנוגעות לדרך בה נבדק התרגיל.

1. את הקוד שלכם אתם צריכים לכתוב בעצמכם ובאסמבלי.
2. אנא ודאו שהתוכנית שלכם יוצאת (מסתיימת) באופן תקין, דרך main של קובץ הבדיקה שקורא לפונקציה שלכם, ולא על ידי syscall exit בקוד שכתבתם, במקרה שבו השתמשתם ב-syscall exit ראו תמונה מטה[1].
   הערה זו נכתבה בדם ביטים (של קוד של סטודנטים מסמסטרים קודמים).
   על מנת לוודא את ערך החזרה של התוכנית, תוכלו להשתמש בפקודת ה-bash הבאה:
   ```
   echo $?
   ```
   (תזכורת: ערך החזרה של התוכנית, אם יצאה בצורה תקינה, הוא הערך ש-main מחזירה ב-return האחרון שלה).
3. שימו לב שיהיה timeout (20-120 שניות, בהתאם לטסט) איתו הטסטים ייבדקו. כתבו קוד יעיל ככל האפשר.
4. אם הכל עובד כשורה, אתם יכולים לעבור לחלק ב' של תרגיל הבית, ולאחריו לחלק ג', שהוא בסך הכל הוראות הגשה לתרגיל כולו (שימו לב שאתם מגישים את שני החלקים יחד!).

1

חלק ב – פסיקות (אין קשר לחלק א)

מבוא

לפני שאתם מתחילים את חלק זה, אנחנו ממליצים לכם לחזור על תרגול 6 ולראות שאתם יודעים לענות על השאלות הבאות – מה זה IDT? מה מכילות כניסות ב-IDT? מהי שגרת טיפול בפסיקה? באילו הרשאות היא רצה? מהי חלוקת האחריות בין שגרת הטיפול לבין המעבד? באיזו מחסנית נשתמש? איך היא נראית בכל שלב? במה זה תלוי?
קראו את כל השלבים בחלק זה, לפני שתתחילו לעבוד על הקוד.
בתרגיל זה נרצה לכתוב שגרת טיפול בפסיקת המעבד המתבצעת כאשר מבצעים פקודה לא חוקית (כלומר, כאשר המעבד מקבל opcode שאינו מוגדר בו).

- כאשר המעבד מקבל קידוד פקודה שאינו חוקי, המעבד קורא לשגרת הטיפול בפסיקה ב-IDT.
- שגרת הטיפול בלינוקס שולחת סיגנל SIGILL לתוכנית שביצעה את הפקודה הלא חוקית. למשל:

https://github.com/torvalds/linux/blob/16f73eb02d7e1765ccab3d2018e0bd98eb93d973/arch/x86/kernel/traps.c#L321

נרצה לשנות את קוד הקרנל כך ששגרת הטיפול בפסיקה תשתנה (שאלה למחשבה - למה חייבים לשנות את קוד הקרנל?) ונעשה זאת באמצעות kernel module.

מה תבצע שגרת הטיפול החדשה?

שגרת הטיפול בפסיקה שלנו (שאותה אתם הולכים לממש באסמבלי בעצמכם, בקובץ ili_handler.asm), תיקרא **my_ili_handler** ותבצע את הדברים הבאים:

1. בדיקת הפקודה שהובילה לפסיקה זו. הנחות:
   - הניחו כי הפקודה השגויה היא פקודה של אופקוד בלבד. כלומר, לפני ואחרי ה-opcode השגוי אין עוד בייטים (אין legacy prefix, אין REX).
     - לכן, אורך הפקודה השגויה הוא באורך 1-3 bytes. בתרגיל זה הניחו כי אורך האופקוד השגוי הוא לכל היותר 2 בייטים.
2. קריאה לפונקציה what_to_do עם ה-byte האחרון של האופקוד הלא חוקי, כפרמטר.
   - היזכרו בחומר של קידוד פקודות:
     - אם האופקוד אינו מתחיל ב-0x0F, הוא באורך byte אחד.
     - אחרת (כן מתחיל ב-0x0F), אם הוא אינו מתחיל ב-0x0F3A או 0x0F38, אזי הוא באורך 2 בייטים. לכן, הניחו כי הבייט השני באופקוד אינו 0x3A או 0x38 (אין צורך לבדוק זאת).
   - דוגמאות:
     - עבור האופקוד 0x27, שהינה פקודה לא חוקית בארכיטקטורת x86-64, נבצע קריאה ל-what_to_do עם 0x27.
     - עבור האופקוד 0x0F04, גם לא חוקית, נבצע קריאה ל-what_to_do עם הפרמטר 0x04.
3. בדיקת ערך החזרה של what_to_do
   - אם הוא אינו 0 – חזרה מהפסיקה, כך שהתוכנית תוכל להמשיך לרוץ (תצביע לפקודה הבאה לביצוע מיד לאחר הפקודה הסוררת) וערכו של רגיסטר rdi% יהיה ערך ההחזרה של what_to_do. [2]
     - ➢ שימו לב #1: שימו לב ש-invalid opcode הינה פסיקה מסוג fault. חשבו מה זה אומר על ערכו של רגיסטר rip% בעת החזרה משגרת הטיפול ושנו אותו בהתאם.
     - ➢ שימו לב #2: היעזרו בספר אינטל, volume 3 [3], עמוד 223, המדבר על הפסיקה שלנו, בכדי לוודא את תשובתכם ל"שימו לב #1" וגם כדי להחליט האם יש error code או לא.
     - ➢ שימו לב #3: what_to_do הינה שגרה שתינתן על ידנו בזמן הבדיקה. אין להניח לגביה דבר, מלבד חתימתה (כלומר - שם השגרה, טיפוס פרמטר הקלט וטיפוס ערך החזרה).
   - אחרת (הוא 0) – העברת השליטה לשגרת הטיפול המקורית.

---

[2] ב"עולם האמיתי" אסור לשנות ערכים של רגיסטרים וצריך להחזיר את מצב התוכנית כפי שקיבלתם אותו. כאן אתם נדרשים כן לשנות ערך של רגיסטר, כך שמצב התוכנית לא יהיה כפי שהיה כשהתרחשה הפסיקה. זה בסדר, זה לצורך התרגיל 😊

[3] https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325384-sdm-vol-3abcd.pdf

לפני תחילת העבודה – מה קיבלתם?

בתרגיל זה תעבדו על מכונה וירטואלית דרך qemu (בתוך המכונה הוירטואלית - Virtualiception). על המכונה הזו, אנחנו נריץ [4] kernel module שיבצע את החלפת שגרת הטיפול לזו שמימשתם בעצמכם.
היות והקוד רץ ב-ring (0 kernel mode), במקרה של תקלה מערכת ההפעלה תקרוס. אך זה לא נורא! עליכם פשוט להפעיל את qemu מחדש.
לרשותכם נמצאים הקבצים הבאים בתיקייה part 2:

- initial_setup.sh - הריצו סקריפט זה לפני כל דבר אחר. סקריפט זה מכין את המכונה הוירטואלית לריצת qemu. <u>עליכם להריץ אותו פעם אחת בלבד</u> (לא יקרה כלום אם תריצו יותר, אך זה לא נחוץ).
  - יכול להיות שתצטרכו להריץ את הפקודה הבאה, לפני ההרצה (בגלל בעיית הרשאות):

```
chmod +x initial_setup.sh
```

- compile.sh - הריצו סקריפט זה בכל פעם שתרצו לקמפל את הקוד ולטעון אותו (עם המודול המקומפל) למכונה הוירטואלית של qemu (<u>שימו לב:</u> עליכם לצאת מ-qemu קודם).
  - גם כאן ייתכן ותזדקקו להרצה של chmod באותו אופן כמו בסעיף הקודם.
- start.sh - הריצו סקריפט זה כדי להפעיל את המכונה הוירטואלית של qemu, לאחר שקימפלתם את תיקיית code וטענתם אותה אל המכונה הוירטואלית של qemu.
  - גם כאן ייתכן ותזדקקו להרצה של chmod באותו אופן כמו בסעיף הקודם.
- filesystem.img - המכונה הוירטואלית אותה תריצו ב-qemu.
- קבצי הקוד שנכתוב, כחלק מהמודול (והיא זו שתקומפל ותרוץ לבסוף ב-qemu) וה-makefile:
  - ili_handler.asm, ili_main.c, ili_utils.c, inst_test.c, Makefile

איך הכל מתחבר - כתיבת המודול

בתיקיה code סיפקנו לכן מספר קבצים:

- **inst_test.c** – simple code example that executes invalid opcode. Use it for basic testing.
- **ili_main.c** – initialize the kernel module – provided to you for testing.
- **ili_utils.c** – implementation of ili_main's functionality – **<u>YOUR JOB TO FILL</u>**
- **ili_handler.asm** – exception handling in assembly – **<u>YOUR JOB TO FILL</u>**
- **Makefile** – commands to build the kernel module and inst_test.

ממשו את הפונקציות ב-ili_utils.c, כך שהשגרה my_ili_handler תיקרא כאשר מנסים לבצע פקודה לא חוקית. איך? Well, זהו לב התרגיל, אז נסו להיזכר בחומר הקורס. כיצד נקבעת השגרה שנקראת בעת פסיקה? פעלו בהתאם.
שימו לב כי ב-ili_utils.c אנו רוצים לגשת לחומרה. מהי הדרך לשנות קוד כאשר אנו רוצים לבצע פקודות ב-low level?
לאחר מכן, ממשו את הפונקציה **my_ili_handler** ב-ili_handler.asm שתבצע את מה שהוגדר בשלב II.

[4] למי שלא מכיר את המונח kernel module, בלי פאניקה (כי panic זה רע, אבל זה עוד יותר רע בקרנל. פאניקה! בדיסקו זה דווקא בסדר) – מדובר בדרך להוסיף לקרנל קוד בזמן ריצה (ניתן להוסיף לקרנל קוד ולקמפל לאחר מכן את כל הקרנל מחדש, אך כאן לא הזמן ולא המקום לזה). למעשה, נכתוב קוד שירוץ ב-kernel mode ולכן יהיה בעל הרשאות מלאות. אנו נדרש לזה – הרי אנו רוצים לשנות את קוד הקרנל.

זמן בדיקות - הרצת המודול

לאחר שסיימתם לכתוב את המודול, בצעו את השלבים הבאים:

1. הריצו את `./compile.sh` כדי לקמפל את קוד הקרנל ולהכניסו למכונת ה-QEMU
2. הריצו את `./start.sh` כדי לפתוח מכונה פנימית באמצעות QEMU
   a. התעלמו מכל המלל שיופיע, כולל הערות בצבעים אדומים וירוקים. זה תקין.
   b. לאחר העליה –
   משתמש: root, סיסמא: root
   כעת אתם בתוך ה-QEMU וכל השלבים הבאים מתייחסים לריצת QEMU.
3. `./bad_inst` כדי להריץ **את** הקוד `inst_test.asm`, עם הפקודה הלא חוקית (ולקבל הודעת שגיאה בהתאם). ניתן גם להריץ את `bad_inst_2` כדי להריץ את הקוד ב-`inst_test_2.asm`.
4. `insmod ili.ko` כדי לטעון את המודול שלכם (ודאו שהוא נטען ע"י הרצת dmesg).
5. `./bad_inst` כדי להריץ שוב, אך לקבל התנהגות שונה, מכיוון שהפעם ה-handler **שלכם** נקרא.

שימו לב שלאחר ביצוע שלב 1 תקבלו את האזהרה הבאה:

```
WARNING: could not find /home/student/Documents/atam2/part2/.ili_handler.o.cmd f
or /home/student/Documents/atam2/part2/ili_handler.o
```

יש להתעלם מאזהרה זו. שגיאות אחרות עלולות להצביע על בעיה מהותית ואף לאי יצירת הקבצים הרלוונטיים (בעיות בנייה, בעיות קישור ועוד), לכן שימו לב מהן ההערות שמופיעות בעת הרצת compile.

דוגמת הרצה תקינה ב-QEMU (עם הטסטים **inst_test** ו- **inst_test_2**, ומימוש **what_to_do** שסופק לכם כדוגמה):

```
root@ubuntu18:~# ./bad_inst
start
Illegal instruction
root@ubuntu18:~# insmod ili.ko
root@ubuntu18:~# ./bad_inst
start
root@ubuntu18:~# echo $?
35
root@ubuntu18:~# rmmod ili.ko
rmmod: ERROR: ../libkmod/libkmod.c:514 lookup_builtin_file() could not open buil
tin file '/lib/modules/4.15.0-60-generic/modules.builtin.bin'
root@ubuntu18:~# ./bad_inst_2
start
Illegal instruction
root@ubuntu18:~# insmod ili.ko
root@ubuntu18:~# ./bad_inst_2
start
Illegal instruction
```

what_to_do) מחזירה את הקלט שלה פחות 4. בטסט הראשון הפקודה הלא חוקית היא 0x27, לכן ערך החזרה הוא 0x23, שזה 35. ערך זה הוא גם ערך היציאה של התוכנית, כי כך נכתב הטסט[5], לכן echo $? מדפיס 35. בטסט השני, הפקודה הלא חוקית היא 0x0F04, לכן ערך החזרה של what_to_do הוא 0 והתוכנית חוזרת לשגרה המקורית לטיפול, ששולחת את הסיגנל Illegal Instruction).

[5] הטסט נכתב כך שמיד לאחר הפקודה הלא חוקית יש ביצוע של קריאת המערכת exit. אתם משנים את %rdi בשגרת הטיפול, לכן ערך היציאה של הטסט ישתנה בהתאם.

פקודות שימושיות

insmod ili.ko
(טוען את המודול ili.ko לקרנל ומפעיל את הפונקציה init_ko שבמודול)
rmmod ili.ko
(מפעיל את הפונקציה exit_ko שבמודול ומוציא את המודול ili.ko מהקרנל)
SHIFT + page up
(גלילת המסך למעלה)
SHIFT + page down
(גלילת המסך למטה)

תקלות נפוצות (מתעדכן)

במקרה של תקלת "אין מקום בדיסק" שמתקבלת בזמן הרצת compile/. – עליכם להוריד מחדש את הקובץ filesystem.img ולהחליף את העותק הישן באחד החדש ואז להריץ את compile/. שוב.

הערות כלליות

נשים לב שבתרגיל זה שינינו את הקוד של הגרעין! ומכיוון שזה קוד של גרעין אז אין לנו דרך לדבג אותו ולהבין אם הוא אכן עובד כפי שציפינו שיעבוד, אך אל חשש יש פיתרון!
על מנת להבין מה קורה בקרנל – תוכלו להשתמש בפונקציה print() המוגדרת בקובץ ili_main.c, ולראות את הודעות הקרנל ע"י כתיבה של הפקודה dmesg בטרמינל של ה-qemu.
למה print() ולא printf()?
הפונקציה printf() היא פונקציה של הסיפרייה libc, וכאשר אנו כותבים קוד גרעין או עושים לו מודיפיקציה אין לנו את האפשרות לגשת לסיפרייה זו (ואנחנו גם לא צריכים כי יש לנו רמת הרשאה 0, ו-libc היא ספרייה עבור משתמשים), לכן אנו צריכים לגשת לפונקציות שנמצאות בגרעין – אז עזרנו לכם וכתבנו עבורכם מעטפת נחמדה 😊
דבר נוסף, אנא קראו את ההערות בקבצי הקוד שעליכם למלא – אני בטוחה שזה יעזור 😊

תיעוד של qemu ניתן למצוא כאן: https://qemu.weilnetz.de/doc/2.11/qemu-doc.html

חלק ג' - הוראות הגשה לתרגיל בית רטוב 2

אם הגעתם לכאן, זו בהחלט סיבה לחגיגה. אך בבקשה, לא לנוח על זרי הדפנה ולתת את הפוש האחרון אל עבר ההגשה – חבל מאוד שתצטרכו להתעסק בעוד מספר שבועות מעכשיו בערעורים, רק על הגשת הקבצים לא כפי שנתבקשתם. אז קראו בעיון ושימו לב שאתם מגישים את כל מה שצריך ורק את מה שצריך.

עליכם להגיש את הקבצים בתוך zip אחד:

hw2_wet.zip

בתוך קובץ zip זה יהיו 2 תיקיות:

- part1
- part2

ובתוך כל תיקייה יהיו הקבצים הבאים (מחולק לפי תיקיות):

- part1:
  - students_code.S
- part2:
  - ili_handler.asm
  - ili_utils.c

בהצלחה!!!

Photo: Paste/Pinterest