ארגון ותכנות המחשב

תרגיל בית 4 (יבש)

מיכאל בייגל	213764251
אביב אביטל	207260258

חלק ראשון – קידוד פקודות וקבצי ELF (25 נקודות)

ניק ועדן הצליחו להביס את זאקום בעזרת קריאות המערכת שיצרו במטלה הקודמת, ומצאו בין אבני המאגמה שבו מקדשו נמצא, דיסק-און-קי מרותח עם סמלים עתיקים עליו. כשניסו לקרוא את הקובץ במחשב, הוא הופיעה בשפה מאוד מוזרה, אך הם יודעים ששמה של התוכנית הינה ForbiddenDropsOfZakum. עזרו לניק ועדן לפענח את התוכנית המצורפת לתרגיל, בעזרת הכלים שנלמדו בקורס (כגון readelf, objdump וכדומה).

סעיף 1 (10 נקודות)

1. מה גודל ה-Section Header Table של התוכנית? **64 בתים**
2. כמה Program Headers מוגדרים בקובץ? **9**
3. מה גודל כל כניסה ב-Program Header Table? **כל אחת 56**
4. עבור כל program header מסוג LOAD, הכניסו את נתוניו לטבלה הבאה (יתכנו שורות ריקות):

הרשאות (סמנו את ההרשאות)	כמה bytes ייקראו מהקובץ ל-segment	גודל שיתפוס בזיכרון הווירטואלי	כתובת טעינה לזיכרון הווירטואלי	מיקום בקובץ (offset בבתים)
R W E	0xaa8	0xaa8	0x400000	0x0
R W E	0x26c	0x270	0x600e10	0xe10
R W E	_____0x	_____0x	_____0x	_____0x

5. באיזו כתובת יתחיל הקוד את ריצתו? 0x4005d0
6. ידוע כי קיים משתנה בשם findme. השלימו את ערך האתחול החסר, ואת כתובתו:

char findme[15] = 133221333123111

כתובת findme הינה (בבסיס הקסהדצימלי) 0x601060.

המשך השאלה בעמוד הבא

סעיף 2 (15 נקודות)

בכל שורה בטבלה הבאה מופיע קוד קצר בפקודות אסמבלי, או קידוד שלו לשפת מכונה.

עבור כל שורה מלאו (כפי שמופיע בשורה הראשונה כדוגמה):

1. את קידוד הפקודות לפי סדר הופעתן, משמאל לימין, או את הפקודות לפי סדר תרגומן.
2. כתובת בזיכרון התוכנית שבו נמצא קידוד הפקודות. אם הקידוד מופיע בכמה אזורי זיכרון, בחרו באזור בעל הרשאות הרצה.

רמז: הכתובת בה מופיע הקידוד יכולה להיות שילוב של חלקים מקידוד של פקודות אחרות. מומלץ לקרוא על הכלי grep ולהשתמש בכלים כגון objdump ו-hexdump, שנלמדו בתרגול 8.

פקודות	כתובת	קידוד
movq %rcx,%rax	0x400758	48 89 c8
leave ret	0x40082f	c9 c3
cmp $0x601080,%rax mov %rsp,%rbp	0x400616	48 3d 80 10 60 00 48 89 e5
test %al,(%rax)	0x400835	84 00
movl $0x7271b848,%eax	0x4006de	b8 48 b8 71 72
leaq 0x200865(%rip),%rsi	0x400804	48 8d 35 65 08 20 00

חלק שני – הנדסה לאחור (25 נקודות)

ניק ועדן הצליחו בעזרת לבישת קסדת הזאקום שהכינו מזאקום לאחר הבסתו, שנתנה להם הרבה חוכמה, להוציא חלק מקוד ה-C הכולל את פונקציית main של הקובץ ForbiddenDropsOfZakum.

להלן הקוד שהצליחו להוציא: (secret אינו מוגדר ב-main ותוכנו אינו ידוע)

```
int main() {
    char password[16];

    printf("Enter the password in order to reveal the secret:\n");
    scanf("%s", password);

    if (brokenCheck(password) == 0) {
        printf("The secret is: %s\n", secret);
    } else {
        printf("You were wrong, and soon I will be revived!!\n");
    }

    return 0;
}
```

הם גילו כי הקובץ ישן וכתוצאה מהשנים הרבות שבילה במאגמה של זאקום, פונקציית הבדיקה של password, הנקראת brokenCheck, נהרסה.

ניק ועדן שמו לב, כי קוד מסוג זה בעל חולשה מסויימת, וכי ניתן לבצע התקפת **ROP** עלייה. על התקפת ROP ניתן לקרוא כאן: https://en.wikipedia.org/wiki/Return-oriented_programming

סעיף 1 (5 נקודות)

ניתן לראות כי לפי הקוד, המשתמש מצפה לקבל סיסמא באורך 16 תווים, ובעצם בעזרת הפונקציה scanf קורא את הסיסמא שהמשתמש מכניס אל תוך הבאפר password. מה הבעיה בשימוש ב-scanf בקוד זה?

הבעיה בscanf היא שהשימוש בו בקוד זה לא מגביל את קריאת המחרוזת בגודל. לכן, משתמש זדוני יוכל

להכניס ערך הגדול מ16 שאותו password מקבל, לעשות override לערכים אחרים במחסנית (למשל ערך

חזרה) ולהריץ קוד שלו במקום.

המשך השאלה בעמוד הבא

סעיף 2 (5 נקודות)

ניק ועדן התנסו עם התוכנה, וניסו להכניס את הסיסמה: "maplestoryisthebestnostalgicgame!!".
לאיזו כתובת תקפוץ פקודת ret שמבצעת הפונקציה main בסיום ריצתה?
רמז: מה אורך הקלט שהכניסו ניק ועדן? על מה הדבר משפיע?
הכתובת תהיה:
0x 65 6d 61 67 63 69 67 6c

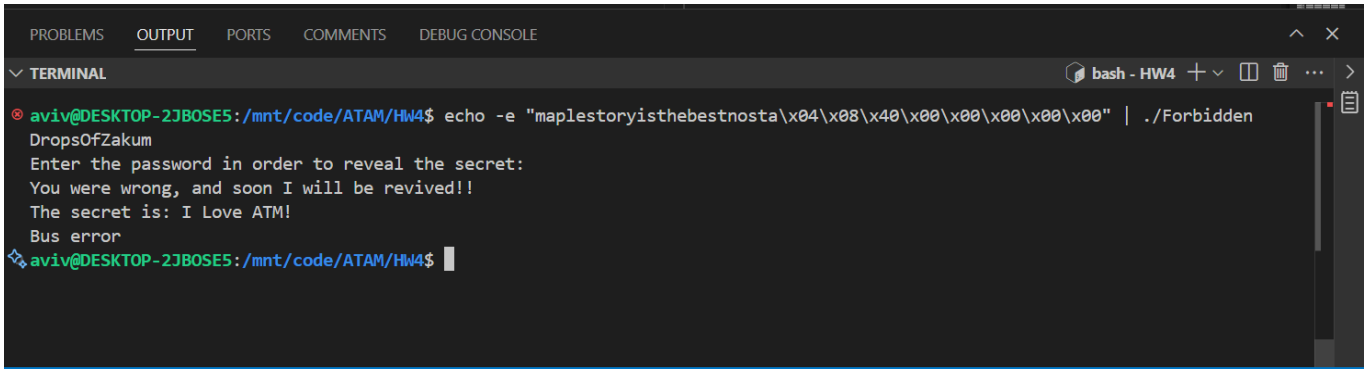
סעיף 3 (15 נקודות)

תנו דוגמא לקלט שיגרום לתוכנית להדפיס את secret, כלומר לבצע את שורה 8 בקוד ה-C. **יש להסביר בקצרה** את דרך פעולתכם כולל ניתוח הקלט שבחרתם. בתשובתכם השתמשו בפורמט \xHH כדי לציין קלט הקסהדצימלי. לדוגמא אם הקלט הינו האות a ואחריה בית עם ערך 0x8F שאחריו 0x90, כתבו "a\x8F\x90".
יש לצרף צילום מסך של ההדפסה. ייתכן וכי יהיו הדפסות נוספות שיקרו, וכי התוכנית לא תסתיים כראוי לאחר ההדפסה.
מומלץ להשתמש בכלי echo כדי להזין את הקלט לתוך הקובץ ForbiddenDropsOfZakum.

קלט לתוכנית: maplestoryisthebestnosta\x04\x08\x40\x00\x00\x00\x00\x00

הסבר: ראינו בסעיפים קודמים כי על מנת שנוכל לדרוס את ערך החזרה, עלינו לדרוס את 8 הבתים ששמורים במחסנית כreturn address. בגלל שהשמירה היא בlittle endian ו char נשמרים תו תו, עלינו לשמור את הכתובת אליה נרצה לקפוץ מהתו הקטן ביותר לתו הגדול ביותר. הכתובת אליה קפצנו היא הכתובת בה שומרים את התוכן של secret בתוך רגיסטר בשביל ההדפסה.

צילום מסך של ההדפסה:



חלק שלישי – לינקר סטטי (30 נקודות)

הילה, סטודנטית בקורס את"מ רוצה להתנסות בקישורי האסמבלי שלה, ובידע החדש שרכשה לגביי לינקרים, וכתבה 2 קבצים – a.o, b.o. הילה השתמשה במחשב ישן ותקול, שהלינקר הסטטי בו לא עובד כראוי, וכאשר הפעילה על הקבצים את הלינקר הסטטי, גילתה כי החורים שהלינקר אחראי למלא בזמן קישור סטטי כלל לא מולאו, וצריכה את עזרתכם בידע החדש שרכשתם בנושא הלינקרים כדי לתקן את קובץ הריצה שיצרה.

בדף הנספחים נמצאים הדפסות של כלים (כגון objdump, readelf) שהופעלו על הקבצים a.o ו-b.o, שני קבצי object files שאוחדו ביחד לקובץ ab.out יחיד.

סעיף 1 (3 נקודות)

מהי פעולת הקישור שהורצה על מנת ליצור את ab.out? יש לנמק בקצרה את מבנה הפקודה.

ld a.o b.o -o ab.out

מבנה הפקודה הוא ליצור את ab.out מ a.o ו b.o, כאשר הפרמטר הראשון הוא a.o. זאת כיוון שבטבלת

הסמלים ניתן לראות כי הסמלים שהוגדרו ב a.o מופיעים קודם

להלן פלט objdump של ab.out, עם חלקים חסרים וללא פיענוח הפקודות (בקובץ זה ולא בנספח):

```
Disassembly of section .text:
00000000004000b0 <_start>:
  4000b0:   48 c7 c6 ec 00 60 00
  4000b7:   48 8b 14 25 ec 00 60
  4000be:   00
  4000bf:   48 c7 c7 e4 00 60 00
  4000c6:   e8 0b 00 00 00
  4000cb:   48 89 c3
  4000ce:   e8 09 00 00 00
  4000d3:   48 89 c1
00000000004000d6 <foo>:
  4000d6:   b8 06 00 00 00
  4000db:   c3
00000000004000dc <cat>:
  4000dc:   48 c7 c0 e4 00 60 00
  4000e3:   c3

Disassembly of section .data:
00000000006000e4 <msg2>:
  6000e4:   08 07
  6000e6:   06
  6000e7:   05 04 03 02 01
```

```
00000000006000ec <msg1>:
  6000ec:   18 17
  6000ee:   16
  6000ef:   15 14 13 12 11
```

המשך השאלה בעמוד הבא

סעיף 2 (51 נקודות)

עליכם למלא את כל החלקים החסרים בפלט ה-objdump של ab.out לעיל, כך שהקוד שנוצר הינו הפלט התקין של הקשר הסטטי לאחר התיקון.

סעיף 3 (5 נקודות)

כאשר הילה הסתכלה על הפקודה מתחילה בכתובת 0x4000c6, תהתה מדוע האסמבלר לא השאיר חור ללינקר הסטטי למלא. מה קידוד הפקודה המתחילה בכתובת זו, ומדוע לא נותר חור ללינקר הסטטי למלא?

הפקודה המתחילה בכתובת זו היא e8 0b 00 00 00, או באסמבלי:

call foo(%rip)

הסיבה שאין חור ללינקר הסטטי למלא נובעת מ2 הדברים הבאים:

- הפקודה משתמשת בכתובת יחסית ולא אבסולוטית (ניתן לראות זאת על ידי הdisplacement)
- הפונקציה נמצאת באותו הsection

כתוצאה מכך, אין צורך בלינקר הסטטי כדי להגדיר איפה foo נמצא. הקובץ נע בשלמותו במרחב הזיכרון במידת הצורך ולכן הפקודה הנ"ל תהיה מוגדרת היטב גם ללא הקישור בין 2 הקבצים.

סעיף 4 (4 נקודות)

יש למלא עבור כל קובץ, אילו סמלים יש להגדיר כ-extern וכ-global **בקוד של הקובץ** על מנת שהקישור יתבצע כרצוי: (אם אין צורך למלא מיקום בטבלה, יש לסמן X או להשאיר ריק)

קובץ	סמלים שיש להגדירם כ-global בקובץ	סמלים שיש להגדירם כ-extern בקובץ
a.asm	_start msg2	msg1 cat
b.asm	msg1 cat	msg2

סעיף 5 (4 נקודות)

הילה, ששונאת לינקרים, החליטה לכתוב כל הקוד בקובץ אחד ולא לחלק את הקוד שלה לכמה קבצים, כדי שלא תצטרך להשתמש בלינקר סטטי. תארו יתרון אחד וחסרון אחד על ההחלטה שלה לכך:

יתרון: כל הפונקציות והמשתנים יהיו מוגדרים באותו מקום, ולכן לא יהיה ספק לגבי הגדרה. כלומר, אין צורך לשתף משתנים או פונקציות בין משתנים - כל מה שצריך להיות מוגדר יכול להיסגר ברמת אותו הקובץ, ולכן ברמת הקומפילציה כל הקישורים יהיו מוגדרים והלינקר הסטטי לא יצטרך לקשר אותם

חסרון: כל שינוי בקוד בכל פונקציה יצריך קמפול מחדש של כל הקוד, כאשר עבור קוד גדול במיוחד יכול להימשך זמן רב. לעומת זאת, עבור מספר קבצים בהם הקוד מופרד, ניתן לקמפל רק את הקובץ ששונה ולקשר אותם מחדש, מה שעשוי לחסוך בזמן ולהיות יעיל בהרבה (פעולת הקמפול פעולה כבדה)

חלק רביעי – לינקר דינאמי (20 נקודות)

הילה רוצה להריץ קוד שמשתמש בפונקציה printf מהספרייה הסטנדרטית של C, והשתכנעה כי יש צורך בלינקר סטטי, אך מתעקשת כי אין צורך בלינקר דינאמי, ולא מעוניינת להשתמש בו כדי להריץ את הקוד שכתבה.

סעיף 1 (4 נקודות)

מה על הילה לעשות על מנת שתוכל להריץ את הפונקציה בקוד המקורי שלה, ללא שימוש בלינקר דינאמי? אין לציין דגל, אלא להסביר מה התוצאה הרצויה.

היא צריכה לבצע לינקינג סטטי של הספרייה הסטנדרטית של C יחד עם הקובץ שלה. כאשר הילה משתמשת בלינקר סטטי הקוד שלה בנוסף לספריות הנדרשות, כולל הפונקציה printf, ייכללו ישירות בקובץ ההרצה הסופי (ה-executable). כך, כל התלויות הנדרשות יהיו כולן יחד בתוך הקובץ הסופי.

סעיף 2 (6 נקודות)

תנו 2 חסרונות עיקריים בחוסר שימוש בלינקר דינאמי במקרה הכללי (מה שימוש בלינקר דינאמי מייעל לנו?):

חסרון ראשון: גודל קובץ ההרצה:

כאשר לא משתמשים בלינקר דינאמי, יש לכלול את כל הקוד הנדרש מהספריות הסטנדרטיות ישירות בתוך קובץ ההרצה, דבר שמוביל לגידול משמעותי בגודל הקובץ. לעומת זאת, לינקינג דינאמי מאפשר לקובץ ההרצה להיות קטן יותר, שכן הוא מכיל רק את הקריאות הנדרשות לספריות הדינאמיות החיצוניות.

חסרון שני: תחזוקת ועדכון קוד:

כאשר לא משתמשים בלינקר דינאמי כל ספרייה נדרשת כלולה בקובץ ההרצה. לכן, כל שינוי או עדכון בקוד הספריות ידרוש קימפול וקישור מחדש של כל התוכנית כדי לכלול את השינויים. לעומת זאת, כאשר כן בלינקר דינאמי, עדכוני ספריות נעשים באופן עצמאי ולא דורשים קימפול מחדש של קבצי ההרצה, מה שמייעל את תחזוקת המערכת ומאפשר עדכונים שוטפים מבלי לשנות את קוד המשתמש.

המשך השאלה בעמוד הבא

סעיף 3 (5 נקודות)

הילה השתכנעה, וכעת החליטה להשתמש בלינקר דינאמי, כלומר תקשר באופן דינאמי את הספרייה הדינאמית glibc, ומשתמשת בהדפסות בקוד שלה ואך ורק בפונקציה printf מ-glibc. מה על הילה לעשות כדי שזמן **טעינת** התוכנית (כל הזמן מרגע "ההפעלה" ועד תחילת ריצת הקוד מה-entry point שלו) שלה יקטן? יש להשתמש בידע הנלמד בקורס בלבד.

בכדי שהקוד של הילה **יטען** בזמן קצר יותר, הילה יכולה להשתמש בlazy loading. כלומר, הילה יכולה להפעיל את הקשר הדינמי באופן lazy. בדרך זו, כאשר הילה תטען את התוכנית, הקשר הדינמי לא יבצע את התיקונים הנדרשים לריצה בזמן העליה, אלא רק כאשר תיקרא printf בפעם הראשונה, תתוקן טבלת הGOT לערך של printf (פתרון זה יעלה להילה בזמן ריצה).

סעיף 4 (5 נקודות)

בהנחה שהילה קימפלה את הקוד שלה עם הדגלים `-Wl,-zlazy`, מה יהיה ההבדל בין הקריאה לפונקציה printf בפעם הראשונה ובפעם השנייה?

בקריאה הראשונה ל-printf , התוכנית תצטרך לבצע את תהליך הטעינה הדינאמית של הפונקציה.

בפועל, מה שיקרה הוא שכתובת הפונקציה printf לא תהיה ידועה עד שהפונקציה תוזמן בפעם הראשונה. תהליך זה יגזול זמן נוסף, ולכן הקריאה הראשונה לפונקציה printf תהיה איטית יותר בהשוואה לקריאות שלאחר מכן. בקריאה שנייה לפונקציה printf אחרי שהפונקציה כבר נטענה פעם אחת, כתובתה בזיכרון כבר ידועה לתוכנית. לכן, הקריאה השנייה והקריאות הבאות לפונקציה יבוצעו ישירות וללא צורך בביצוע תהליך נוסף של טעינה וקישור, מה שהופך אותן למהירות יותר.

בהצלחה!