

הנדסה לאחר 236496 – תרגיל בית 1

מגישים: שליו ריסין, אלון פליסקוב

חלק ב – היכרות ראשונית עם קובץ C מהודר

נפרט על הטעויות בקבצים partb.nopt.s ו-partb.opt.s-i.

בקובץ partb.nopt.s :

1. בלייבל LC2, מופיעה המחזורת "%c\0" במקום המחזורת "%d\0". בגלל טעות זו, בפונקציה scanf, כאשר המשתמש מכניס מספר לקלט, אז המספר נקרא בתור מחזורת של ספרות ascii, במקום בתור מספר (int). בנוסף, תיקרא רק ספרה אחת מהקלט במקרה הזה, כי מחזורת הפורמט %c מתאימה לקליטת תו יחיד.
2. לפני הקריאה ל-scanf, לא נדחפים הפרמטרים למחסנית – גם הכתובת של המשתנה בו יישמר קלט המשתמש, וגם המחזורת המתאימה ל-scanf, שבמקרה זה היא "%d". המחזורת שמועברת ל-scanf היא אותה המחזורת שהועברה ל-printf, שהיא "enter your guess:". מכיוון שלא מופיעים במחזורת אחוזים כגון %c %d, לא ייקלט קלט מהמשתמש. על מנת לתקן את הטעויות, הוספנו שתי שורות לקוד, המסומנות בהערות.

```
50      call    _printf
51      lea     eax, [ebp-8]
52      mov     DWORD PTR [esp+4], eax           # first addition
53      mov     DWORD PTR [esp], OFFSET FLAT:LC2 # second addition
54      call    _scanf
```

3. ללייבל L3 אמורים להגיע אחרי ניחוש לא מוצלח, וממנו אמורים לקפוץ חזרה לנסיון הניחוש – L2, אך פקודת ה-jmp בסוף L3 לא קיימת, לכן כל נסיון ניחוש של המשתמש היה מצליח. כדי לתקן את טעות זו, הוספנו פקודה "jmp L6" תחת הלייבל L3. כעת, אחרי שהוספנו את השורה הזו, יש להוסיף גם את השורה "jmp L8" לפני הלייבל L3, כדי שבעת ניחוש מוצלח (L4), נקפוץ לסיום התכנית. ולא חזרה ל-L6.

בקובץ partb.opt.s :

1. נשים לב, כי נקראה הפונקציה time ונקראה הפונקציה rand, אך לא נקראה הפונקציה srand. הרצה של הפונקציה rand ללא שימוש ב-srand לפניה, מובילה לכך שה-seed קבוע ושווה ל-1, כלומר שבכל הרצה של התכנית, המספר ה"רנדומלי" יהיה זהה. תיקנו זאת על ידי הוספת שתי שורות:

```
28      mov     DWORD PTR [esp], 0
29      call    _time
30      mov     DWORD PTR [esp], eax           # first addition
31      call    _srand                         # second addition
32      call    _rand
```

כעת תיקרא srand עם הפלט של time כפרמטר, מה שיוביל לריצה המצופה של התכנית.

2. בשורה 42 (לאחר הוספת שתי השורות לעיל), מתבצעת קריאה ל-printf עם המחרוזת "Guess a number between 1 and 100\n0", אחרי שכבר הדפסנו את המחרוזת הזו עם puts. החלפנו את הלייבל של המחרוזת בשורה 41 ללייבל המתאים כך שתודפס המחרוזת המתאימה: "Enter your guess:\n0".

```
mov DWORD PTR [esp], OFFSET FLAT:LC1 # was LC0
```

3. בשורה 47 (לאחר הוספת שתי השורות), רשומה הפקודה "jge L2". הפקודה תגרום לכך שלא אפשר יהיה להגיע ל-L3/L5, כלומר לא ניתן לסיים את ריצת התכנית. בנוסף, זה לא מתאים ללוגיקה של התכנית. לכן, החלפנו את הפקודה ב-"jge L3" – ששם בודקים האם הניחוש נכון ואפשר לסיים את התכנית, או שהמספר קטן מדי, ויש לנסות שנית.

```
47 | jge L3 # Mistake, was L2
```

חלק ג – היריבים של קטאן

ראשית, נכנסנו לאתר, פתחנו את קוד ה-HTML של האתר, וזיהינו כפתור חבוי –

```
<div class="button disabled" onclick="challenge_me()">
Passwords Recovery</div>
```

לאחר מחיקת המילה "disabled", הכפתור נגלה ולחיצה עליו הובילה לאתר ה- passwords.recovery

The image shows a web browser displaying the login page for 'Rivals of Catan'. The page features a sunset background and the title 'Rivals of Catan' in a stylized font. Below the title are two input fields labeled 'Username' and 'Password', followed by two green buttons: 'Login' and 'Passwords Recovery'. At the bottom, a black banner reads 'Please log in to access this page.' To the right of the browser window, the source code is visible, showing the HTML structure. The 'Passwords Recovery' button is highlighted in blue, and its source code is shown as:

```
<div class="slot_container">
  <form name="login" method="POST" action="/login">
    <div class="plate"></div>
    <div style="position: relative;"></div>
    <div style="position: relative;"></div>
    <div class="button" onclick="attempt_login()">Login</div>
    <div class="button" onclick="challenge_me()">Passwords Recovery</div>
    <div class="alert">Please log in to access this page.</div>
  </form>
</div>
</div>
<script src="/static/login/js/login.js"></script>
</body>
</html>
```

The screenshot shows a coding challenge titled "Limited Resource" with a cartoon barbarian character on the left and a stone hut on the right. The challenge text is as follows:

CHALLENGE
Limited Resource

Write a program that counts the resources needed to buy or build the given goals. Each line in the input contains a single goal from {road, settlement, city, development}.

The program outputs 2 digit costs, space separated, with a new line at the end. For example, given the input: [road, road, city, development] the returned value would be the following

At the bottom, there is a resource bar with five items: WOOD (02), BRICK (02), WOOL (01), GRAIN (03), and ORE (04). Each item has a corresponding icon and a small house icon above it. On the right side of the interface, there is a hexagonal frame containing an image of a stone hut with a thatched roof and a small fire. Below the frame, there are three buttons: "Back", "Upload Solution", and "Submit".

נתבקשנו לכתוב תכנית מסוימת ולכונן ל-2KB ומטה:

כתבנו תכנית פשוטה ב-C שעושה את הנדרש. העברנו אותה ב-compiler explorer באינטרנט, כדי לקבל קוד אסמבלי. לאחר מכן, שינינו את כל הקריאות ל-printf \ scanf, וטענו אותן דינמית כפי שהראו בהדגמה בתרגול 2 (LoadLibraryA וכן הלאה). בקוד האסמבלי, המחרוזות היו נתונות תחת לייבלים, ושינינו זאת לדחיפה ידנית על המחסנית תוך כדי ריצה – כפי שראינו בהדגמה.

קמפלנו את קובץ האסמבלי לקובץ בינארי וחיברנו אותו למימוש של FindFunction, ואחר כך ל-PE header מתאים, תוך התאמת גודל ה-text section. העלינו את קובץ ה-exe לאתר, ועברנו לשלב הבא:

הורדנו את ה-recovery source, והתחלנו לקרוא אותו.

level 1

תחילה, הסתכלנו על “_main”, ושמנו לב שלפני ההדפסה של “Level 1 passed!”, משווים את הארגומנט הראשון שהפונקציה מקבלת, עם 1. גילינו שזו השוואה שבודקת האם argc גדול מ-1, ואם כן אז עברנו את השלב הראשון. מכיוון שתמיד $argc \geq 1$, כי תמיד שם התכנית הוא הארגומנט הראשון, אז השלב הראשון יעבור אם הכנסנו לפחות ארגומנט אחד נוסף.

level 2

לפני הקריאה ל-level1, מתבצעת קריאה ל-atoi, על האיבר השני במערך argv – שהוא בעצם ה-command line argument הראשון. את ערך החזרה (integer) מעבירים כארגומנט ל-level2. התכנית קוראת ל-scanf על מנת לקבל start offset ו-end offset, שאותם מוסיפים לכתובת של תחילת מערך בזכרון, ולבסוף מבוצע עליהן alignment לגודל של 4 בתים. לאחר מכן, נקראת הפונקציה printArray במידה והקלט שהכנסנו תקין – שהמספר הראשון הוא אי-שלילי. הפונקציה הזו מדפיסה למסך את תוכן הזכרון בין שתי הכתובות הללו. השתמשנו בפלט הפונקציה כדי לראות מה בדיוק קיים בזכרון, והשתמשנו במידע הזה כדי לעבור את רמה 2.

ראינו כי על מנת שיודפס “Level 2 passed!”, עלינו לעבור בהצלחה סדרה של 9 השוואות בין בתים שונים בזכרון. כברירת מחדל, ההשוואות לא עוברות – הבתים אינם זהים.

סדרת הבתים הראשונה: ראינו כי עם seed מקובע (0), אנחנו מקבלים באופן עקבי את 12 הבתים (הראשונים) הבאים, על ידי קריאה ל-printArray בריצת התכנית:

042C5444 5FAC5A1F 04AF1079

סדרת הבתים השנייה:

המספרים שנקבעו מראש בתכנית, והם:

```
403 mov DWORD PTR [ebp-46], 0x4FC3698C
404 mov DWORD PTR [ebp-42], 0x5444BA19
405 mov BYTE PTR [ebp-38], 0x2C
```

ראשית, שמים לב כי הסדרה הראשונה באורך 12 והשנייה באורך 9, לכן 3 בתים מהסדרה הראשונה הם מיותרים, ולא נערכת איתם השוואה. אלה הם שלושה הבתים הראשונים ברצף – התייחסו אליהם כ-don't-care.

נתבונן כעת בחלק הזה של התכנית, שרץ במסגרת level2:

L38:


```
lea eax, [ebp-52] # address of hexa variable
mov DWORD PTR [esp+4], eax # push hexa variable address
mov DWORD PTR [esp], OFFSET FLAT:LC16 # string "%X"
call _scanf
mov eax, DWORD PTR [ebp-8] # eax = start address
mov eax, DWORD PTR [eax] # eax = *eax
mov edx, eax # edx = array[start]
mov eax, DWORD PTR [ebp-52] # eax = hexa (4 bytes)
xor eax, edx # eax = eax xor edx
mov edx, eax # edx = eax
mov eax, DWORD PTR [ebp-8] # eax = start address
mov DWORD PTR [eax], edx #
add DWORD PTR [ebp-8], 4 # address += 4
```

החלק הזה מתבצע בתכנית מספר פעמים כמספר double words שמודפסות למסך בprintArray. הפונקציה scanf נקראת כדי לקרוא קלט הקסה דצימלי. הקלט (4 בתים) עובר פעולת xor יחד עם 4 בתים מתאימים במערך, והתוצאה דורסת את תוכן המערך.

למשל:

הקלט הראשון שהכנסנו הוא 0x88000000. התוצאה של ה-xor היא 8C2C5444. כעת, דורסים את מתבצע: 0x88000000 XOR 042C5444. עם התוכן החדש – 8C2C5444. ידענו כי פעולת ה-xor מקיימת שאם $A \oplus B = C$, אז גם $C \oplus B = A$. אז הסקנו כי אם נבצע XOR בין סדרת המספרים הראשונה לסדרת המספרים השנייה, תוך כדי התעלמות משלושת הבתים הראשונים בסדרה הראשונה – אז נקבל סדרת קלט מתאימה. ואז כשנכניס את הקלט הזה לתכנית בחלק של L38, תוכן המערך בזכרון (סדרת הבתים הראשונה) יידרס עם סדרת הבתים השנייה – מה שיגרום לשתי סדרות הבתים להיות זהות, ובסוף לסיום השלב.

הוכחה לסיום השלב השני:


Download Recovery Source

Command line arguments

0

Input:

0
12
88000000
46e39976
28fb54c3

Run Recovery Tool

Output:

Level 1 Passed!
042C5444 5FAC5A1F 04AF1079
Level 2 Passed!

Errors:

:Level 3

לאחר התבוננות נוספת בקוד ראינו שצריך לקרוא לפונקציה `_dummy__` על מנת לעבור את שלב 3. לצערנו הרב (☹) הפונקציה אינה נקראת באף שורה בקוד. לכן הרעיון היה לשנות את כתובת החזרה מ `main` כדי שתהיה הכתובת של `dummy` (חשבנו גם על לשנות את כתובת החזרה של `level 2` אבל כבר ראינו שנצטרך לשלב הבא את המשך `main` ולכן הלכנו עם הגישה הראשונה) בשביל לכתוב ל `stack` השתמשנו בחלק משלב 2 אשר כותב על המחסנית. כעת היינו צריכים לגלות מה הכתובת של `dummy` ומה הכתובת חזרה מ `main` (נזכור שהכתיבה בשלב 2 היא `xor` ולא פשוט החלפת התוכן לכן צריך לדעת את שניהם). לפי הקוד, בתחילת `main` הכתובת נשמרת על המחסנית בדיוק 8 בתים מתחת ל `return address` מ `main` (`ebp+4` אל מול `ebp-4`). בנוסף `main` של `main` נמצא בדיוק מעל `frame` של `level` ולאחר חישוב מהיר ראינו שאם בשלב 2 ניתן את המספרים 0 ו-72 `print array` תכתוב מספיק בתים מהמחסנית ונוכל לגלות את כתובת החזרה והכתובת של `dummy`:

קיבלנו:


Output:

```
Level 1 Passed!  
042C5444 5FAC5A1F 04AF1079 00000048  
00000000 006BFF30 006BFEE8 771AB119  
00401841 00000009 006BFF28 004019CD  
B904C0C7 00720DF8 00000000 00401771  
006BFF80 00401233  
Level 2 Passed!  
Level 3 Passed!
```

:Level 4

בתחילת הקוד קוראים לפונקציה signal עם הפונקציה handler ומספר הסיגנל 8. הקריאה מגדירה כי כעת שגרת הטיפול לסיגנל מספר 8 (floating point exception) שמתרחש בחלוקה ב0 היא handler.

החלוקה היחידה בתכנית הינה בפונקציה dummy שבה מחלקים בdivider, אשר ערכו מתקבל על ידי שימוש בארגומנט הראשון בתור seed להגרלת מספר רנדומלי. לאחר מכן יש מניפולציות אלגבריות על התוצאה כשלקראת הסוף עושים AND עם המספר 31. כלומר, modulo 32. לכן ניחשנו מספרים עבור הארגומנט הראשון עד שהמספר שנכנס לdivider היה 0 ועברנו את שלב 4:




Download Recovery Source

Command line arguments

Input:

0
72
88000000
46e39976
28fb54c3
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000



Run Recovery Tool

Output:

Level 1 Passed!
C02C5444 9CACA8F1 568B4DBC 00000048
00000000 006BFF30 006BFEE8 771AB119
00401841 00000009 006BFF28 004019CD
D656F83D 007E0DF8 0000001E 00401771
006BFF80 00401233
Level 3 Passed!
Level 4 Passed!

Errors:

אך כעת, מכיוון ששינינו את הארגומנט הראשון גם seedn בשלב 2 השתנה ולכן המידע בstack array שמוגרל באקראי השתנה ואנחנו לא עוברים את הבדיקה של השלב. לכן התאמנו את input למידע החדש ככה שהבדיקה בשלב 2 תעבור וקיבלנו:

The screenshot shows a web application interface with a green header and a red banner. On the left, there is a green button labeled "Download Recovery Source" with a cloud and download icon. Below it, the "Command line arguments" section contains the text "30 0". The "Input:" section is a text area containing a list of hexadecimal and alphanumeric strings: 0, 72, 4c000000, 85e36b98, 7adf0906, and several 00000000 strings. On the right, the "Output:" section displays the results of the process, showing four levels passed and a list of hexadecimal values. The "Errors:" section is empty.

:db_access

ראינו כי בקוד של dummy נקראת הפונקציה db_access. חשדנו שבעזרתה נוכל לחלץ את פרטי ההתחברות של המשתמשים. ראינו שלפני db_access, מופיעה השאילתא הבאה:

```
124 v LC7:
125 | .ascii "select username, password from users where username='\0"
```

ותוך כדי ריצת db_access, מתבצע:

1. חיבור ל-database.
2. שאילתא ל-database.
3. סגירת החיבור ל-database.

השאילתא ששואלים הינה:

`select username, password from users where username=' || argv[2] || '`

כאשר ראינו את השאילתא שנוצרת, עלה לנו הרעיון של SQL Injection, כלומר – אם נוכל להכניס ארגומנט מתאים ב-argv[2], כך שהתנאי של ה-where תמיד יתקיים (טאוטולוגיה) אז יחזרו כל ה-username וה-passwords.

בנוסף, לפי הפונקציה, הוגבלנו ב-11 תווים (לפי שורה 159) –

```
158     mov eax, DWORD PTR _arg
159     mov DWORD PTR [esp+8], 11
160     mov DWORD PTR [esp+4], eax
161     lea eax, [ebp-276]
162     mov DWORD PTR [esp], eax
163     call _strncat
```

בשביל לבצע SQL Injection, המחרוזת צריכה להכיל:

1. גרש (התו ') שסוגר את הגרש שכבר קיים במחרוזת – ואז מקבלים empty string.
2. תנאי נוסף ל-where כך שיהיה טאוטולוגיה.
3. הערת שורה (שני מקפים --) לביטול הגרש שמתווסף אחר כך.

הגענו למחרוזת הבאה, כפי שרואים בתמונה:

The screenshot shows a web application security tool interface. On the left, there is a 'Download Recovery Source' button with a cloud icon. Below it, the 'Command line arguments' section contains the input '30 "' OR 3<4 --"'. The 'Input:' section shows a list of hexadecimal values: 0, 72, 4c000000, 85e36b98, 7adf0906, and several 00000000 values. At the bottom left is a 'Run Recovery Tool' button with a person icon. On the right, the 'Output:' section displays a list of hexadecimal values: 00000000, 006BFF30, 006BFEE8, 771AB119, 00401841, 00000009, 006BFF28, 004019CD, D656F83D, 00770DF8, 0000001E, 00401771, 006BFF80, 00401233. Below these are the messages 'Level 2 Passed!', 'Level 3 Passed!', and 'Level 4 Passed!'. At the bottom right, the 'Errors:' section is empty.

קיבלנו פרטים של ארבעה משתמשים, והצלחנו להתחבר בעזרתם למערכת.