החולשה

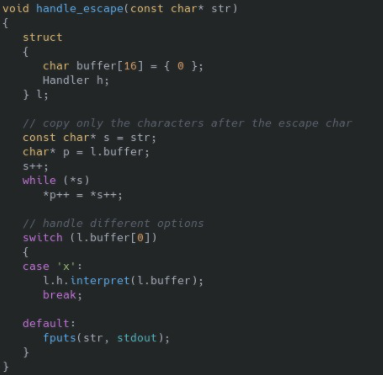
על מנת לנצל את החולשה אנו משתמשים בעקרונות שונים כמו יצירת משתנה סביבה, דריסת virtual table pointer ועוד...אך כל אלו לא היו שימושיים לנו ללא החולשה אשר פותחת לנו את "קן הצרעות" אשר מאפשר לנו לתקוף את המערכת ולקרוא לפונקציה .

ראשית, נגדיר את החולשה

סוג: Stack buffer overflow

הפונקציה בה קיימת החולשה: handle\_escape

חלק הקוד בו קיימת החולשה:



מדוע זאת חולשה?

גודל המערך char buffer מוגדר בזמן קומפילציה להיות בגודל 16 בתים וכל תא מאותחל עם הערך 0(הכוונה לערך 0 ממש, **לא** לערך '0' אסקי) בתור תכונה של סטראקט(מבנה) בשם l. משמע כאשר נקרא לפונקציה handle\_escape יאותחלו במחסנית הקריאות של התוכנית 16 בתים אשר ייעודם יהיה עבור מבנה הנתונים l.buffer .

בקטע הקוד המסומן במסגרת צהובה מתרחשת העתקה, אנו מעתיקים את ערכו של s (לא כולל התו הראשון) ל- l.buffer. אנו מבצעים העתקה זאת על פי תנאי התנאי while(\*s)משמע כל עוד לא הגענו לתא האחרון של s, וכל פעם מתקדמים לתא הבא של הסטרינג )sאשר מתקבל בתור פרמטר של הפונקציה) ו- p (אשר מאותחל להצביע על התא הראשון במערך l.buffer) ללא התחשבות בקיבולת התווים המקסימלית שl.buffer- יכול להכיל. משמע למרות שהוקצו לl.buffer- 16 בתים בלבד זה לא מונע מאיתנו לכתוב לזיכרון המחסנית אשר נמצא אחר איפה ש-l.buffer הוקצה למרות שזאת לא כוונת כותב הקוד, ובכך אנו מסוגלים לכתוב ולשכתב חלקים בזיכרון המחסנית ללא אישור ולנצל חולשה זאת כרצוננו. (לדוגמה ניתן לנצל זאת על מנת לדרוס את ערך החזרה של eip ובכך להחדיר shellcode זדוני אשר ייתן לנו הרשאות של root). ואכן כמו שציינו, זאתי חולשה מסוג Stack buffer overflow כיוון שאנו בעלי היכולת לשנות תאי זיכרון במחסנית הקריאות מחוץ לקטע של מבנה הנתונים אשר הוגדר לנו(במקרה זה l.buffer).

התקפה

על מנת לבצע התקפה מוצלחת עלינו לשאול את עצמנו כמה שאלות לדוגמה:

כיצד ניתן לנצל את החולשה המתוארת על מנת לקרוא לפונקציה ?

כיצד ניתן להגיע בכלל להגיע לקטע הקוד שבפונקציה handle\_escape ?

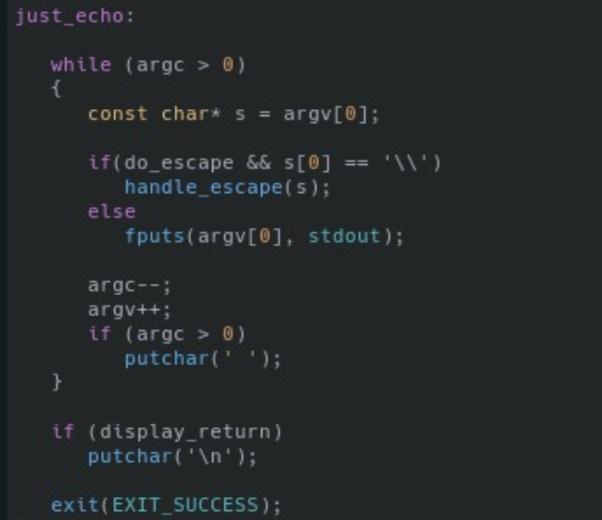
התקפה זאת מורכבת מכמה חלקים שונים ולכן ננסה לחלק ולפשט אותה לכמה שיותר שלבים.

ראשית,

קריאה לפונקציה handle\_escape

נתחיל מלראות איפה בקוד יש קריאה לפונקציה handle\_escape.

נגלה כי יש קריאה יחידה לפונקציה זאת והיא בקטע הקוד הנ"ל אשר נמצא בפונקציית המיין:



נשים לב כי הקוד קורא לפונקציה הנחשקת אם"ם מתקיים התנאי הבא:

ערך המשתנה הבוליאני do\_escape הוא true.

ערך האות הראשונה בסטרינג s אשר מצביע לסטרינג argv[0](אחד מן הפרמטרים אשר מתקבלים בקריאה לקובץ ההרצה..) הוא '\\' .

בנוסף אסור לשכוח שתנאי זה עטוף בתוך while אשר פועל אם"ם ערך המשתנה argc גדול מ-1.

argc הוא משתנה אשר מונה את כמות הארגומנטים אשר התקבלו עם הפעלת קובץ ההרצה. הוא תמיד גדול מ-1 בעת הכניסה לפונקציית המיין בכיוון שבצורה דיפולטיבית למרות שלא רושמים את זה מפורשות הארגומנט הראשון במקום 0 במערך argv הוא השם המלא של קובץ ההרצה.

אז רגע, מדוע לבדוק תנאי שמתקיים תמיד?

לאחר בדיקה קלה נגלה כי בפונקציית המיין יש את הקוד הנ"ל



משמע, ערך argc נעשה קטן באחד והסטרינג הראשון אשר argv מצביע עליו הוא זה שהיה במקום השני לפני זה.

כעת, כיוון שערך המשתנה הבוליאני do\_escape מאותחל להיות false עלינו למצוא דרך לשנות את ערכו לtrue. נסתכל בקוד ונשים לב כי אכן יש שורה אשר בקוד אשר משנה את ערכו לtrue:



אך איך נגיע אליה?

על מנת להגיע אליה יש לבצע 2 דברים

1. נסתכל על קטע הקוד, ניתן להבין כי אם הארגומנט השני אשר ניתן למיין יהיה e"-" ערך המשתנה do\_escape ישתנה לtrue.
2. אך על מנת להגיע לקטע הקוד עלינו להיכנס לתוך התנאי העוטף, וניתן להכנס לתנאי אם"ם ערך המשתנה הבוליאני allow\_options הוא true.

נסתכל ונגלה כי ערך המשתנה allow\_options הוא אמת אם"ם המשתנה env מצביע על כתובת כלשהי אשר אינה NULL. אך מתי זה מתקיים? 

ממבט על מימוש הפונקציה dupenv נגלה כי הפונקציה מחזירה העתק של הערך של המשתנה הסביבתי ECHOUTIL\_OPT\_ON ואם לא קיים היא תחזיר NULL. לכן כל מה שעלינו לעשות על מנת שערך המשתנה allow\_options יהיה אמת זה ליצור משתנה סביבתי בעל השם הנ"ל עם ערך כלשהו, (נזכיר כי אנו עובדים בלינוקס) ולכן לפני הרצת התוכנית נרשום את הפקודה:

export ECHOUTIL\_OPT\_ON=1

וכעת כאשר נריץ את התוכנית ערך המשתנה הבוליאני allow\_options יהיה אמת ונכנס לתנאי.

נניח כי שם קובץ ההרצה הוא a.out אז הרצת השורה

./a.out -e

תשנה את ערך המשתנה do\_escape לאמת כשרצינו אך זה עדיין לא מספיק על מנת לקרוא לפונקציה handle\_escape! עלינו לדאוג כי התנאי s[0]=='\\' יתקיים.

קטע הקוד



מתקיים פעמיים!



ולכן יש לנו צורך בארגומנט למיין נוסף אשר ערך התא הראשון יהיה '\\'! נעזר בפייתון בשביל לייצר ארגומנטים כרצוננו,

./a.out $(python2 -c 'print("-e")') $(python2 -c 'print("\\")')

והידד! כאשר נריץ שורה זאת אכן יתקבל כי תקרא הפונקציה handle\_escape אשר ייחלנו להכנס אליה.

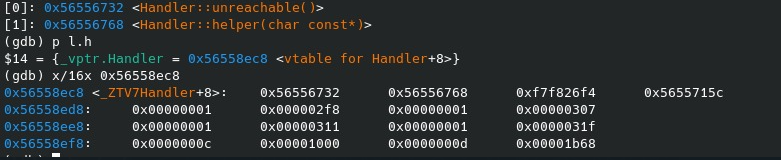
ניצול החולשה בפונקציה handle\_escape

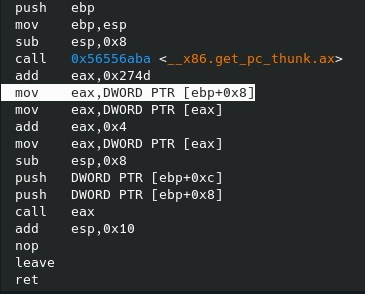
הרעיון הכללי:

באמצעות ניצול חולשת ה Stack buffer overflowנשנה את ערך המצביע virtual table pointer של האובייקט h ממחלקת Handler אשר נמצא בסטראקט(מבנה) l כך שכאשר נקרא לפונקציה interpret של מחלקת Handler במקום לקרוא לפונקציה helperהיא תקרא לפונקציה unreachable.

לאחר הרצת הקוד והדיבוג שלו נגלה כי ערך המצביע virtual table pointer של l.h מוגדר במחסנית הקריאות **לפני** המערך l.buffer משמע כאשר נדרוס את הערכים אשר נמצאים אחרי l.buffer המצביע לטבלה הווירטואלית יהיה אחד מהם, ונוכל לשנות את ערכו כרצוננו.

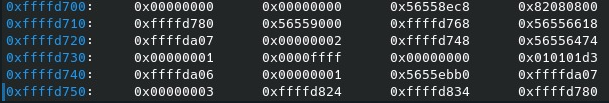
אך לאיזה ערך נשנה אותו?

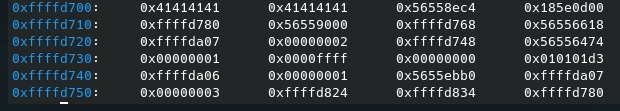
ניתן לראות בתמונה כי המצביע לטבלה הווירטואלית מצביע להיכן שנמצאת הכתובת של הפונקציה unreachable ו-4 בתים לאחר מכן נמצאת הכתובת לפונקציה helper.



מהסתכלות על קוד האסמבלי של התוכנית ניתן להבין כי הקוד בעצם ניגש לכתובת אליה המצביע לטבלה הוירטואלית מצביע, מתקדם 4 בתים קדימה (add eax, 0x4) ולאחר מכן לוקח מהתא בזיכרון את הערך הנמצא בו וקורא לו(במקרה שלנו זאת הכתובת לפונקציה helper ). ולכן ניתן להסיק כי אם נקטין את ערך המצביע לטבלה הווירטואלית ב-4 הפונקציה שתקרא במקום תהיה unreachable.

נסתכל בזיכרון המחסנית

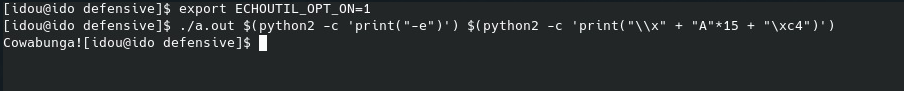


עלינו "לרפד" את התאים בזיכרון אשר באים לפני ערך הכתובת של המצביע הווירטואלי ולאחר מכן להקטין ב-4 את הכתובת. כך זה יראה: 

בעקבות הזרימה הלוגית של הפונקציה handle\_escape על מנת לקרוא לפונקציה interpret עלינו להוסיף את האות x לאחר התו הראשון '\\' של הארגומנט השני למיין, לאחר מכן את הריפוד ולבסוף את ערך ההקסה החדש אשר באמצעותו נדרוס את הקודם.

נבצע זאת באמצעות הפקודה

./a.out $(python2 -c 'print("-e")') $(python2 -c 'print("\\x" + "A"\*15 + "\xc4")')

והרי ! השגנו את התוצאה רצויה: 

ההגנה

במהלך ההתקפה עברנו על נושאים שונים והשתמשנו בטכניקות רבות על מנת לבצע את הנדרש, אך הכל מתחיל ונגמר **בחולשה** עצמה, לכן על מנת למנוע התקפה זאת ולהגן על הקוד שלנו עלינו להגן מפני החולשה Stack buffer overflow.

כיצד נעשה זאת?

נאתחל את הסטרינג l.buffer בצורה דינאמית כך שיהיה בגודל של str + 1 (בשביל התא הסופי אשר יהיה NULL אשר אומר כי זהו סופו של הסטרינג) ולכן לא נחרוג משום גבולות זיכרון כי גודלו של l.buffer יהיה בדיוק כנדרש ! (לא לשכוח לעשות #include <cstring> בראש הקובץ)

תיקון

void handle\_escape(const char\* str)

{

struct

{

char\* buffer = 0;

Handler h;

} l;

// copy only the characters after the escape char

const char\* s = str;

l.buffer = new char[strlen(str)+1]; //dynamic alloc

char\* p = l.buffer;

s++;

while (\*s)

\*p++ = \*s++;

// handle different options

switch (l.buffer[0])

{

case 'x':

l.h.interpret(l.buffer);

break;

default:

fputs(str, stdout);

}

}