

# ***Hírközlő rendszerek biztonsága laboratórium***

## **Számítógépes rendszerek sebezhetőségének vizsgálata**



Budapesti Műszaki és Gazdaságtudományi Egyetem  
Híradástechnikai Tanszék  
2009.

# Számítógépes rendszerek sebezhetőségének vizsgálata

készítette: Hornák Zoltán, 2003. okt. 3.

(átvette a CrySys laboratórium 2009-ben)

*"Minden érdekes program tartalmaz  
- legalább egy változót  
- legalább egy ciklust  
- és legalább egy hibát"*

~ Murphy programokra és programozókra vonatkozó 1. törvénye ~

## 1 Mérés célja

A programfejlesztés mai technikája mellett szinte minden rendszerben rendkívül sok olyan programozási hiba marad, amelyek a valós használat során gyakorlatilag sohasem jelentkeznek, és így a valós adatokkal végzett tesztek során felderítetlenül maradnak. Ezen ártalmatlannak tűnő, a hétköznapi működést legtöbbször nem is befolyásoló hibák egy rosszindulatú támadó számára gyakran olyan lehetőségeket rejtenek, amelyek segítségével könnyen visszaéléseket tud elkövetni egy rendszerben.

A mérés célja annak bemutatása, hogy egyszerű programozási hibák milyen komoly biztonsági lyukakhoz vezethetnek, és hogy ezen veszélyek – bár elsöre elméleti, rendkívül nehezen kihasználható hibáknak tűnhetnek – milyen könnyen kihasználhatóak.

A mérés során a két leggyakoribb és egyben leglátványosabb betörési módszerrel ismerkedünk meg. Az úgynevezett buffer overflow (BoF – puffer túlcsordulásos), illetve a printf hibákból eredő biztonsági lyukak kihasználási lehetőségeivel.

## 2 Buffer overflow

A buffer overflow a hagyományos programozási nyelvekben (C, C++, Pascal) egy gyakran elkövetett hiba, amikor is egy fix hosszúságúra lefoglalt tömb, illetve buffer határait a program nem ellenőrzi és így bizonyos helyzetekben (tipikusan valamilyen túlzottan hosszú bementeti érték hatására) a tömb számára lefoglalt memóriatartományon kívül is felülír értékeket. Ezen értékek felülírása pedig nem várt módon megváltoztathatja egy program működését. Mint látni fogjuk a gyakorlati életben egy támadó egyszerű módszerekkel úgy manipulálhatja ezt a nem kívánt helyzetet, hogy az általa készített tetszőleges programkód lefuttatását is el tudja érni.

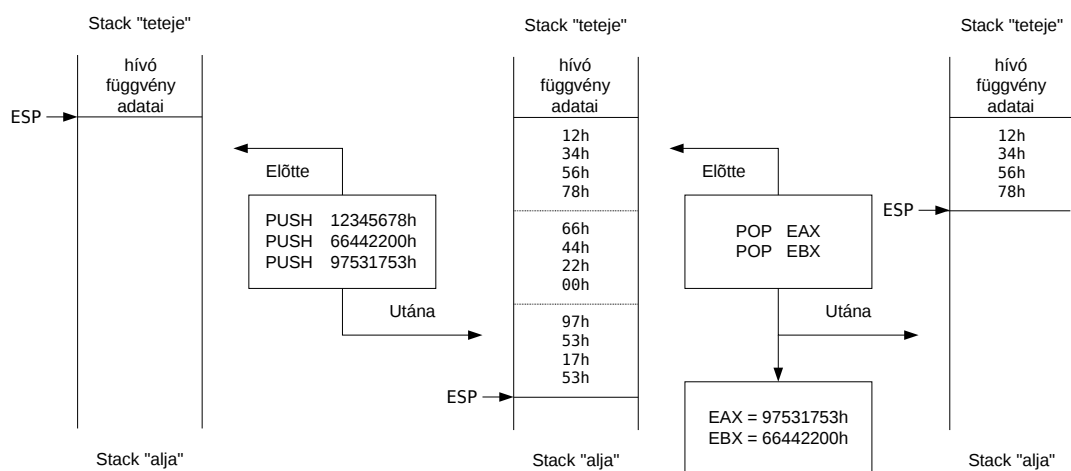
A legnagyobb veszély akkor jelentkezik, ha a szóban forgó fix hosszúságú tömböt lokális változóként definiálják, ugyanis ilyenkor a tömb a stacken tárolódik, amiből következően a tömb határán túlírva lehetőség nyílik egy függvény visszatérési címének felülírására. Ennek segítségével pedig egy támadó elérheti, hogy egy program futtatása az általa meghatározott címen – tipikusan ahol az általa elhelyezett rosszindulatú kódsorozat található – folytatódjon.

Hogy a támadási módszer pontos működését megértsük nézzük először hogyan működik a függvényhívás, paraméterátadás, illetve a lokális változók lefoglalása hagyományos programozási nyelvekben.

## 2.1 Függvényhívás mechanizmusa

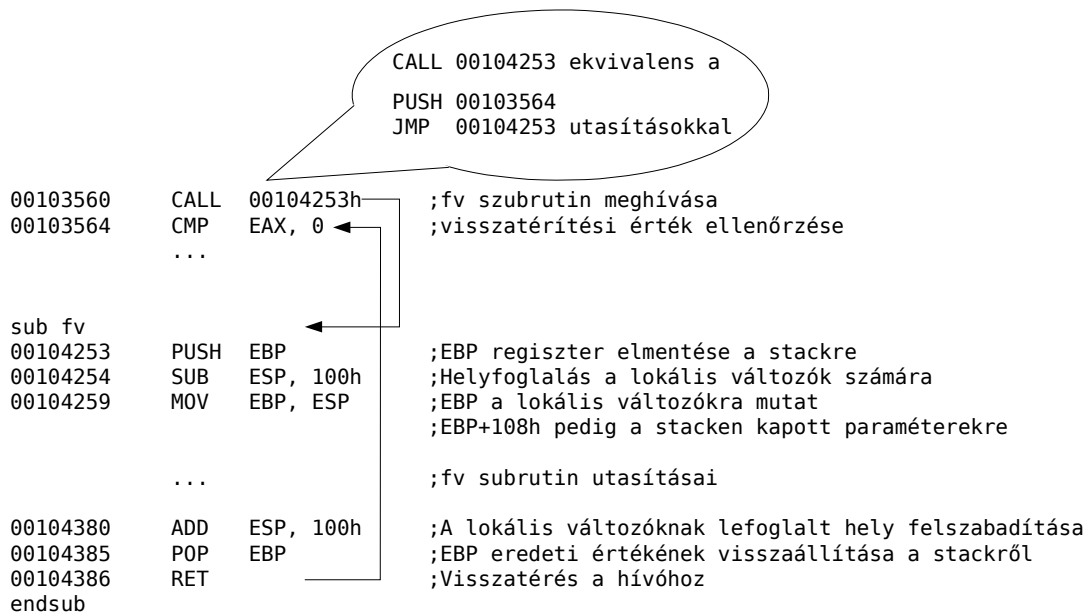
Ismert, hogy a fordítóprogramok a magas szintű programozási nyelveket gépkód szintű utasításokra fordítják le. Gépkód szinten pedig nincsenek olyan absztrakt fogalmak, mint típusok, függvények, hanem csak egyszerű utasítások vannak, amelyek sorozatából képesek a fordító programok bonyolultabb műveleteket összeállítani.

A függvény-paraméterek átadását, illetve a függvényhívásokat egy verem (stack) segítségével oldják meg. A verem egy külön lefoglalt memóriaterület, amelyhez a hozzáférés last-in-first-out (LIFO) elvű, azaz az utoljára berakott érték olvasható ki legelőször. A verem kezelését a processzor egy speciális regiszter, a stack pointer (ESP) segítségével oldja meg. Amikor valamit a verembe be kell tenni (PUSH assembly művelet), akkor az ESP értékét 4-gyel csökkenti a processzor, és az így mutatott memóriacímre helyezi be a PUSH utasítás operandusát. Amikor a stackból olvasnak (POP assembly utasítás), akkor a processzor az ESP által mutatott memóriacímről kiolvassa az ott található értéket és beírja azt a POP utasítás operandusába, majd az ESP értékét növeli 4-gyel. Vegyük észre, hogy ez a mechanizmus úgy tűnik, mintha egy verem objektumot valósítana meg, holott a valóságban csak két egyszerű műveleti lépést összefogó utasítás-párosról van szó.



1. ábra: A stack működése (PUSH és POP utasítások)

A függvényhívás (CALL), illetve a "visszatérés a hívóhoz" (RET) műveleteket is a verem segítségével valósítja meg a processzor. A függvény belépési pontjára való ugrást a CALL utasítás hajtja végre, amely a CALL utasítást követő gépkódú utasítás memóriacímét elmenti a verembe (csökkenti ESP értékét 4-gyel, majd az így mutatott címre menti a CALL utasítás utáni memóriacímet) majd a függvény első utasításával folytatja a végrehajtást. A függvényből való visszatérés pedig a RET utasítással történik, amely kiolvassa az stackból (az ESP által mutatott memóriapozícióból) a visszatérési címet (normál esetben a hívó CALL utasítás utáni utasítás címét), majd ott folytatja a végrehajtást. Látható, hogy itt is egy aránylag egyszerű utasítás-páros segítségével egy rendkívül rugalmas, akár többször is egymásba ágyazható függvényhívási mechanizmust lehet kialakítani.



2. ábra: Függvényhívás és visszatérés

## 2.2 Paraméterátadás mechanizmusa

A függvények rendkívül előnyös tetszőleges egymásba ágyazhatóságának megőrzése érdekében a paraméterátadás mechanizmusát is a stacken keresztül oldják meg. Azaz az adott függvényt hívó programrészlet a függvénynek szánt paramétereket úgy adja át, hogy hívás előtt elmenti azokat a stackre (a PUSH utasítás segítségével), amit majd a függvényből való visszatéréskor természetesen törölni kell a veremből. (Megjegyzés: az egyes programozási nyelvek abban is eltérnek egymástól, hogy az így a stacken feleslegessé vált paramétereket kinek kell "kitisztítania", a hívónak vagy a hívott függvénynek. A Pascal esetén a hívott fél szabadítja fel a stacket, míg C/C++ esetén ez a hívó feladata. A C hozzáállása annyiban megérthető, hogy változó számú paraméterrel működő függvények esetén gyakorlatilag csak a hívó tudja, hogy hány értéket mentett a stackre, így ő tudja azt legbiztosabban kezelni.)

A hívott függvény az így megvalósított paraméterátadáskor a stacken az ő szemszögéből a következőket látja: <visszatérési érték>, <első paraméter>, <második paraméter>, ..., <utolsó paraméter>. (Megjegyzés: vegyük észre, hogy ennek a sorrendnek a biztosításához a hívónak először a függvény utolsó paraméterét kell a stackre PUSH-olnia és csak utoljára az elsőt. Ez a magyarázata annak, hogy a C nyelv miért visszafelé számolja ki egy függvény paramétereit, ha azok nem konkrét értékek.)

00103551	...	PUSH	_c ;A C
	és C++ is fordított sorrendben teszi a		
00103556		PUSH	_b ;
	stackre a hívási paramétereket, ezért értékeli		
0010355B		PUSH	_a ; ki
	azokat fordított sorrendben		
00103560		CALL	00104253 ;Az fv()
	függvény meghívása		
00103564		ADD	ESP, 0Ch ;A függvény-
	paraméterek törlése a stackről		
00103569		CMP	EAX, 0 ;Visszatérés
	i érték ellenőrzése		
	...		
	<b>int fv(int a, int b, int c)</b>		
00104253		PUSH	EBP ;EBP
	regiszter elmentése a stackre		
00104254		SUB	ESP, 100h ;Helyfoglalá
	s a lokális változók számára		
00104259		MOV	EBP, ESP ;EBP a
	lokális változókra mutat		
	08h pedig a stacken kapott paraméterekre		;EBP+1
	...		;fv()
	függvény utasításai		
0010437B		MOV	EAX, 0 ;Visszatérés
	i érték		
00104380		ADD	ESP, 100h ;A lokális
	változónak lefoglalt hely		;
	felszabadítása		
00104385		POP	EBP ;EBP
	eredeti értékének visszaállítása a stackről		
00104386		RET	;Visszatérés
	(hívási paraméterek a stacken		;
	maradnak)		

**3. ábra: Függvény-paraméterek átadása és felszabadítása C, C++ esetén (hívó takarít)**

00103551	...		
	PUSH	a	;A
00103556	PASCAL sorban teszi fel a hívási paramétereket	b	;a
	PUSH	c	
0010355B	stackre, és balról-jobbra értékeli ki őket.		
00103560	CALL	00104253	;Az fv()
	függvény meghívása		
			;A
	meghívott függvény törli mindent a stackről		
00103565	CMP	EAX, 0	;Visszatérés
	i érték ellenőrzése		
	...		
<b>Function fv(a, b, c:integer):integer</b>			
00104253	PUSH	EBP	;EBP
	regiszter elmentése a stackre		
00104254	SUB	ESP, 100h	;Helyfoglalás
	s a lokális változók számára		
00104259	MOV	EBP, ESP	;EBP a
	lokális változókra mutat		
			;EBP+1
	08h pedig a stacken kapott paraméterekre		
	...		;fv()
	függvény utasításai		
0010437B	MOV	EAX, 0	;Visszatérés
	i érték		
00104380	ADD	ESP, 100h	;A lokális
	változónak lefoglalt hely		
			;felsz
	abadítása		
00104385	POP	EBP	;EBP
	eredeti értékének visszaállítása a stackről		
00104386	RET	0Ch	;Visszatérés
	a függvény-paraméterek törlésével		
			;RET
	0Ch = RET + ADD ESP, 0Ch		

**4. ábra: Függvény-paraméterek átadása és felszabadítása Pascal esetén (a meghívott függvény takarít)**

Annak érdekében, hogy a hívott függvénynek ne kelljen POP utasításokkal kiolvasni a számára átadott paramétereket a 80x86 processzorcsalád esetében bevezették az EBP (base pointer) mutatót, amely segít a stack terület hagyományos memóriaterületként való megcímezésében. Azaz tipikusan egy függvény meghívásakor az ESP aktuális értékét az EBP-be másolva, az EBP-hez képesti relatív címmel megadott hívási paraméterek könnyen, közvetlenül elérhetők. Például [BP+4] hivatkozik a visszatérési címre, [BP+8] az első paraméterre, [BP+0Ch] a másodikra, és így tovább. (Az EBP regiszterre azért van alapvetően szükség, mert [ESP+4] memóriahivatkozási mód nincs.) Ezzel a címezési módszerrel gyakorlatilag a stack, illetve a stack pointer módosítása nélkül könnyen elérhetők az átadott paraméterek.

## 2.3 Lokális változók kezelése

Mint ahogy az előző példából is kiderül (2-es, 3-as, 4-es ábrák) a függvények lokális változóikat úgyszintén a stacken tárolják. Ezzel biztosítható, hogy egy függvény tetszőlegesen sokszor meghívhassa önmagát (rekurzió) anélkül, hogy a lokális változóinak tartalma "összekeveredne".

A stack tartalmának megértéséhez vegyünk egy egyszerű példát:

```
int SampleFunction(int a, int b, int c)
{
    int i,j,k;
    char buffer[100];

    buffer[0]=0;
    i=a;
    j=b;
    k=c;
    return 0;
}
```

5. ábra: Példa függvény a lokális változók kezeléséhez

Ebben az esetben a stack tartalma a függvény végrehajtása közben a következő lesz:

---

6. ábra: Stack tartalma a `SampleFunction()` függvény végrehajtása közben

```

int SampleFunction(int a, int b, int c)
00104253      PUSH      EBP      ;EBP
              regiszter elmentése a stackre
00104254      SUB       ESP, 70h  ;Helyfoglalás
              s a lokális változók számára
              ;100+4
              +4+4=112=70h
00104257      MOV       EBP, ESP  ;EBP a
              ;EBP a
              lokális változókra mutat:
              ;EBP+7
              8h pedig a stacken kapott paraméterekre
              ;A
              stack tartalma tehát:
              ;
              [EBP+80h]      c      ;
              [EBP+7Ch]     b      ;
              [EBP+78h]     a      ;
              [EBP+74h]
              ;
              <visszatérési cím>
              ;
              [EBP+70h]     <EBP  ;
              elmentett értéke>
              ;
              [EBP+6Ch]     i      ;
              [EBP+68h]     j      ;
              [EBP+64h]     k      ;
              [EBP+0]      buffer[0] ;
buffer[0]=0;
00104259      MOV       BYTE PTR
              [EBP+0], 0 ;EBP a buffer elejére mutat
i=a;

```

0010425D	MOV	EAX,
	[EBP+78h]	;[EBP+78h] = a
00104260	MOV	[EBP+6Ch],
	EAX	;[EBP+6Ch] = i
<b>j=b;</b>		
00104263	MOV	EAX,
	[EBP+7Ch]	;[EBP+7Ch] = b
00104266	MOV	[EBP+68h],
	EAX	;[EBP+68h] = j
<b>k=c;</b>		
00104269	MOV	EAX,
	[EBP+80h]	;[EBP+78h] = c
0010426F	MOV	[EBP+64h],
	EAX	;[EBP+64h] = k
<b>return 0;</b>		
00104372	MOV	EAX, 0
		;Visszatérés
	i érték	
00104377	ADD	ESP, 70h
		;A lokális
	változóknak lefoglalt hely	
		;
	felszabadítása	
0010437A	POP	EBP ;EBP
	eredeti értékének visszaállítása a stackről	
0010437B	RET	
		;Visszatérés
	(hívási paraméterek a stacken	
		;
	maradnak)	

7. ábra: Lokális változók kezelése – `SampleFunction()` függvény gépkódban

## 2.4 Túlcsordulásos hiba

A fenti ismertető után nézzük, mi történik, ha egy lokális változóként foglalt tömb túlcsordul. Tekintsünk ehhez egy olyan példát, amely nem ellenőrzi a tömb-határértékeket:

```

void BOF_ExploitableFunction(char *input)
{
    char buffer[100];

    strcpy(input,buffer); // strcpy nem vizsgálja buffer méretét
};

int main(int argc, char *argv[])
{
    BOF_ExploitableFunction(argv[1]);           // A
                                                paraméterként kapott értéket

                                                // egy
                                                az egyben átadja a

                                                //
                                                függvénynek

    return 0;
};

```

### 8. ábra: Buffer overflow hibát tartalmazó függvény

Ebben az esetben, ha 100 (pontosabban 99, mert a string végét lezáró 0-t is számolni kell) karakternél hosszabb szöveget adunk meg bemenetnek, akkor az `strcpy` a `buffer` elejétől elkezdi azt átmásolni, majd – mivel `buffer` hosszát nem ismeri – nem vizsgálja, hogy annak határát átlépi-e, így felül fog írni olyan memóriaterületet is, amely már máshoz tartozik.

Ezek a hibák a gyakorlatban "különös" hatásokat okoznak, hiszen olyan változók értéke is módosulhat ilyen módon, amelyeknek az adott programrészlethez egyébként semmi közük sincs. Súlyosabb a probléma akkor, ha – lokális `buffer` esetén – annyira túlírunk egy lefoglalt tömb területén, hogy a függvény visszatérési címet is felülírjuk a stacken. Ekkor a függvény végén visszatéréskor a processzor kiolvassa a stackből a felülírt visszatérési címet és megpróbálja ott folytatni a program végrehajtását. Tekintve, hogy ilyen véletlenszerű pozíció általában nincs értelmes futtatható kódrészlet, általában az ekkora mértékű túlcsoordulás hatására a program "elszáll".

Azonban, ha valaki ügyes módon olyan input adatot ad egy programnak, amely ilyen túlcsoordulásos hibát tartalmaz, hogy a függvény visszatérési címét nem értelmetlen utasításokra mutató, hanem értelmezhető kódsorozatra mutató címmel írja felül, akkor elérheti, hogy gyakorlatilag tetszőleges programot, parancssorozatot végrehajtsa az adott géppel.

A buffer overflow "lelegegánsabb" kihasználási módja az, ha egyből magában az inputban olyan futtatható kódsorozatot helyeznek el, amely segítségével a támadó át tudja venni az irányítást a gép felett, majd eléri, hogy a túlcsoordulás révén olyan visszatérési címmel írja felül a stacket, amely az inputban elhelyezett kódsorozatra mutat! Ekkor tehát nem csak azt tudja a támadó elérni, hogy tetszőleges helyen folytatódjék egy program futtatása, hanem azt is, hogy az általa megírt program induljon el a megtámadott gépen!

Ennek a támadási módnak így különös veszélye az, hogy nem szükséges semmilyen jogosultság a támadó részéről az adott gépen, illetve az, hogy bármilyen alkalmazáson keresztül, amelyben ilyen túlcsoordulásos hiba van és amelyik a külvilágból kaphat bemenetet, lehetőség nyílik betörésre. Így nem elegendő pusztán az operációs

rendszer, illetve a biztonsági modulok hibamentes implementációja annak érdekében, hogy ezen támadásokat kiküszöböljük, hanem az összes telepített programnak hibamentesnek kell lennie! Ez a követelmény pedig a mai rendszereknél nem teljesül, és gyakorlatilag nem is teljesíthető.

Találtak már buffer overflow hibát, és demonstrálták is kihasználhatóságát például kép megjelenítő programokban, hang, mozgóképf lejátszó plug-inekben (RealMedia), de dokumentum nézegetőben (AcrobatReader) is.

Tekintve, hogy *a túlcserdülésos hibák nagyon gyakoriak* és mint látni fogjuk a fenti elméletben leírt támadási mód *a gyakorlatban sokkal könnyebben kihasználható*, mint ahogyan az első látásra tűnik, a buffer overflow hibákon alapuló támadások jelentik a legnagyobb veszélyt a mai számítástechnikai rendszerekre nézve mind gyakoriság, mind súlyosság tekintetében.

### 3 Printf format string hiba

A standard C library robusztus, könnyen kezelhető kiíró függvénye a `printf()`, illetve annak változatai (`nprintf()`, `sprintf()`, `snprintf()`, `fprintf()`, `vprintf()`, stb.). Ezek előnyös, könnyen használható szolgáltatása, hogy egy formátum string megadásával egyszerűen leírható, hogy a megadott különböző típusú paraméterek a megjelenített szövegben hol és milyen alakban jelenjenek meg.

Abban az esetben azonban, ha a formátum string és az utána átadott paraméterek nem felelnek meg egymásnak, akkor hibás lesz a működés, amely – mint látni fogjuk – támadásra ad lehetőséget. Ezen a téren a leggyakoribb hiba, hogy ha valamilyen külső inputból származó szöveget formátum stringként kezelünk, ugyanis ekkor nem kontrollálható, hogy kerülnek-e be vezérlő karakterek az adott szövegbe.

Egy ilyen tipikus hiba látható az alábbi ábrán:

```
int main(int argc, char *argv[])
{
    printf(argv[1]);          // A paraméterként kapott értéket          // egy
                             az egyben átadja a printf                  //
                             függvénynek                                //
    return 0;
};
```

9. ábra: Példa a `printf` hibás használatára  
(helyesen `printf("%s", input)`-ot kellene írni)

#### 3.1 A hiba következményei

Az input stringben vezérlő karaktereket elhelyezve a támadó olyan "hibás működést" képes előidézni, amely révén információkat tud kiolvasni a program memóriájából, manipulálni képes memóriacímek tartalmát és akár át is tudja venni a vezérlést a támadott gép felett. Például a fenti programot a következő paraméterekkel meghívva:

```
%X %X %X %X %X %X
```

kiírja hexadecimális számrendszerben a stacken tárolt értékeket (amelyek között titkosnak minősülő adatok is lehetnek), illetve a

```
%s %s %s %s %s %s
```

pointerként értelmezi a stacken található értékeket, így nem csak a veremből, hanem e pointerok által mutatott memóriatartományból is egyszerűen lényeges információkhoz juthatunk.

A vezérlő karakterek között azonban a `%n` a legérdekesebb, mert ez nem csupán a megjelenést befolyásolja, hanem *memóriapozíciók felülírására is képes*. Ennek a vezérlő karakternek a funkciója, hogy a paraméterként megadott pointer által mutatott memóriapozícióra kiírja, hogy az adott `printf` végrehajtása során eddig hány karaktert jelent meg a képernyőn. Tekintve, hogy megfelelő string megválasztásával a

képernyőn megjelentetett karakterek számát könnyen befolyásolni lehet, gyakorlatilag megoldható, hogy a %n tetszőleges értéket írjon be a megcímzett memória rekeszbe.

## 3.2 A hiba gyakorisága

A printf hiba elkövetésének gyakorisága annak köszönhető, hogy sajnos a formátum string, illetve annak szokásos használata egy kicsit félrevezető. Az, hogy akár hosszabb szövegek is megadhatóak benne, azt az érzetet kelti, mintha a formátum stringnek kellene tartalmaznia a megjelenített szöveget, holott filozófiája szerint a formátum string csak a megjelenítendő szöveg formátumát írja le. Ebből a félreértésből nem származik probléma egészen addig, amíg a formátum stringként megadott szöveg nem tartalmaz, illetve nem tartalmazhat speciális vezérlő karaktert (%-kal kezdett betűkombinációt).

Fix szövegek esetén ez viszonylag ritkán fordul elő, illetve a tesztelés során kiderül. A probléma akkor jelentkezik, amikor a printf valamilyen változó inputot kap formátum stringként, különösen akkor ha ez az input külső forrásból származik.

Tekintve, hogy a printf függvények kihasználják a C programozási nyelvnek azt a sokak által vitatott lehetőségét, hogy változó számú és eltérő típusú függvény-paramétert engednek meg és mivel a paraméterek száma és típusa pedig a formátum string tartalmától, az abban megadott vezérlőkarakter-kombinációktól függ, *csak a programozón múlik*, hogy a formátum stringnek megfelelő számú és típusú paramétert adjon meg. *Sem a fordítónak, sem futás közben a meghívott függvénynek nincs módja arra, hogy felismerje ha a formátum string és az átadott paraméterek nem felelnek meg egymásnak!* Ilyen esetben a printf függvények működése szinte kiszámíthatatlanná válik. A helyzetet súlyosbítja az is, hogy a bemeneteit a függvény a stacken keresztül kapja, így hiányzó paraméterek esetén az érzékeny veremben fog túlcímezni az eljárást. Vagyis egy apró elírás ki nem derülő hibához vezethet, amely azonban visszaélés elkövetéséhez kihasználható.

## 3.3 A hiba kihasználásának lehetőségei

A printf hiba kihasználásának lehetőségei alapvetően kétfélek. Egyrészt könnyen és egyszerűen kiíratathatók olyan – adott esetben titkos – információk a stackről, amelyekhez egyébként nem szabadna hozzáférni. Ez az *információ-szivárgás* alkalmas lehet például rejtjel kulcsok megszerzéséhez. Másrészt a %n opciót kihasználva gyakorlatilag *tetszőleges memória terület felülírható tetszőleges értékekkel*. Futtatható kódot felülírva pedig elérhető, hogy a támadó által készített utasítás-sorozatot hajtsa végre a számítógép, ezzel átvéve felette a teljes irányítást.

### 3.3.1 Információ-szivárgás

Tekintsük a következő printf hibát tartalmazó példa programot:

```

int main(int argc, char *argv[])
{
    char *secret="Ez itt a titkos kulcs";
    int PINcode=1234;

    printf(argv[1]);

    return 0;
};

```

### 10. ábra: Példa a printf hiba kihasználására

Ekkor a lefordított programot a következő input stringgel meghívva ki tudjuk írni a stacken keresztül elérhető titkos információkat:

```

%08X_%08X_%08X_%08X_%08X_%08X_%08X_%08X_%08X_%08X_%08X_%08X_%08X_%08X_%08X_%08X_%08X_%08X_PINcode=%d_Secret=%s

```

Az eredmény pedig a következő lesz:

```

00132588_00132580_7FFDF000_CCCCCCCC_CCCCCCCC_CCCCCCCC_CCCCCCCC_CCCCCCCC_CCCC
CCCC_CCCCCCCC_CCCCCCCC_CCCCCCCC_CCCCCCCC_CCCCCCCC_CCCCCCCC_CCCCCCCC
_CCCCCCCC_CCCCCCCC_PINcode=1234_Secret=Ez itt a titkos kulcs

```

Vagyis látható, hogy a printf függvényből a stack területen keresztül bármely korábbi hívó függvény lokális változóinak értékét, vagy akár az azok által mutatott memóriacímeket egyszerűen olvasni tudjuk. Tekintve, hogy a programok az általuk feldolgozott adatokat legtöbbször lokális változóiban kezelik, ez az információszivárgási lehetőség rendkívül komoly veszélyeket hordoz magában.

### 3.3.2 Tetszőleges terület felülírása

Az információ szivárgáson felül sokkal komolyabb problémát jelent, hogy egy szinte soha senki által sem használt vezérlő karakter, a %n lehetőséget biztosít gyakorlatilag tetszőleges memóriaterület felülírására. A %n a paraméterként átadott pointer által mutatott memória címre kírja a printf által az adott pontig kiírt karakterek számát.

Ezen lehetőség a kihasználásához két dolgot kell megoldani. Egyrészt kontrollálni kell, hogy hova írjon a %n, másrészt befolyásolni kell tudni, hogy oda mit írjon. Ehhez a felülírni kívánt memóriapozíciókról egy listát kell készíteni és ezt a listát az inputként beadott stringhez hozzá kell fűzni. Ezzel biztosítható az, hogy ez az input string *valahova bekerüljön a stackre*. Kellő számú %x megadásával pedig a lista elhelyezkedése könnyen megállapítható. Ekkor a feladat már csak annak biztosítása, hogy a %n előtt annyi karaktert írjunk ki, hogy a megcímzett bajtokra az általunk kívánt értékek kerüljenek. (Megjegyzés: a %n nem csak egy bajtot ír felül, de ha a következő lépésben felülírjuk a következő bajt értéket, akkor könnyen elérhető, hogy 256 karakterenként túlcserélve mindig a kívánt karakterszámú outputnál tartson a printf és így ezt az értéket írja az általunk listában felsorolt címekre.)



## 4 Feladatok

Az alábbi feladatok elvégzéséhez szükséges ismeretek:

- jártasság az assembly szintű programozásban,
- C programozási nyelv ismerete,
- Visual C++ használata.

Mérési környezet:

- PC munkaállomás
- Windows NT, 2000 vagy XP operációs rendszer
- Visual C++
- Microsoft Developer Network (MSDN)

## 4.1 Buffer overflow feladatok

A gyakorlat célja egy konkrét példán keresztül bemutatni, hogy C programozási nyelven a buffer overflow jellegű hibák milyen komoly veszélyt jelentenek, és mennyire könnyen kihasználható biztonsági rést eredményeznek. Ennek érdekében a következő feladatok lépésről lépésre mutatják be e hiba veszélyeit.

### 4.1.1 BOF1 feladat

Készítsen olyan egyszerű C programot Visual C++ fejlesztői környezetben<sup>1</sup>, amely a bemenetként kapott stringet bemásolja egy fix méretű lokális tömbbe, határérték ellenőrzés nélkül. A hibát tartalmazó függvény az elkészített program első argumentumát kapja meg inputként.

- Demonstrálja tesztekkel illetve dokumentálja, hogy a program normálisan működik olyan rövid inputokon, amely még nem eredményez túlsordulást, majd az input hosszát növelve figyelje meg az okozott hatást. (A bemenő input a Visual C++ Project / Properties / Configuration Properties / Debugging / Command Arguments alatt adható meg.)
- Milyen hosszú bemenetnél kezd el hibásan működni a program? Mikortól száll el?
- Végezzen kísérleteket a fix méretű tömb nagyságának változtatásával és a lokális változók definiálásának sorrendjével is! (Megjegyzés: Érdekes eredményhez minden optimalizálást kapcsoljon ki, vagy definiálja *volatile*-nak a lokális változókat. Ez utóbbi esetben a későbbi feladatok sikeres végrehajtása érdekében a buffernek kell lennie az elsőként definiált lokális változónak. Extra feladat ha valaki más sorrendnél is el tudja érni, hogy a program ne szálljon el az egyéb lokális változók felülírása miatt, még mielőtt a függvényből visszatérne. Ötlet: pontosan olyan értékkel kel felülírni minden memóriacímet, mint ami benne volt.)

### 4.1.2 BOF2 feladat

Az előző feladatban készített program esetében térképezze fel, hogy a buffer kezdetétől számítva hol helyezkedik el a függvény visszatérési címe a stacken, illetve, hogy az input stringben mely karakterek azok, amelyek ezt az értéket felülírják.

#### Első részfeladat

A visszatérési cím meghatározásához a Visual C++ debuggere segítségével vizsgálja meg, hogy a fordítóprogram milyen gépkódú utasításokra fordította le a C programot.

- Jegyzőkönyvéhez csatolja ezt az assembly listát.
- Jelölje be rajta a stack műveleteket.
- Számítsa ki, hogy a visszatérési cím hol helyezkedik el.
- Rajzolja fel a függvény teljes stack térképét (mely lokális változók hol helyezkednek el).

---

<sup>1</sup> Nyissa meg a Dokumentumok\Visual Studio 2005\Projects\bof\_meres\bof\_meres.sln projektet

## Második részfeladat

Határozza meg, hogy az input string mely karakterei írják felül a függvény visszatérési címét. Javaslat: adjon meg eltérő karakterekből álló inputot, például "ABCDEFGHJIJ..."-t.

- Figyelje meg, hogy amikor elszáll a program, az milyen címen történik: Elszállás után indítsa el a debuggert, használja annak disassembly módját (View / Debug Windows / Disassembly ALT-8). Ebből kikövetkeztethető, hogy az input string mely karakterei írták felül a visszatérési címet.
- Milyen értékek találhatóak ekkor a stacken? ESP értékét a View / Debug Windows / Registers ALT - 5 ablakban találja, a stack tartalmát pedig a View / Debug Windows / Memory ALT - 6 ablak segítségével nézheti meg.

### **4.1.3 BOF3 feladat**

A visszatérési cím felülírásával érje el, hogy a program egy olyan függvényt hajtson végre, amely egyébként sohasem hajtódna végre.

- Ehhez írjon egy `BOF_TargetFunction()` függvényt, amely mondjuk kiír valamit a képernyőre, majd kilép a programból.
- Állapítsa meg a `BOF_TargetFunction()` függvény memória címét. (Megjegyzés: ahhoz, hogy ezt a függvényt a fordító mindenképpen beletegy a lefordított programba, vagy valamilyen hivatkozást kell rá tenni vagy ki kell kapcsolni a vonatkozó optimalizálási opciót.) A cím meghatározását végezze el a debugger segítségével is, illetve írassa ki azt a programból.
- Készítsen olyan bemenő input stringet a hibás függvényhez, amely a vezérlést úgy téríti el, hogy a program végrehajtása a `BOF_TargetFunction()` függvényen folytatódjon. (Megjegyzés: ehhez célszerű a `main()` eljárásban programból feltölteni az input stringet, mert command line paraméterként előfordulhat, hogy nem adható meg tetszőleges érték, hiszen az elvben csak olvasható karaktereket tartalmazhat. Az input string feltöltésekor vegye figyelembe a string-fordítást!)
- Demonstrálja, hogy rövid input esetén a program helyesen működik, a speciális input esetén pedig a `BOF_TargetFunction()` függvény fut le!

### **4.1.4 BOF4 feladat**

Cél az input stringben megadott kódsorozat futtatása.

Mint az ismertetőben is kitértünk rá, a buffer overflow hibák "elegáns" kihasználása az, ha a támadó az általa készített programrészlet futtatását képes elérni. Ilyenkor ezt a programrészletet a bemenő adatokba beszúrva, majd a visszatérési címet erre a pozícióra ráirányítva ez megoldható.

Ennek megvalósításához induljon ki az előző feladatnál elkészített programból. A feladat az, hogy az input stringbe egy olyan ugró utasítást szúrjon be, amely a `BOF_TargetFunction()` függvényre fog ugrani.

- Ehhez meg kell határozni, hogy az input string a program memóriájában hol helyezkedik el. Ezt legcélszerűbb úgy meghatározni, hogy az inputként beküldött

szöveget megkeressük a memóriában, amikor a program elszállását követően bejön a debugger. Az input stringben meg kell határozni azt a pontot, ahova a futtatandó kódot besúrjuk. Jelöljük ezt a memória címet `sca`-val – shellcode address. Figyelem, ennek a pontnak a meghatározásánál gondot okozhat, hogy ha valamelyik címben 0 bájt előfordul, akkor ugyanis a string másolás a 0 utáni karaktereket már nem másolja. Így az `sca` pontot olyan helyre kell tenni, ami nem befolyásolja ezt, tipikusan a visszatérési címet felülíró értékek utáni bájtra szokás tenni.

- A kiválasztott `sca`-nak megfelelő helyre, az input stringben be kell szűrni az ugró utasítás (JMP) gépi kódját (0xE9).
- Ki kell számítani az ugró utasítás operandusát. Ezt a 80x86 processzor családnál relatív módon kell megadni, azaz a JMP utasítást követő címhez képest plusz-mínusz hol folytatódjon a vezérlés. Tehát ez az érték a `BOF_TargetFunction()` függvény címe (jelöljük `btf`-fel) mínusz a JMP utáni utasítás címe (`sca+5`) lesz. Az így meghatározott értéket (`btf-sca-5`) kell az input stringben elhelyezni. (Figyeljünk oda rá, hogy a 80x86-os processzorok a több bájtból álló számokat fordított sorrendben tárolják, azaz az alacsonyabb memória címen helyezkedik el az alacsonyabb helyiértékű bájtja egy nagyobb számnak, illetve vegyük figyelembe a sting-fordítást is.)

#### 4.1.5 BOF5 feladat

Opcionális feladat. Külön otthoni felkészülést igényel.

Az előzőekben láttuk, hogy egy olyan példa értékű ugró utasítást tudunk végrehajtani így, amely az input adatokban található. Belátható, hogy egy támadó ennél komplexebb utasítás sorozatot is futtathat ilyen módon. Tekintve, hogy ez a támadási mód évek óta ismert és gyakran kihasznált, számos olyan "konzerv" kódsorozat (úgynevezett shellcode-ot) fejlesztettek már ki, amelyek azt a célt szolgálják, hogy ha egy támadó el tudja érni, hogy a vezérlés erre a kódsorozatra kerül, akkor az átveszi az irányítást az adott gép felett. Ezek a shellcode-ok az Interneten széles körben elérhetőek, így sajnos a buffer overflow jellegű hibákat azok is ki tudják használni, akik esetleg a mélyebb gépi kódú, rendszerszintű programozáshoz nem értenek.

Feladat: keressen az Interneten ilyen shellcode-okat (írja be tetszőleges keresőbe a "buffer overflow shellcode" fogalmakat). Illessze be azokat a fenti példa programokba és érje el, hogy ilyen módon átvegye az irányítást a gép felett.

## 4.2 *printf* format string feladatok

A gyakorlat célja egy konkrét példán keresztül bemutatni, hogy C programozási nyelven a *printf* format string jellegű hibák milyen komoly veszélyt jelentenek, és mennyire könnyen kihasználható biztonsági rést eredményeznek. Ennek érdekében a következő feladatok lépésről lépésre mutatják be e hiba veszélyeit.

### 4.2.1 PFS1 feladat

Készítsen olyan egyszerű C programot Visual C++ fejlesztői környezetben, amely a bemenetként kapott stringet a *printf* függvény első paramétereként írja ki. A hibát tartalmazó függvény az elkészített program első argumentumát kapja meg inputként.

- Demonstrálja tesztekkel illetve dokumentálja, hogy a program normálisan működik olyan inputokon, amelyek nem tartalmaznak % karaktert, majd különböző vezérlő karakterek megadásával (tipikusan %08X, %S) figyelje meg, hogy milyen értékeket ír ki a program. (A bemenő input a Visual C++ Project / Settings... ALT+F7 / Debug / Program arguments alatt adható meg.)

### 4.2.2 PFS2 feladat

Az előző feladatban készített program esetében térképezze fel, hogy a stacken hol található még meg az inputként megadott string (ha kell, akkor ehhez a `main` eljárásban az `argv[1]`-et másolja át egy lokális karakter tömbbe).

- A kinyert adatok alapján rajzolja fel a függvény teljes stack térképét (mely lokális változók hol helyezkednek el).
- A %n vezérlő karakter segítségével írjon felül egy lokális változót a `main` eljárásban. Írassa ki a manipulált változó értékét a *printf* hibát tartalmazó függvény meghívása előtt és után is. (Megjegyzés: ehhez az input stringben el kell helyezni a felülírni kívánt változó címét, majd addig kell %X kiíratásokat ismételni, amíg ezt a címet a stacken nem éri el a *printf*.)