Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Műszaki Informatikai Szak

BIZTONSÁGI SZEMPONTBÓL VESZÉLYES PROGRAMOZÓI HIBÁK ÉS AZ ELLENÜK VALÓ HATÉKONY VÉDEKEZÉS FORTÉLYAI

Adatbiztonság c. tárgy előadás jegyzete

Árendás Csaba

BUDAPEST, 2005

Tartalomjegyzék

1. Bevezetés

A programfejlesztés mai technikája mellett a legtöbb szoftverben rendkívül sok olyan programozási hiba marad, amelyek jogosulatlan visszaélésekre adhatnak módot, alapjaiban ásva alá ezzel a rendszer védelmét, amivel sokszor a teljes rendszer is kiszolgáltatottá válik a támadásoknak. A probléma fontosságát és a veszély nagyságát csak növeli, hogy adott esetben a támadó számára egyetlen hiba megtalálása és kihasználása elegendő a védelmi eszközök megkerüléséhez és a számítógép feletti teljes irányítás átvételéhez. Mivel ezek a hibák rendkívül komoly veszélyt jelentenek a biztonságra, az ellenük való védekezés alapvető fontosságú. Sajnos a közhiedelemben az él, hogy ezen hibák ellen védekezni szinte reménytelen – mondván, hogy az összes hibát megtalálni és kivédeni nem lehet és ezért sokszor a gyakorlatban nem is tesznek semmit - holott ugyanúgy mint minden más veszélyforrás esetén, a biztonsági szempontból kritikus hibákon belül az egyes típushibák ellen is lehet és ezáltal célszerű is specifikus védelmet alkalmazni.

Az egyik legismertebb, ebbe a családba tartózó hiba a buffer overflow, ami hagyományos programozási nyelvekben (C, C++, Pascal) egy gyakran elkövetett hiba, amikor 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. Sajnos a gyakorlati életben egy támadó meglepően egyszerű módszerekkel úgy manipulálhatja ezt a nem kívánt helyzetet, hogy az általa készített tetszőleges programkód lefuttatását is el tudja érni, aminek eredménye képpen képes átvenni a teljes irányítást a számítógép felett. A puffer túlcsordulásos hiba sajnos csak egy a sok közül, a sort folytathatjuk tovább több, ugyanebbe a kategóriába sorolható hibával: heap-, integer-, printf-, unicode bug, array indexing error, stb.

Az ilyen jellegű biztonsági hibák már régóta ismertek, ennek megfelelően számos védekezési módszer született ellenük. Lehetőség van olyan megoldások alkalmazására , amelyek megelőzik, hogy egy ilyen típusú hiba a fejlesztés során bekövetkezzen, illetve igyekszenek elkerülni azon szituációkat amikor ezeket a típushibákat a programozók el szokták követni, ezzel csökkentve a biztonsági lyukak kialakulásának lehetőségét (megelőző védelem). Az esetlegesen elkövetett hibákat fordítási és/vagy futásidőben felismerő megoldásokra is több lehetőség van (felismerő

védelmek). Valamint, ha egy hiba mindezek ellenére előfordul és felfedezetlen marad még akkor is több lehetőség van arra, hogy megakadályozzuk a hiba biztonsági szempontból való kihasználását azt, hogy az adott hibát támadáshoz valaki felhasználja (kihasználás megnehezítése).

2. Biztonsági szempontból veszélyes programozási hibák

A programokat – de legalább a programkód egy jelentős részét – emberek írják, ezért természetesnek mondható, hogy a kódban sok hiba fordul elő. Ezeket a hibákat többnyire figyelmetlenség okozza, de gyakran előfordul, hogy a programozó hozzá nem értéséből adódnak. Jelen jegyzet a programozói hibák közül kifejezetten a biztonsági szempontból veszélyes programozási hibákkal foglalkozik, célja, hogy egy átfogó betekintést nyújtson a témakörbe, nagy hangsúlyt fektetve a védekezési lehetőségekre. Ezen alcsoportba sorolt hibák a programozási hibák egy igen fontos és veszélyes csoportja, ugyanis nem csak a program működésére lehetnek negatív hatással – a biztonsági hibák a normál működés során ugyanis legtöbb esetben nem jelentkeznek –, hanem károkozásra, visszaélésekre is lehetőséget adnak. A biztonsági hibák felkutatása éppen ezért kulcsfontosságú.

A fejezet további részében a teljesség igénye nélkül ismertetem a leggyakrabban előforduló hibákat, mind elvi szinten, mind pedig konkrét példákkal szemléltetve. A példák alapvetően PC-s platformra jellemzőek, de nagy részük minimális változtatások kivételével architektúrától független.

2.1. Veremtúlcsordulás (stack overflow)

A veremtúlcsordulás (stack overflow) [2],[3] (más néven buffer overflow) a hagyományos programozási nyelvekben (C, C++, Pascal) egy gyakran elkövetett [1] hiba. A rendellenes működés abból ered, hogy 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 stack-en tárolódik, amiből következően a tömb határán túlírva lehetőség nyílik egy függvény visszatérési címének felülírására. Ennek segítségével pedig egy támadó elérheti, hogy a program futtatása az általa meghatározott címen – tipikusan ahol az általa elhelyezett rosszindulatú kódsorozat található – folytatódjon.

A hibát jól szemlélteti az 1. ábrán látható hibásan megírt program. A mintaprogram az első argumentumaként kapott karaktersorozat kezdőcímét átadja a `hibas_fuggveny`-nek, majd a függvényen belül az `strcpy` könyvtári függvény segítségével átmásoljuk a lokálisan deklarált `buffer` nevű tömbbe és kiírtatjuk a `buffer` tömb tartalmát. A tömb mellet foglalunk helyet még egész változóknak is, szemléltetés céljából, műveletet nem végzünk velük. Annak ellenére, hogy ez egy rendkívül egyszerű program, mégis komoly hiba bújik meg az utasítások mögött.

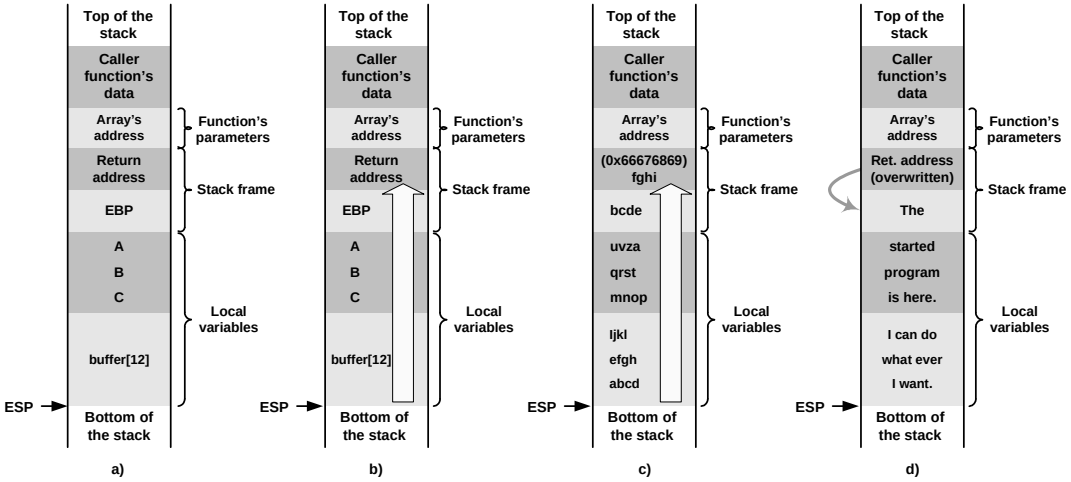
```
#include "string.h"
void hibas_fuggveny(char* tomb)
{
      int a=1;
      int b=2;
      int c=3;
      char buffer[10];

      strcpy(buffer,tomb);
      printf("%s",buffer);
}
int main(int argc, char* argv[])
{
      int i=4;
      int j=5;
      int k=6;

      hibas_fuggveny(argv[1]);

      return 0;
}
```

1. ábra: Buffer overflow hibát tartalmazó függvény

A hiba felszínre hozásához adjunk változó hosszúságú argumentumot a programnak! Rövid kísérletezés után kideríthetjük, hogy nem túl hosszú (1-24 karakteres) bemenet esetén minden a várakozásainknak megfelelően működik. Ettől hosszabb inputnál viszont a program hibaüzenettel elszáll. A probléma forrása természetesen a lokálisan deklarált 10 elemű buffer, amit túl hozzú bemenet estén túlírunk. A programozó által beépített ellenőrzés híján az `strcpy` függvény addig másol, míg a string végét jelölő 0-s értékű bájtot el nem éri. Semmiféle beépített ellenőrző rutin nem vizsgálja, hogy van-e elegendő hely a cél területen. Ennek

köszönhetően olyan memóriacímeket is felülírunk, amelyek értéke a program zavartalan futásához nélkülözhetetlen.

Egy függvény meghívásakor a C/C++, de a legtöbb programozási nyelv estén is a stack-re kerülnek a hívó függvény adatai, a függvény paraméterei, a visszatérési cím és egy regisztermentés (2/a ábra). Az EBP mentést követően folytonosan foglalnak helyet a lokális változók, az a, b és c egész értékek (az integer típus 32 bites, azaz 4 bájtos). Ezt követően allokálódik hely a 10 bájtos karaktertömbnek (ez egészen pontosan 12 bájt lesz, a memória 4 bájtonkénti foglalhatósága miatt). Az itt leírtak természetesen sokban függenek a használt fordítótól és architektúrától (32 vagy 64 bites processzor, az integer is csak általában 4 bájtos és a karakter is lehet 2 bájtos, mint például unicode esetén).



2. ábra: a) A stack tartalma a függvény végrehajtása közben, normál méretű bemenettel. b) A stacken tárolt buffer túlcsordul, felülírva a viszatérési címet is. c) Buffer overflow hiba következik be, a bemeneti input: „abcdefghijklmnopqrstuvzabcdeghi”. d) A visszatérési címet a támadó úgy módosította, hogy a bufferbe betöltött programkód induljon el.

A hiba tehát abból fakad, hogy a tömböt túlírva a számára lefoglalt memória tartomány feletti értékek is módosulnak, szélsőséges esetben egészen a visszatérési címig felülírjuk a memóriát (2/b ábra). Egy függvény befejeződésével a program futásának a függvényhívás utáni sorral kell folytatódnia, amit a processzor úgy old meg, hogy a függvényhíváskor a stack-re mentett visszatérési címet beolvassa a stack-ről és ott található címről folytatja a végrehajtást. Ha a visszatérési címet felülírtuk, ezzel befolyásoljuk a program további futását. A 2/c ábra jól szemlélteti azt az esetet, amikor a program argumentumának az ”`abcdefghijklmnopqrstuvzabcdefghi`” karaktersorozatot adjuk, és ez hogyan fogja felülírni a stack kritikus memória pozícióit. Jól látható és kiszámolható, hogy a 29., 30., 31., és a 32. karakter fogja felülírni a visszatérési címet. Ezen karakterek hexadecimális kódja lesz az a memóriacím, ahol a

program folytatódni fog. Véletlen túlírásnál, az esetek nagy részében ezen a címen nincs végrehajtható utasítás, így a program elszáll. (Különös segítség a támadó számára, hogy még a forráskód ismerete nélkül is könnyedén meghatározható, hogy egy ilyen input stringnek hányadik karakterei írják felül a visszatérési címet, hiszen a fenti input érték hatására a támadott program elszáll és az oprendszer kiírja az elszállás memória pozícióját (esetünkben 81807F7E), ami világosan mutatja a kívánt pozíciót. Ez az egyszerű trükk is azt mutatja, hogy egy ilyen hibát kihasználni könnyű, ha a támadó tudja mit keres.)

Ez az érték viszont célzottan is módosítható, tudatosan beleavatkozva a program normális futásába. Amint azt a példaprogram is mutatja (3. ábra), akár a sohasem meghívott „bármi” nevű függvényt is elindíthatjuk. Mindössze az elindítani kívánt függvény kezdőcímét kell a visszatérési cím helyébe írni.

```
#include "string.h"
void barmi()
{
      printf("Azt teszek, amit akarok!");
}

void hibas_fuggveny(char* tomb)
{
      int a=1;
      int b=2;
      int c=3;
      char buffer[10];
      strcpy(buffer,tomb);
      printf("%s",buffer);
}

int main(int argc, char* argv[])
{
      int i=4;
      int j=5;
      int k=6;
      hibas_fuggveny(argv[1]);
      return 0;
}
```

3. ábra: Buffer overflow hibát kihasználva elindítható a soha meg nem hívott „bármi” függvény

Exploitokban (támadó programoknál) a tipikus avagy „elegáns” kihasználási trükk, hogy a buffert töltjük fel az elindítani kívánt támadó programmal, majd a visszatérési címet úgy módosítjuk, hogy az ily módon beinjektált programunk induljon el (2/d ábra).

2.2. Egésztúlcsordulás (integer overflow) / Előjeles hiba (signedness bug)

A stack overflow (veremtúlcsordulás) kimagasló bekövetkezési aránya következtében a fejlesztők elkezdték komolyabban kezelni a hibát, bár koránt sem körültekintően. Ennek eredménye, hogy bár jónak tűnő védelmet építenek a programba, addig egy más típusú, de hasonlóan gyakran elkövetett hiba miatt továbbra is támadható marad a program.

Az integer overflow (egész túlcsordulás) [2],[4] és a signedness bug (előjeles hiba) programozási hibák, amelyek speciális esetekben lehetőséget teremthetnek a rosszindulatú felhasználónak, támadónak, hogy befolyásolják a program végrehajtási útvonalát. Mindkét típushiba kihasználásának alapja, hogy nem várt változó értékeket adunk a programnak. Ezen hibák önmagukban nem olyan könnyen kihasználhatóak, mint a stack overflow vagy a később ismertetésre kerülő format string bug, hisz önmagukban nem alkalmasak memóriaterületek átírására, de segítségükkel könnyen előidézhetőek más típus hibák, amelyek már kihasználhatóak (például stack és heap overflow).

Egy integer típusú változó fix méretű, így nyilvánvalóan létezik egy legnagyobb érték, amit tárolni tud. Architektúrától és fordítótól függ, hogy ez az érték mekkora (16, 32, 64 bit esetén rendre 65536, 4294967296, 18446744073709551616). Amikor egy nagyobb értéket akarunk eltárolni benne, az túlcsordul, ezt nevezzük integer overflow-nak.

Egy integer túlcsordulást nem lehet olyan könnyen detektálni (létezik természetesen integer overflow hardveres interrupt, de erre nem lehet alapozni, mert majd minden előjeles aritmetikai művelet kihasználja az integer overflow-t, ahogy az a 4. ábrán látható) miután bekövetkezett (ellentétben más hibákkal, mint buffer overflow, ahol valószínűleg elszáll a program).

```
-700        +     800   =     100
0xfffffd44  +     0x320 =     0x100000064
```

4. ábra: Az integer overflow hardveres interrupt általában nem alkalmas a hiba bekövetkezésének detektálására, mert a legtöbb előjeles aritmetikai művelet kihasználja ezt a kalkulációkhoz.

Tehát a feldolgozó programrész nem tudja és nem is tudhatja, hogy a kapott érték helyes-e. Ez nagyon veszélyes lehet, ha a túlcsordulást tartalmazó kalkuláció eredményét használjuk fel valamilyen kritikus művelethez, mint például buffer foglaláshoz vagy tömb címzéshez. Természetesen a legtöbb esetben ez nem

kihasználható, mert a memória közvetlenül nem módosul, azonban könnyen más típushibához vezethet, esetünkben például buffer overflow-hoz.

2.2.1. Méretbeli egész túlcsordulás (widthness integer overflow)

Különböző okból következhet be integer overflow hiba, legegyszerűbb a bekövetkezés szerint csoportosítani őket. Ezen csoportosítás első fajtája a méretbeli egész túlcsordulás (widthness integer overflow) [4]. Ez a hiba annak az eredménye, ha egy nagyobb méretű változót szeretnénk eltárolni egy kisebb területen, amely azt így nem képes befogadni, ami miatt csonkolódni fog. Ilyen hibát tartalmazó programkód látható a 5. ábrán.

```
/* ex1.c - loss of precision */
    #include <stdio.h>

    int main(void)
    {
        int l;
        short s;
        char c;

        l = 0xdeadbeef;
        s = l;
        c = l;

        printf("l = 0x%x (%d bits)\n", l, sizeof(l) * 8);
        printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
        printf("c = 0x%x (%d bits)\n", c, sizeof(c) * 8);

        return 0;
    }
```

5. ábra: Widthness integer overflow

A 6. ábrán látható a program kimenete. A kezdeti 32 bites értéket rendre átmásoljuk egy 16, illetve egy 8 bites változóba, amikor is mindkét esetben csonkolás történik.

```
./ex1
    l = 0xdeadbeef (32 bits)
    s = 0xffffbeef (16 bits)
    c = 0xffffffef (8 bits)
```

6. ábra: Widthness integer overflow program kimenete

A 7. ábrán látható program egy konkrét példát mutat arra, hogyan okozhat komoly problémát egy integer overflow típusú hiba. A hiba önmagában nem jelentene nagy veszélyt, de ha a túlcsordult, hibás értéket felhasználjuk az már adott esetben kihasználható hibához vezet.

```
/* width1.c - exploiting a trivial widthness bug */
    #include <stdio.h>
    #include <string.h>
    int main(int argc, char *argv[])
    {
        unsigned short s;
        int i;
        char buf[80];

        if(argc < 3)
        {
            return -1;
        }

        i = atoi(argv[1]);
        s = i;

        if(s >= 80)
        {           /* [w1] */
            printf("Oh no you don't!\n");
            return -1;
        }

        printf("s = %d\n", s);

        memcpy(buf, argv[2], i);
        buf[i] = '\0';
        printf("%s\n", buf);
        return 0;
    }
```

7. ábra: Widthness bug hiba kihasználása

A program egy rendkívül egyszerű működést valósít meg. Lényege, hogy a paraméterben megkapott string-et átmásolja a lokálisan deklarált, 80 karakter hosszúságú bufferbe. Így természetesen felmerül a buffer overflow lehetősége, túl hosszú input esetén. Hogy megelőzzük a buffer túlírás lehetőségét, a második paraméterben meg kell adni a string hosszát is. Amennyiben 80 karakternél nagyobb hossz értéket adunk meg, a program hibaüzenettel leáll. Hossz mezőnek 80 alattit adva, de hosszabb string-et téve az inputra sem történik hiba, mert csak a paraméterben megadott hosszig történik a string átmásolása a bufferbe. Hagyományos jellegű (buffer overflow hibát kereső) próbálkozás sikertelenül zárul. Ennek ellenére, integer overflow hibát kihasználva buffer overflow hibát idézhetünk elő, a következőkben részletezett módon.

A 8. ábrán láthatóak a különböző bemenettel tett próbálkozások. Az utolsó kísérlet esetén a program elszáll. Az oka, hogy a hosszt egy `i` integerben (32 bit) és egy `s` unsigned short integerben (16 bit) is tároljuk. A túlírás figyeléshez az értéket az `s` változóból vesszük, míg a másoló `for` ciklusban az `i` változó értékét használjuk fel. A harmadik mintabemenet esetén 65536-ot adtunk meg a hossznak, ami első ránézésre

nyilvánvalóan megbukna, de mivel egy 16 bites változóba másoljuk az értékét, az integer változó túlcsordul, így az értéke 0 lesz. Mivel a másolásnál nem ezt, hanem a 32 bites `i` integer értékével dolgozunk, 65536 bájtot fogunk belemásolni a 80 bájtos tömbbe, vagyis egy klasszikus buffer overflow hibához jutottunk az integer overflow hiba segítségével.

```
./width1 5 hello
    s = 5
    hello
./width1 80 hello
    Oh no you don't!
./width1 65536 hello
    s = 0
Segmentation fault (core dumped)
```

8. ábra: Widthness bug-ot tartalmazó program, különböző bemenetekkel

2.2.2. Aritmetikai túlcsordulás (aritmetical overflow)

Ahogy a widthness hiba esetén már láthattuk, ha megkísérelünk egy egész típust egy kisebb méretű egészben tárolni, az érték csonkolódik. Ha a tárolt érték egy aritmetikai művelet eredménye, aritmetikai túlcsordulásról beszélünk [4]. A program minden olyan további része, ami használja ezt az értéket, hibásan fog tovább dolgozni, hisz hamis bemenetet kap.

```
// ex2.c forráskódja:>
#include <stdio.h>
    int main(void){
            unsigned int num = 0xffffffff;

            printf("num is %d bits long\n", sizeof(num) * 8);
            printf("num = 0x%x\n", num);
            printf("num + 1 = 0x%x\n", num + 1);

            return 0;
    }
```

9. ábra: Aritmetikai overflow hiba

A 9. és a 10. ábrán jól látszik, hogy egy művelet után az eredmény változó túlcsordul és hamis értéket eredményez.

```
./ex2
    num is 32 bits long
    num = 0xffffffff
    num + 1 = 0x0
```

10. ábra: Aritmetikai overflow hibát tartalmazó program kimenete

A túlcsordulás mellet a másik nagyon gyakori hibalehetőség az előjelváltás. Egy integer alapesetben előjeles, így integer túlcsordulás előjelváltást okozhat (11. ábra, 12. ábra).

```
/* ex3.c - change of signedness */
    #include <stdio.h>
    int main(void)
    {
            int l;
            l = 0x7fffffff;

            printf("l = %d (0x%x)\n", l, l);
            printf("l + 1 = %d (0x%x)\n", l + 1 , l + 1);

            return 0;
    }
```

11. ábra: Előjelváltás integer overflow hatására

```
nova:signed {38} ./ex3
    l = 2147483647 (0x7fffffff)
    l + 1 = -2147483648 (0x80000000)
```

12. ábra: Előjelváltás integer overflow hatására C program kimenete

Természetesen szinte bármilyen aritmetikai művelet okozhat integer overflow-t, nem csak a mintaprogramban alkalmazott összeadás. Gyakorlatilag bármilyen operandus, ami módosítja a változó értékét, okozhat ilyet. A kivonás művelet egy kicsit más, hisz integer underflow-t okoz. Ilyen módon egy összeadó operandust tartalmazó programsort rá tudunk venni, hogy a művelet eredménye olyan legyen mintha kivonást végeztünk volna el (hasonlóan igaz ez a többi műveletre is).

Az aritmetikai túlcsordulásos hiba könnyedén kihasználható, ha a kalkuláció folyamán azt számoljuk ki, hogy mekkora buffert allokáljunk. Legtöbbször egy programnak allokálnia kell helyet elemek egy tömbjének, ezért a `malloc`, `calloc` függvényt használja. A szükséges terület kikalkulálásához összeszorozza az elemek számát egy elem méretével. Ahogyan korábban már láthattuk, ha módunk van befolyásolni a két operandus valamelyikét (az elemek számát, vagy az elem méretét), valószínűleg lehetőségünk van ál buffer méretet beállítani, ahogyan a 13. ábrán látható.

```
int myfunction(int *array, int len)
    {
        int *myarray, i;

        myarray = malloc(len * sizeof(int));    /* [1] */
        if(myarray == NULL)
        {
            return -1;
        }

        for(i = 0; i < len; i++)                /* [2] */
        {
            myarray[i] = array[i];
        }

        return myarray;
    }
```

13. ábra: Aritmetikai overflow kihasználása

Ez a függvény, bár elsőre teljesen ártatlannak tűnik, komoly hibát rejt, mert semmiféle ellenőrzést nem végez a „len” paraméteren. A `malloc` függvény argumentumában szereplő szorzás segítségével integer overflow hibát idézhetünk elő, tehát gyakorlatilag bármekkora buffert lefoglalhatunk. Egy megfelelő len (hossz) kiválasztásával elérhetjük, hogy a for ciklus túlírjon a lefoglalt buffer területén túl, ezzel heap overflow-t idézhetünk elő, amivel befolyásolni tudjuk a program futásának menetét, illetve idegen kód végrehajtása is elérhető.

```
int catvars(char *buf1, char *buf2, unsigned int len1,
                unsigned int len2)
    {
        char mybuf[256];

        if((len1 + len2) > 256)         /* [1] */
        {
            return -1;
        }

        memcpy(mybuf, buf1, len1);      /* [2] */
        memcpy(mybuf + len1, buf2, len2);

        do_some_stuff(mybuf);
        return 0;
    }
```

14. ábra: Aritmetikai overflow kihasználása #2

A következő mintapéldában (14. ábra), a méretellenőrzés kijátszható, megfelelő `len1` és `len2` értékek megadásával. Kellően nagy érték esetén az integer túlcsordul, kis érték lesz belőle, ami az ellenőrzésen átmegy (ilyen értékek például a `len1 = 0x104` és `len2=0xfffffc`, amikor a két érték összeadódik, túlcsordulás következik be, az új érték 256 lesz. Ez az érék 1-essel jelölt méretellenőrzésen átmegy, majd a 2. jelölésnél a forráskódban már a két összeadandó van használva, így bőven túlírják a buffert), azonban a `memcpy` függvények az eredeti `len1` és `len2` értékekkel dolgoznak, amik így sokkal nagyobb memóriarészt fognak beleírni a bufferbe, mint amekkorát el tudna tárolni, tehát buffer overflow hiba lép fel.

2.2.3. Előjeles hiba (signedness bug)

Előjeles hibákról [4] akkor beszélünk, ha egy előjel nélküli változót előjelesként értelmezünk, illetve ha egy előjeles változót előjel nélküliként. Ez azért történhet meg, mert nincs különbség a tárolási módok között.

Ilyen jellegű hibák rendkívül sokféleképpen előfordulhatnak, de ezek közt a leggyakoribbak a következők:

- előjeles integer használata összehasonlításnál,
- előjeles integer használata aritmetikai műveletnél,
- előjel nélküli integer összehasonlítása előjeles integerrel.

A 15. ábrán látható mintapéldában a probléma az, hogy a `memcpy` függvény egy előjelmentes integert vár a `len` (hossz) változóként, míg az előtte levő méretellenőrzés előjeles integert használ. Egy negatív számot értékül adva a `len` változónak, a feltételvizsgálatot becsapja, így túljutva rajta (a negatív szám mindig kisebb, mit a buffer méret, ami pozitív lesz). Ezt követően a `memcpy` függvény fogja felhasználni ezt az értéket, viszont előjelmentesként fog rá hivatkozni, így pedig egy nagyon nagy számként fogja értelmezni, aminek köszönhetően bőven túlíródik a buffer a lefoglalt területen.

```
int copy_something(char *buf, int len)
    {
        char kbuf[800];

        if(len > sizeof(kbuf))          /* [1] */
        {
            return -1;
        }

        return memcpy(kbuf, buf, len);  /* [2] */
    }
```

15. ábra: Előjelességi hiba

Leggyakrabban ezt a típushibát nehéz támadás céljára kihasználni annak köszönhetően, hogy amikor egy előjeles integerre előjelmentesként hivatkozunk, az egy nagy számot eredményez. Például a -1 hexadecimális szám ábrázolásban **`0XFFFFFFFF`**. Ha előjelmentesként értelmezzük, akkor a 32 biten ábrázolható legnagyobb számot jelenti (4 294 967 295). Vagyis ez, vagy egy ehhez közeli érték adódik át a `memcpy` függvénynek, így egy 4 GB-hoz közeli értéket próbál a célbufferbe belemásolni. Természetesen ez leggyakrabban hibához vezet és a program elszáll, vagy egy hatalmas memóriaterület foglal le a stacken vagy a heapen feleslegesen. Ezt a problémát néha lehetséges kikerülni, ha egy nagyon alacsony értéket adunk meg a forrás címnek.

A következő példaprogram a hiba kihasználására mutat egy lehetséges megoldást. A 16. ábrán látható példa tipikusan hálózai kommunikációs programoknál fordulhat elő, ha a csomagméret információ a csomag része (más szóval egy nem megbízható felhasználótól érkezik, hiszen bárki módosíthatja a hálózaton érkező csomagot). A forráskódban [1]-el jelölt résznél látható összeadás azt számolja ki, hogy a

két csomag együttes mérete nem haladja-e meg a célbuffer méretét. Megfelelő `size1` és `szize2` érték megadásával elérhető, hogy az összeget tároló változó túlcsorduljon, egy negatív számot eredményezve. Például

```
Size1=0x7fffffff
Size2=0x7fffffff
0x7fffffff+0x7fffffff=0xfffffffe(=-2)
```

Ha ez megtörténik, a [2]-vel jelölt méretellenőrzésen túljut a végrehajtás és a buffer túlíródik. Sőt az `out+size1`-nek köszönhetően tetszőleges memóriacím is felülírható, hiszen az egyik címet használhatjuk arra, hogy integer overflow-t idézzünk elő, míg a másikat használhatjuk a precíz célzásra.

```
int get_two_vars(int sock, char *out, int len)
    {
        char buf1[512], buf2[512];
        unsigned int size1, size2;
        int size;

        if(recv(sock, buf1, sizeof(buf1), 0) < 0)
        {
            return -1;
        }
        if(recv(sock, buf2, sizeof(buf2), 0) < 0)
        {
            return -1;
        }

        /* packet begins with length information */
        memcpy(&size1, buf1, sizeof(int));
        memcpy(&size2, buf2, sizeof(int));

        size = size1 + size2;       /* [1] */

        if(size > len)              /* [2] */
        {
            return -1;
        }

        memcpy(out, buf1, size1);
        memcpy(out + size1, buf2, size2);

        return size;
    }
```

16. ábra: Integer overflow hibát okozó előjelességi hiba

Ezek a hibák a gyakorlatban pontosan olyan módon használhatóak ki, mint a hagyományos előjelességi hibák.

3. A támadási eszköztár végtelen lehetőségei

Az előző fejezetben áttekintettem a jelenleg leggyakrabban előforduló, ebbe a családba tartózó leggyakoribb biztonsági hibákat. A következőkben egyrészt minden egyes korábban ismertetett biztonsági hibára mutatok egy konkrét, a valóságban ismert hibát és támadó, kihasználó (úgynevezett exploit) programot. Másrészt a fejezet második részében kronológiai sorrendbe rendezem ezeket a hibákat. Arra szeretnék rávilágítani, hogy rendkívül sok ilyen hiba van és amint egyre egy kielégítő védelem felbukkan, csak idő kérdése, hogy mikor áll elő valaki valamilyen eredeti ötlettel, egy újabb sebezhetőség típussal. A rész másik célja alátámasztani, hogy egy aktív, fontos kutatási területről van szó, amely egyáltalán nem nevezhető megoldott problémának a nagyszámú sikeres támadás miatt. Annak ellenére, hogy sok hatékony védekezési módszer áll rendelkezésre, így is rendkívül nagy a rés, amelyen keresztül kompromittálhatóvá válhat a teljes rendszer.

3.1. Példa támadások, minta exploitok

3.1.1. Verem túlcsordulás - Search and replace ZIP file search buffer overflow

A Search and replace 5.0 egy Windows alá írt segédalkalmazás. Bináris és szöveges állományokban való hatékony keresést/szövegcserét tesz lehetővé minta karaktersorozat, illetőleg reguláis kifejezés alapján. Tömörített (ZIP) állományban történő keresés során a csomagolt állományban lévő fájlnevek feldolgozását végző rutinban buffer overflow hiba van. Ez tehát egy lokálisan kihasználható hiba, vagyis a felhasználót rá kell venni, hogy keressen az exploitot tartalmazó tömörített fájlban. Erre a leggyakoribb példa csatolt fájl küldése e-mailben. Az alább látható exploit program egy `tamado.zip` állományt készít, amelyben keresve a programmal a hiba kihasználhatóvá válik, támadó kód elindítása lehetséges.

Hibát tartalmazó program: Search and replace 5.0

Kihasználhatóság: lokális, hiba leírás [24], exploit [23]

3.1.2. Egész túlcsordulás - Cisco IOS 12.x/11.x HTTP integer overflow remote Exploit

Cisco IOS 12.x/11.x operációs rendszerében egy távolról is kihasználható integer overflow hibát sikerült felfedezni 2003 augusztusában. Az exploit program speciálisan formázott HTTP GET kérésekkel bombázza a cél eszközt. Az integer overflow előidézéséhez a rengeteg kérést kell küldeni, kb. 2GB-ot. A shell kód sikeres alkalmazása után a rendszer nem kér hozzáférési jelszót.

Hibát tartalmazó program: CISCO IOS 12.x/11.x

Kihasználhatóság: távoli

Hiba leírás [25], exploit [25]

3.1.3. Előjeles hiba

Néhány rendszerhívás (`accept`, `getsockname`, `getpeername`, stb.) felhasználása során, a fejlesztők abból a téves feltételezésből indultak ki, hogy a függvényeknek átadott paraméterek pozitív egészek, holott előjelesek alapértelmezésből. Ennek következtében a méretellenőrzés elbukhat, ha negatív szám szerepel bemeneti értékként. Ennek következtében nagy negatív számmal meghívva ezeket a függvényeket, visszatérésnél a kernel terület egy szeletének a másolatát adja vissza, ami információszivárgáshoz vezet (kódolatlan felhasználói jelszavak megszerzése, stb.).

Hibát tartalmazó program: FreeBSD system 4.6.1

Kihasználhatóság: távoli

Hiba leírás [26]

3.2. Biztonsági hibák időrendi felfedezésének áttekintése és jövőbeli kitekintés

- **1960**

  Az első és legrégebbi időpont, amikor a buffer overflow hiba lehetősége felmerül. Itt még sokkal inkább programozási hibáról és hibás működésről van szó, mint támadásról. Ilyen szinten ma is oktatják az egyetemeken a hiba létezését, hogy a tömb túlcímezhető és ennek megakadályozása, az input méretének ellenőrzése a programozó feladata. A tényleges támadási célú kihasználásra és az ötletre még várni kell.

- **1980**

  Az első támadási céllal kihasznált overflow típusú hiba felfedezése. Ettől az időponttól ismeri a szakirodalom a Stack Overflow fogalmát. A veszély tényleges, nagyszámú realizálódásáról ekkor még nem beszélhetünk.

- **1988. november 2.**

Egy önreplikáló Internet Worm indult el az útjára, amely több tízezer számítógépet fertőzött meg, amely akkoriban az Internet 60%-át tette ki. Ez az első és egyben legismertebb, legátfogóbb és a mai napig az Internet arányaiban nézve legsikeresebb támadás.

- **1995**

A hiba mindennapos megjelenésére 1995-ig várni kellett. Az akkoriban sokaknak újnak számító technológia végigsöpört a világ biztonsági laborjain.

- **1996**

1996-ban Alaph One publikálta [3] a „Smashing The Stack For Fun and Profit” címen, amely a legtöbbet hivatkozott publikáció mind a mai napig.

- **1997**

Tömeges megjelenése a különböző publikációknak, támadási hogyanoknak (how-to-knak) és ebből kifolyólag drasztikusan megnő az ilyen típusú hibákat kihasználó támadások száma. Ezzel együtt megjelennek a különböző védekezési módszerek is, úgy mint a non-executable stack, amely első ránézésre megoldja a problémát.

- **1999**

A buffer overflow problémát még csak éppen felismeri a szakma és elkezdődnek az ez irányú, meglehetősen kevés eredményt elérő kutatások, megjelenik egy új sebezhetőség, amely alapjaiban ugyan sokban hasonlít a stack overflow-ra, de mégis teljesen új lehetőségeket rejt magában. Az új sebezhetőség neve: Heap Oveflow. Az első tanulmányok egyikét a w00w00 Security Team publikálta [6] 1999-ben „w00w00 on Heap Overflows” címen. Ez a technológia kicsit bonyolultabb volt, mint a Buffer Overflow, hiszen a különböző memóriacímek futásidőben kerültek meghatározásra, így sok esetben jelentősen megnehezítve kihasználást és megkönnyítve a felderítést.

- **1999 vége**

Egy ELF szerkezeti hibát kihasználva felmerült a „.dtors” overflow lehetősége, amelyről az első tanulmány 2000-ben jelent meg.

- **2000**

A következő áttörést a 2000 végén felfedezett Format String Overflow technika jelentette, amely ismét rengeteg fejtörést okozott a fejlesztőknek.

- **2002**

2002 decemberében a Phrack magazinban publikálták az integer overflow lehetőségét, ami rendkívül érdekes olvasmány. Szignifikánsan látható, hogy a programozók nagy nehezen tanultak a buffer overflow hibák fenyegetéséből és igyekeztek valamilyen input hossz ellenőrzési kódsorozatot beépíteni a kritikus programrész elé. Az integer overflow egyértelmű visszakontrázás, amely jól szemlélteti a kreatív gondolkodás és a támadási lehetőségek sokszínűségét.

- **2004**

Nagy felhajtások közepette megjelenik az NX bit-et (non-executable) támogató processzorok sora. Mind az AMD beépíti az Opteron, Athlon 64 frissen megjelenő processzoraiba, csak úgy mint az Intel is bejelenti, hogy a Transmeta processzor már támogatni fogja. A védekezési módszer lényege, hogy nem végrehajthatóvá tesszük a stacket, így elvi védelmet nyújtva buffer overflow támadások ellen, amikor is a támadó a túlírt buffert rosszindulatú kóddal tölti fel, majd a visszatérési címet úgy módosítja, hogy a frissen beinjektált programkódot indítsa el. A technológiát a Microsoft is nagy erőkkel népszerűsíti, bár érdekes, hogy az ötlet már 1997-ben felmerült és meg is valósították. A megoldás kétség kívül megnehezíti a kihasználást, de a retun-into-libc támadási módszernek köszönhetően [28] az esetek többségében könnyen kikerülhető ez a védelem is.

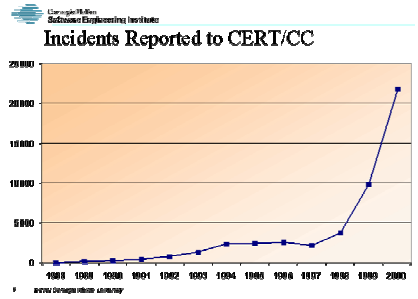
A return into libc megoldás alapötlete rendkívül eredeti. Mivel a stack nem végrehajtható, nem töltjük fel a buffert a támadó kóddal, hanem helyette a visszatérési cím helyére egy rendszerhívás memóriacímét írjuk (tipikusan system()), majd az ezt a 4 bájtot követő helyre tipikusan a „/bin/sh” karaktersorozatot. Amikor a ret utasítást végrehajtja a processzor, a stack-ről próbálja meg beolvasni a függvény argumentumát, ami esetünkben a /bin/sh. Vagyis egy rendszergazdai konzolt indítottunk el, minden satckre helyezett program indítása nélkül. Ez jó példája annak, hogy a buffer túlcsordulásos hibák száma és kihasználási módja határtalan. Sajnos annak ellenére, hogy már számos megoldás létezik, amint azt a következő fejezetben részletezem, még

sok tennivaló akad. Sajnos, vagy éppen szerencsére az emberi találékonyság határtalan, amint azt e hibák megtalálása és kihasználása is jól példázza.

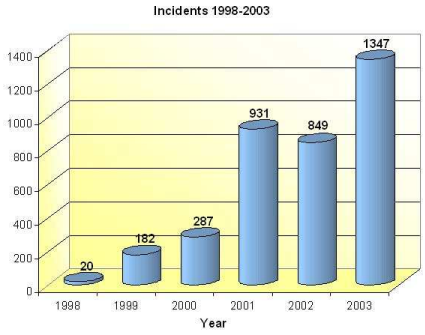
Viszont fontos hangsúlyozni, hogy ugyan rendkívül sokszínű a támadási eszköztár, hatékony, átgondolt védekezéssel nagyon nagy mértékben megnehezíthető a támadó dolga, ezért mindent meg kell tenni annak érdekében, hogy a rendelkezésre álló védekezési módszereket és lehetőségeket a gyakorlatban mindinkább alkalmazzuk.

4. Védelmi megoldások elemzése és összehasonlítása

Számos tanulmány támasztja alá, hogy a biztonsági visszaélések száma az Interneten folyamatosan növekszik. Hiába nő ugrásszerűen a felhasználók száma, az egy gépre vetített támadások száma így is nő.

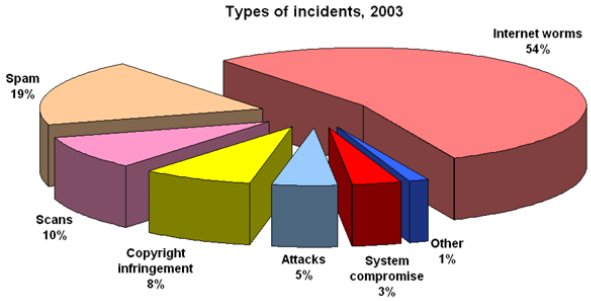


17. ábra: Regisztrált visszaélések az Interneten 1988-2000



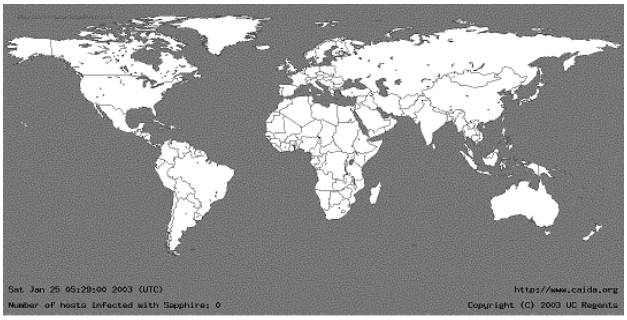
18. ábra: Regisztrált visszaélések az Interneten 1999-2003

Aszerint, hogy a sikeres támadást milyen jellegű sebezhetőség kihasználásával hajtották végre, a hibákat illetőleg a támadásokat különböző csoportokba oszthatjuk:

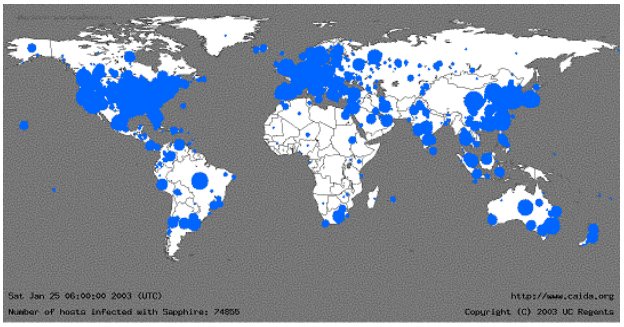


19. ábra: A 2003-as évben lezajlott támadások típusonkénti megoszlása

Ezen támadásokat kimagaslóan vezetik a biztonsági hibák, azon belül is a buffer overflow hiba. A sikeres betörések mintegy 80%-a ilyen jellegű gyengeségekre vezethető vissza. Rendkívül veszélyes egy ilyen hibát tartalmazó program, hiszen látszólag a rendeltetésnek megfelelően működik, holott egy látszólag értelmetlen input betáplálásával a támadó befolyásolhatja a program és ezzel a számítógép működését. Egy ilyen hiba rosszakaratú megtalálója készíthet egy olyan célzott támadó programot, ami a sebezhetőséget kihasználva a célgépre be tud hatolni. Sajnos rendkívül sok ilyen célprogram forog közgézen, ezért nem kell kimagasló szakértelem ilyen eszköz megalkotásához, elegendő csupán letölteni a megfelelő helyről. A másik manapság nagyon divatos módszer, hogy a megtalált hibát vírusterjedéshez használják fel. A terjedéshez ilyen hibát kihasználó vírust féregnek (worm-nek) nevezünk. Egy ilyen ismeretlen hibát először kihasználó vírus rettenetes sebességgel képes elterjedni az Interneten. A leggyorsabban dolgozó víruskereső gyártó cégeknek legkevesebb 4 óra szükséges a vírus analizálásához. Ez sajnos bőven elég a vírusnak a teljes Internet megfertőzéséhez. Jó példa erre a közelmúltban megjelent MS-SQL worm, amely szintén egy ilyen sebezhetőséget használt ki.



20. ábra: 2003. január 25. 5:29:00 - A támadás megindulása előtt



21. ábra: 2003. január 25. 6:00:00 – 30 perccel a támadás megindítása után már mindenütt jelen van az MS-SQL worm

Nagyon fontos ezért tudatosítani, hogy mindent meg kell tenni ezen hibák eliminálásáért. Sajnos közel sem olyan mértékű a védekezés, mint amit a technika

jelenlegi szintje biztosítani tudna. Az ilyen hibák strukturált, szervezett kiszűrése költséges a fejlesztés során, így hajlamosak a nagy szoftverfejlesztő cégek mellőzni a problémát, hisz a hibából eredő károkat nem ők viselik, hanem az szinte egészben csak a végfelhasználónál csapódik le.

Az alábbiakban minden egyes korábban ismertetett típushibára felsorolok specifikus védelmeket. A konkrét hibatípus-védekezés párosítások felsorolása előtt külön áttekintem az általánosan alkalmazható módszereket, mint például a patching, a behatolás detektálás, a vírusirtók, illetve a hozzáférés-szabályozás is. Ezek nem sorolhatók be specifikusan egy hibatípus alá, mert gyakorlatilag bármely hiba javításánál, észlelésénél használhatóak.

4.1. Foltozás (patching)

A foltozás, avagy gyorsjavítások (hotfix) telepítése a leggyakrabban alkalmazott védekezési módszer. Nem csak biztonsági szempontból fontos, hanem funkcionalitásbeli hibák korrigálására is alkalmazzák. A patching lényege, hogy ha a szoftvergyártó hibát talál a kiadott szoftverében (belső munkatárs vagy külső szakember közreműködésével), akkor gyorsjavítást ad ki, amely az érintett rendszeren telepítve lecseréli a hibás szoftverkomponenst a javítottra. Igazából nem kifejezett védelemről van szó, hanem csak foltozgatásról, ennek ellenére manapság ez a leggyakoribb védekezési mód.

4.2. Behatolás érzékelő rendszerek és vírusirtók

A behatolás érzékelő rendszerek (IDS: Intrusion Detection System) a hálózati forgalmat figyelik folyamatosan és a rendelkezésére álló adatbázis segítségével értelmezik a hálózati forgalmat. Gyakorlatilag víruskeresőnek is tekinthetők: a behatolás érzékelése és az esetleges ellenintézkedések foganatosítása a hangsúlyos. Ismert támadási kódminták illetve anomáliák felismerésével képes a legtöbb biztonsági hibát felismerni.

Konkrét megoldások:

SNORT (Open Source Intrusion Detection System)

Forgalomban lévő víruskeresők

4.3. Szabályrendszer alapú hozzáférés-szabályozás

A szabályrendszer alapú hozzáférés-szabályozás a védekezések csoportosítása szempontjából inkább a hozzáférés védelemhez tartozik. Itt nem azt célozzuk meg, hogy megpróbáljuk megakadályozni a biztonsági hiba kihasználását, hanem a legrosszabb esetből indulunk ki, vagyis feltételezzük, hogy a támadónak sikerült áthatolni a védelmi rendszereinken és rendszergazdai, root jogosultságot szerzett a gépünkön. Ez a keretrendszer a rendszergazda jogainak korlátozására használható, például megoldható, olyan naplóállomány létrehozása, amelyhez csak hozzáírni lehet, törölni nem.

Az ilyen jellegű rendszerek tehát egy nagyon fejlett hozzáférésvédelem-szabályozást valósítanak meg, és rendkívül hatékony helyes beállítások esetén. Gyakorlatilag elérhető, hogy bármilyen programnak egyedi hozzáférés szabályozást adjunk, elkülönítőbe (jailbe) is helyezhetjük, aminek következtében csak az engedélyezett erőforrásokhoz férhet hozzá a program. Így, ha egy sebezhetőséget ki is használ a támadó, azzal csak az adott program jogosultságait tudja megszerezni.

Konkrét megoldások:

NSA Security Enchanced Linux [7]

RSBAC Rule Set Based Access Control [20]

4.4. Specifikus típushibák elleni védekezés

A továbbiakban minden egyes a ismertetett típushiba elleni célzott védelmek felsorolása és értékelése következik. A védekezési módszereket a védekezés elve szerint csoportosítva ismertetem. Ezek a csoportok a következők: megelőzés, felismerés, kihasználás megakadályozása.

Prevenció: Olyan eszközöket, megoldásokat alkalmazunk, aminek következtében az esetlegesen biztonsági lyukakat eredményező hiba előfordulását igyekszünk megakadályozni. Ebbe a kategóriába sorolhatóak legtipikusabban a programnyelvi eszközök és a fejlesztés során alkalmazott tesztelő programok.

Detekció: A kész, adott esetben eladott szoftverben való hibakeresés. Ez történhet futás közbeni ellenőrzéssel, vagy célirányos teszteléssel.

Kihasználás megakadályozása: Ez esetben azt akarjuk elérni, hogy egy fel nem fedezett hiba ne hozzon létre kihasználható biztonsági lyukat. Ez úgy érhető el, hogy a hiba támadásra való kihasználását megnehezítjük, jó esetben ellehetetlenítjük.

4.4.1. Veremtúlcsordulás (stack overflow)

A leggyakrabban elkövetett és egyben legrégebbi programozói hiba. Ennek megfelelően számos védelem áll rendelkezésre a kiküszöbölésére (sajnos a gyakorlatban ennek ellenére ezeket a védelmeket csak nagyon ritkán alkalmazzák).

4.4.1.1. Megelőzés

Az emberi tényezőn túl a buffer overflow hibák másik forrása maga a használt programozási nyelv, vagyis az, hogy a nyelv lehetőséget ad-e ilyen jellegű hibák elkövetésére vagy eleve kizárja azt. Programnyelvi eszközökkel a következő lehetőségek merülnek fel megelőzésre, azaz annak megakadályozására, hogy a hiba egyáltalán előfordulhasson.

4.4.1.1.1 Érzéketlen nyelvek

A szoftverfejlesztés során használhatunk olyan programnyelveket, amelyekben eleve fel sem merül a buffer overflow hiba lehetősége. Ezek tipikusan nem C alapú, hanem magas szintű nyelvek. Hátrányuk, hogy az áttérés új programnyelvre rendkívül költséges lehet, és bizonyos esetekben nem is lehetséges; például ezek a nyelvek alacsony szintű hardverhozzáférést általában nem biztosítanak és teljesítményben is jócskán elmaradnak a C-től.

Konkrét megoldások:

Java [8]

Perl [9]

C# .NET [8]

4.4.1.1.2 C alapú nyelvek

A hiba kiküszöbölésére módosíthatjuk az alapnyelvünket, a C-t. Ez az átállást nagyban megkönnyíti, azonban az új nyelv gyakran bonyolult új kódolási technikát igényel.

Konkrét megoldások:

Cyclone: A Cyclone [12] elnevezésű C változatot eleve a buffer overflow hibát zem előtt tartva fejlesztették ki. A megoldás sokkal szigorúbb pointer-kezelést tartalmaz és komoly hátránya, hogy az objektum orientált programozást nem támogatja.

4.4.1.1.3 Biztonságos bufferek

Használhatunk saját biztonságos buffer implementációt is: ilyenkor az osztály automatikus méretellenőrzést, szükség esetén csonkítást vagy átméretezést, valamint kivételkezelést végez. A megoldás hátránya, hogy rontja a teljesítményt (különösen a változó méretű bufferek memória-újrafoglalásai miatt). Egy ilyen lehetséges megvalósítás a C++-ból ismert CString.

Konkrét megoldások:

CString

4.4.1.2. Felismerés

Ha a hibát nem tudjuk teljesen kiküszöbölni, megfelelő eszközökkel még észlelhetjük a hiba keletkezését, illetve lehetséges hiba kihasználására irányuló próbálkozásokat, így megállítható egy esetleges behatolási kísérlet.

4.4.1.2.1 SNORT: Open Source Intruder Detection System

A snort [16] egy IDS rendszer (Intrusion Detection System – Behatolás érzékelő rendszer). A hálózati forgalmat figyeli folyamatosan és a rendelkezésére álló adatbázis segítségével értelmezi azt. Célja, hogy az adatbázisában tárolt, előre definiált konkrét támadási mintákat felismerje és megakadályozza. A Snort-ot felkészítették buffer overflow exploitok felismerésére is. Itt a legegyszerűbb esetben csak ellenőrzi, hogy az aktuálisan ellenőrzött tartalom nem tartalmazza-e az adatbázisban tárolt minták egyikét: például shell code-okra könnyen rá lehet keresni, ha valaki ilyet akar beküldeni a rendszerbe, akkor biztos, hogy támadni akar. A shell code-ok legtöbbször tartalmaznak egy `/bin/sh` karaktersorozatot, ami így újabb, az adatbázisban nem létező shell code-okat is képes detektálni.

4.4.1.2.2 Forráskód elemzés, tesztelés

A forráskód automatizált elemzésével [22],[15] rendkívül sok típushiba kiszűrhető és kis költséggel kijavítható. A legtöbb programnyelvhez rendelkezésre áll

ilyen elemző, külön opcióként képesek megjegyzésekkel kiegészíteni a forráskódot, minden egyes veszélyesnek ítélt parancsnál javaslatot téve a kijavítására.

4.4.1.2.3 Bináris állomány tesztelése

Buffer overflow jellegű hibák nagy többsége kiszűrhető pusztán a bináris állomány tesztelésével is, bár erre kiforrott, jól működő, széles körben elérhető megoldások még nincsenek. Számos ilyen irányú kutatás azonban folyik.

4.4.1.2.4 Buffer keretezés

A keretezés célja, hogy futásidőben észrevegyük, hogy egy írási művelet a buffer határait megsértette. A fordítóprogram egy-egy rendszerint négybájtos értéket fűz a buffer elejéhez illetve végéhez. A bufferbe történő írás után a fordító által hozzáadott kód ellenőrzi, hogy a keretező értékek megegyeztek-e az írási művelet előtt illetve után. Amennyiben az írás utáni érték nem egyezik a korábbival, úgy feltételezhető, hogy buffertúlcsordulás történt, ezért biztonsági okokból a program futását meg kell szakítani.

A módszer előnye, hogy képes azonnal érzékelni a túlírást, valamint nem igényli a meglévő forráskód megváltoztatását. Természetesen újrafordításra szükség van.

Hátránya, hogy a keretezés növeli a memóriaigényt, valamint a sorozatos ellenőrzések rontják a teljesítményt. Egy gyorsítási lehetőség, ha az ellenőrzés csak a hívott függvény végén történik meg. Ekkor ugyan nem észlelhető azonnal a buffer túlírás, de jelentősen csökken a detektálás teljesítményigénye és a legfontosabb visszatérési cím felülírása ellen továbbra is védelmet nyújt (Ez a védelem persze nem tökéletes, mert nem biztos, hogy túlcsorduláskor a keret biztosan módosul, illetve a támadó intelligensen az előző értékeket is használhatja felülírás során, ami így nem detektálható ezzel a módszerrel).

Konkrét megoldások:

Visual C++ 7.1: a keretezéshez a hozzáadott kód a teljes lokálisváltozó-területet ***0xCC*** *bájtokkal tölti fel, illetve helyet hagy a buffer kereteknek. Az ellenőrzés a kódhoz fűzött rövid adatblokk alapján történik, amely alapján egy univerzális rutin azonosítja és ellenőrzi a bufferhatárokat. A keretezés konstans, tehát ha a buffertúlírás pont a keretező értékkel (tehát* ***0xCC****-vel) történik, akkor a védelem megkerülhető, tehát kibocsátott termékek biztonságosabbá tételére alkalmatlan. A Microsoft ajánlása szerint*

ez a védelem csak a túlírható bufferek azonosítását hivatott meggyorsítani a tesztelési fázisban.

GNU C Compiler: a `–fomit-frame-pointer` kapcsolóval tudjuk aktivizálni az ilyen irányú védelmet. A `–fomit-frame-pointer` üres területeket szúr be a stackre a lokális változók közé. Sajnos ez a védelem könnyen megkerülhető. Helyette az IBM `icc` fordítóját szokták használni (az Adamantix is ezt használja, lásd lentebb).

4.4.1.2.4.1 Security Cookie Check

Segítségével detektálható, ha buffertúlírással kísérlet történt a visszatérési cím felülírására. Függvény hívásakor a lokális változók illetve a vezérlőstruktúrák közé bekerül egy (általában véletlenszerű) speciális érték. Visszatéréskor a hozzáadott kód ellenőrzi, hogy a security cookie értéke megváltozott-e, ha igen, akkor valószínűleg buffertúlírás történt, és megszakítja a program futását.

A módszer előnye, hogy a buffer keretezéshez képest jelentősen csökken a többletmemória-igény, miközben éppen kritikus a visszatérési pointer védelem hatékonyabb. A buffer keretezéshez hasonlóan a kód megváltoztatását nem igényli, de újra kell fordítani a programot.

Hátránya, hogy valamennyire romlik a program teljesítménye (habár a teljesítményromlás jóval enyhébb, mint szigorú buffer keretezés esetén), valamint nem nyújt védelmet egyéb kontroll-adatok (függvénypointerek, exception frame-ek) felülírása ellen.

Konkrét megoldások:

Visual C++ 7.1: a C Runtime Library inicializálásakor létrehoz egy kvázi-véletlen cookie értéket, amelyet aztán minden buffereket használó függvény hívása esetén a stackre másol. A megoldás hátránya lehet, hogy a cookie csak a program indulásakor jön létre, a futás során végig ugyanazt az értéket alkalmazza, tehát ha valamilyen hiba folytán bárhol kiderül a cookie értéke, akkor a védelem megkerülhető.

4.4.1.2.4.2 Újrafordított standard könyvtári függvények

Mivel a standard C könyvtári függvények alapesetben nem tartalmaznak védelmet a hiba ellen, egy lehetséges megközelítés a hiba bekövetkezésének kivédésére, hogy újrafordított standard függvény könyvtárakat használunk. Amikor egy program meghív egy standard libc fügvényt akkor már az új, javított implementáció hívódik meg. A leginkább problematikus függvények:

- `strcpy(char *dest, const char *str)`

- `strcat(char *dest, const char *src)`
- `getwd(char *buf)`
- `gets(char *s)`
- `scanf(const char *format,…)`
- `realpath(char *path,char resolved_path[])`
- `sprintf(char *str, const char *format,…)`

Normális esetben a standard libc függvénykönyvtár azért nem biztonságos, mert nem ellenőrzi le a buffer méretét. Az itt látható implementáció problémája, hogy nem addig másolunk bele a cél bufferbe, amíg be nem telik, hanem amíg a forráson nem olvasunk egy `'\0' karaktert`.

```
char* strcpy(char* dest, const char* src)
{
    char* tmp=dest;
    while( (*dest++=*src++)!='0' );

    return tmp;
}
```

22. ábra: Strcpy implementációja

Libsafe [18] esetében az `strcpy` a 23. ábrán látható módon módosul. Annyi az eltérés, hogy itt van túlírás ellenőrzés.

```
char* strcpy(char* dest, const char* src)
{
    …
    if( (len=strlen(src, max_size)) == max_size )
      _libsafe_die("Overflow caused by strcpy()");

    real_memcpy(dest, src, len+1);
    return dest;
}
```

23. ábra: Libsafe esetén az strcpy implementációja

Ennek a megoldásnak nagy előnye, hogy átlátszó, így a meglévő programjainkkal is használhatjuk. Amennyiben túlírás következik be, a standard outputra kiírja hibaüzenetként a legfontosabb információkat (uid, euid, pid, stack aktuális tartalma, milyen függvény idézte elő). A hibakeresést nagyban megkönnyítik ezek a segéd információk.

Természetesen ennek a megoldásnak is vannak határai. Mivel így egy dinamikus függvénykönyvtárat cseréltünk le, a programunkhoz statikusan linkelünk, akkor megkerültük ezt a védelmet. A megoldás nagy hátránya, hogy jelentős erőforrás szükséglete van.

Konkrét megoldások:

Libsafe[18]

4.4.1.3. Kihasználás megakadályozása

A kihasználás megakadályozásának lényege, hogy még a konkrét hiba előfordulása esetén se lehessen kárt okozni a rendszerben, azaz megakadályozzuk, hogy a hiba biztonsági lyuk keletkezéséhez vezessen.

4.4.1.3.1 Kernel szintű védelmi megoldások

Kernel szintű védelem esetén az operációs rendszer magja tartalmazza az aktív védelmi mechanizmust. Ez lehet a kernel gyártójától származó megoldás is, vagy később is injektálható a kernelbe (ez nyílt forrású kernel esetén tehető meg). A rendszermag módosítása nélkül is elérhetőek ilyen mélységű módosítások, kernel modul, illetve driverek segítségével. Fontos megemlíteni, hogy a kernel szintű megoldások általában nagyon hatékonyak, mert transzparensek és gyorsak. Természetesen az ilyen szintű megoldás sem tudja garantálni a biztonsági hibák tökéletes kiszűrését. A rendszermag szintű védekezési megoldások mind úgy tekintenek a biztonsági lyukak problémájára, hogy feltételezik azok létezését, illetve azt, hogy detektálni sem tudjuk őket, de ilyenkor is megpróbáljuk minél jobban megnehezíteni a kihasználásukat.

4.4.1.3.1.1 Non executable stack, hardvertámogatás nélkül

A legtöbb exploit (kihasználó program) a stack vagy heap túlcsordulását használja ki. A buffert az elindítani kívánt programmal tölti fel (ami általában a stack-en helyezkedik el). Ezek után a függvény visszatérési címét módosítva a bufferben elhelyezett programot indítja el. Az itt következő megoldások azt célozzák meg, hogy ne lehessen végrehajtani a bufferbe bemásolt kódot.

Konkrét megoldások:

Open Wall [17] kernel patch: az ide vonatkozó forráskód (24. ábra, `arch/i386/kernel/traps.c`).

```
/*
* Check if we are returning to the stack area, which is only likely to
happen
* when attempting to exploit a buffer overflow
*/
if ( (addr & 0xFF800000)==0xBF800000 ||
( addr >= PAGE_OFFSET-_STK_LIM && addr<PAGE_OFFSET ) )
            security_alert("return onto stack by ", DEFAULTSECMSG,
                  "returns onto stack",DEFAULTSECARGS);
}
```

24. ábra: Védelem non-executable flag alkalmazásával

Ez a kódrészlet teszteli, hogy az aktuális cím (`addr`) a stack-en van-e. A jelen esetben, ha a cím **`0xBF800000`** és **`0xBFFFFFFF`** között van. Tehát akkor a stack-en vagyunk, hiba generálódik. Ez a megoldás sokkal inkább teljesítménypazarló, mint a következő. Valószínűleg ennek tudható be, hogy legtöbben nem ezt a stack shield-et preferálják.

4.4.1.3.1.2 Non-executable stack and heap hardvertámogatásal

A cél teljes mértékben megegyezik az előbbi megközelítéssel, [21], [19] itt viszont a virtuális memóriakezelés lehetőségeit használjuk ki. Minden egyes memórialaphoz hozzáférési jogosultságot definiálhatunk (read/write/execute). Vagyis a stack terülten lévő lapok execute flag-jét 0-ra állítjuk. Hatékony megoldás, hiszen ehhez hardvertámogatásunk van. Másik lehetséges megoldás az NX flag használata. Így ha stack területről próbálunk meg kódot futtatni, hardveres megszakítás történik.

Konkrét megoldások:

PAX non-executable stack and heap (kernel patch) [21]: A fent leírt módosításokat hajtja végre a Linux kernel virtuális memóriakezelő alrendszerében.

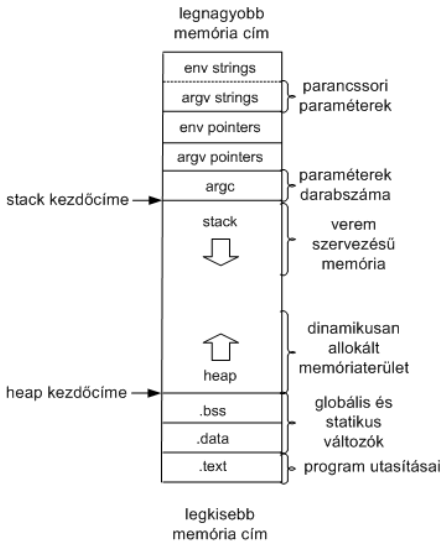
*MSSQL Secure Stack**Hiba! A hivatkozási forrás nem található.**: Nem sokkal az MS-SQL Worm megjelenése után került a piacra a SECURESTACK nevű program, aminek kifejezett célja, hogy az ehhez hasonló MSSQL szerver elleni támadásokat kivédje. A megvalósításhoz a PAX linux security kernel patch non-executable stack megoldását portolták Windows-ra, azonban csak az MSSQL szerver elleni támadásokat képes kivédeni. Amennyiben éppen egy támadás van folyamatban, és a stack területen el akar indulni egy kód, a felhasználót figyelmezteti, és lehetőség van eldönteni, hogy engedélyezzük-e a műveletet vagy sem. Leginkább egy personal firewall működéséhez hasonlítható.*

NSA: Security Enchanced Linux [27]: A védekezési módszer itt is ugyan ez, kicsit más a megvalósítás.

4.4.1.3.1.3 Stack pointer randomization

A virtuális memóriakezelésnek köszönhetően, minden egyes folyamat úgy látja, mintha csak ő lenne egyedül a memóriában. Mivel a számítógép egy determinisztikus gép, ha kétszer egymás után elindítunk egy programot, a memória térképük teljesen meg fog egyezni. A többi folyamat memória használata ebbe nem szól bele, hiszen nekik is megvan a saját területük. Egy konkrét támadásnál ahhoz, hogy el tudjuk indítani a

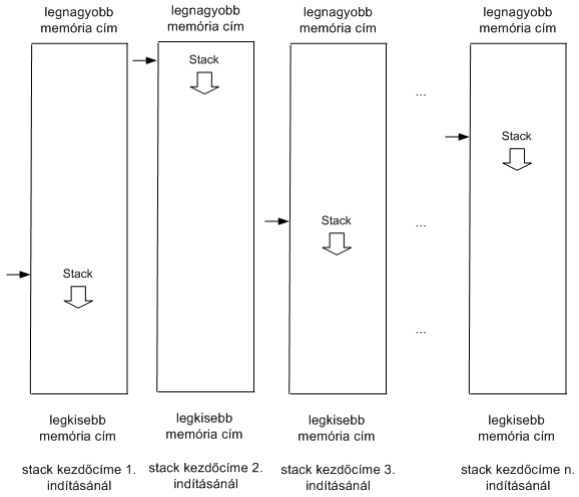
buffer területbe bemásolt programunkat, szükségünk van a programunk kezdőcímére. Az előbb említett determinisztikusságnak köszönhetően ez nem változik. A támadónak egy ugyanolyan kiépítésű rendszeren kell kiderítenie, hogy hol fog allokálódni a buffer, azt a kezdőcímet rögzíti a programjában, majd elkezdheti a támadást. Az ilyen támadó programok tipikusan fel vannak készítve a legismertebb disztribúciókra, vagyis minden disztribúcióhoz tartozik egy hexadecimális memória cím. A támadó program pedig ezeket próbálja végig.



25. ábra: Futó program memóriatérképe Linux rendszereken

Egy hatékony megközelítés, ha megpróbáljuk megnehezíteni a támadó dolgát oly módon, hogy kiszámíthatatlanságot csempészünk a rendszerbe, a virtuális memóriaterületen véletlenszerűen allokálva a stack kezdőcímét (26. ábra). Ebből következik, hogy az előre elkészített (statikus címet feltételező) támadások nem fognak működni; a nyers erő módszere természetesen még mindig rendelkezésre áll, de kivitelezhetőségének esélye próbálkozásos módszerrel szinte nulla.

A stack pointer randomization mellett, ugyanezen ötlet felhasználásával heap randomization, executable randomization, shared libary randomization is bevethető. A módosítás igényéhez képest nagyon hatékonyan működő módszer és gyakorlatilag nincs erőforrás igénye a virtuális memória hardver támogatása miatt.



26. ábra: Véletlenszerű stack elhelyezés

Konkrét megoldások:

A PAX kernel patch[21] randomization része

Adamantix (régebben TrustedDebian [13])

SUN Solaris [14]

FreeBSD [11]

4.4.2. Egész túlcsordulás (integer overflow) / Előjeles hiba (Signedness bug)

A hiba jellegzetessége, hogy önmagában nem kihasználható, csak más hibával párosulva, ami a védekezési módszerekre is rányomja bélyegét.

4.4.2.1. Megelőzés

A stack overflow-nál ismertetett megelőzési módszerek egy része itt is használható.

4.4.2.2. Felismerés

Az integer overflow és az előjeles hibák forrása, hogy többféleképpen is értelmezzük (értelmezhetjük) az integerként tárolt adatot. A kétértelmű hivatkozást nagyon nehéz felismerni. Viszont mind integer overflow, mind pedig előjeles hiba esetén ahhoz, hogy a támadó ezt kihasználja, el kell érnie, hogy az integer túlcsorduljon, vagy pedig negatív értéket vegyen fel. Ennek eléréséhez egy nagy számot, vagy pedig egy olyan számot kell inputként megadnia, amellyel elvégzett művelet után a felvett

érték negatív lesz. A két feltételt összevonva megállapítható, hogy ennek a számnak **0x80000000** és **0xFFFFFFFF** között kell lennie.

A másik fontos megállapítás, amit le lehet vonni a támadási célokból, hogy először a támadónak ki kell kerülnie a védelmet egy ilyen hiba kihasználásával, majd pedig a védelem nélkül maradt buffer túlírását ki kell használnia. Az esetek legtöbbjében egy ciklus valósítja meg a célbufferbe való írást, ezért a sebezhetőség kihasználása detektálható, ha ellenőrizzük, e ciklusok körülfordulásának számát (a ciklusváltozó méretét). Ha tehát „túl sokszor" akar egy ciklus végrehajtódni, akkor megállítható, jelezhető a lehetséges támadás. A fentebb említett beinjektált szám intervallumát és a ciklus futások átlagos számát figyelembe véve megállapítható, hogy 1 milliárd feletti végrehajtási szám potenciális támadásnak tekinthető és ez esetben ellenintézkedések tehetők.

Konkrét megoldások:

GNU gcc compiler patch (big loop integer protection [5])

4.4.2.3. Kihasználás megakadályozása

A tényleges kihasználáshoz egy buffer, illetve heap overflow hibát kell találnia a támadónak, ezért az ott említett megoldások itt is használhatóak.

5. Összefoglalás, konklúzió

Az előző oldalakon olvasható összefoglaló egy széles áttekintést ad a jelenleg használt védekezési technikákról. Ahogyan a 3.2-es fejezet is sugallta, hiányosságok még mindig bőségesen vannak. Sajnos elméletileg is tökéletes védelmet biztosító megoldás jelenleg nem létezik. A hibák megjelenését és a kihasználások egyediségét megfigyelve valószínűsíthető, hogy mindig is lesznek valamilyen támadásra lehetőséget nyújtó rések, így filozófiai megközelítéssel élve gyakorlatilag kizárható, hogy bármikor is tökéletes védelmet lehessen garantálni.

Viszont nagyon fontos tudatosítani, hogy attól, hogy tökéletes védelmet implementálni nem lehet, még alkalmazni kell a rendelkezésre álló védekezési technológiákat, amelyekkel akár egy nagyságrenddel is visszaszorítható az előfordulásuk, illetve a kihasználásuk, ami pedig már hatalmas eredmény! Ezért az adott és már kifejlesztett lehetőségekkel élni kell!

6. Irodalomjegyzék

[1] NetworkWorldFusion, Paul Roberts: Major companies team on vulnerability rating system, 2005 (http://www.nwfusion.com/news/2005/0218rsa-maj.html)

[2] Michael Howard, David LeBlanc: Writing secure code, Microsoft Press, Redmond, Washington, 2003

[3] Aleph One: Smashing the Stack for Fun and Profit. Phrack Magazine, 1996 (http://www.cs.ucsb.edu/~jzhou/security/overflow.html)

[4] Blexim: Basic Integer Overflows. Phrack Magazine, 2002 (http://www.phrack.org/phrack/60/p60-0x0a.txt)

[5] Matt Conover, Oded Horovitz: Big Loop Integer Overflow protection In Phrack Magazine, 2002, (http://www.phrack.org/show.php?p=60&a=9)

[6] w00w00 Security Development (WSD): w00w00 on Heap Overflows, 1999 (http://www.w00w00.org/files/articles/heaptut.txt)

[7] Security Enhanced Linux, National Security Agency, (http://www.nsa.gov/selinux)

[8] Java Technology Home Page (http://java.sun.com/)

[9] Perl Home Page (http://www.perl.com/)

[10] Visual C# Developer Center (http://msdn.microsoft.com/vcsharp/)

[11] FreeBSD Handbook (http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html)

[12] Cyclone, (http://www.research.att.com/projects/cyclone/)

[13] Adamantix (trusted Debian), (http://www.adamantix.org)

[14] Sun Solaris Product Home (http://www.sun.com/software/solaris/)

[15] Flaw finder - source code security scanner, (http://www.dwheeler.com/flawfinder/)

[16] Snort, Intruder Detection System, (http://www.snort.org/)

[17] OpenWall (http://www.openwall.com/linux)

[18] Libsafe Project (http://www.research.avayalabs.com/project/libsafe/)

[19] GR security kernel patch (http://www.grsecurity.net)

[20] RSBAC – Rule Set Based Access Control, (http://www.rsbac.org)

[21] PAX kernel patch (http://pageexec.virtualave.net)

[22] RATS, (http://www.securesoftware.com)

[23] K-OTIK Security: Search and Replace Compressed File search Local Buffer Overflow Exploit, (http://www.k-otik.com/exploits/20050124.searchnreplace.c.php)

[24] K-OTIK Security Advisory: Search and Replace local compressed file search Buffer Overflow, (http://www.k-otik.com/english/advisories/2005/0055)

[25] K-OTIK Security: Cisco IOS 12.x/11.x HTTP integer overflow remote Vulnerability and exploit

[26] FreeBSD-SA-02:38: Boundary checking errors involving signed integers (http://www.securityfocus.com/advisories/4407)

[27] Security Enchanced Linux, National Security Agency, (http://www.nsa.gov/selinux/index.cfm)

[28] Nergal: The advanced return-into-lib(c) exploits, Phrack Magazine, 2001 (http://www.phrack.org/show.php?p=58&a=4)