Debreceni Egyetem

Informatikai Kar

PROGRAMVÉDELEM

Témavezető:

Dr. Juhász István

Egyetemi Adjunktus

Készítette:

Vágner József

Prog. Mat.

Debrecen
2008

Tartalomjegyzék

Bevezetés

A programok védelme nagyon fontos, mégis kevés szó esik róla még napjainkban is, ezért esett rá a választásom. A programvédelem célja megvédeni szellemi termékünket az illegális másolástól, így emelve az eladásokból származó bevételt. A programozók többnyire nem, vagy csak nagyon gyenge módszerekkel védik elkészült programjaikat (holott a programvédelembe fektetett munka kifizetődő), nem is gondolva, hogy valaki feltörheti azt. Ez viszont téves feltételezés, ugyanis a cracker csoportok pont a védelmek eltávolítására szakosodtak.

Ezen csoportok tagjai olyan magas tudású programozók, akik jártasak a rendszerprogramozásban, gond nélkül használják az Assembly nyelvet, és hihetetlenül tájékozottak a biztonság terén, ezért nem meglepő, hogy a köreikből kerülnek ki a legjobb védelmek megalkotói is. Kitartó munkájuk pedig egyáltalán nem anyagi jellegű, csupán az elismerés, és a kihívás hajtja őket. A védelmek hibái pont az ezen a téren megszerzett tudás és tapasztalat hiányában keresendők. A cracker-ek egy komolyabb védelem sikeres kiiktatása után titkos fórumokon osztják meg társaikkal tapasztalataikat, naprakész információikat. Sajnos a programozók nem is gondolják, hogy biztonságosnak hitt védelmük már régen a múlté, és netán következő programjaikba is beépítik.

Dolgozatomban igyekszem olyan módszereket bemutatni, amik segítenek hathatós védelmet építeni programjainkba. Persze külön-külön egyik módszer sem tökéletes, de kombinálva több védelmet, megnehezíthetjük a feltörés esélyét. Persze a cracker körökben sem ismeretlenek ezek a módszerek, így a kódban való elrejtés komoly odafigyelést követel. A legnagyobb figyelmet a nyomkövetőkre helyezem, ezek közül is a népszerű SoftICE debugger elleni módszereket részesítem előnyben. Igyekszem naprakész, minden Windows verzióban működő megoldásokat bemutatni, de a főbb hangsúlyt a manapság legsűrűbben használat Windows XP SP2, és Windows Vista operációs rendszerekre helyezem.

A védelmi módszereket legkönnyebben Assembly nyelven lehet megvalósítani, de mivel a nyelv már nem annyira közismert és használatos a programozók köreiben, ahol lehet, magasabb szintű nyelvet használok a példák bemutatására, illetve foglalkozom a magasabb szintű nyelvekbe való Assembly beillesztésével is.

A dolgozat végén említésre kerülnek az optikai lemezen történő terjesztés folyamán fellépő problémák, és az ezek ellen használható másolásvédelmi módszerek is.

1 A programvédelem típusai

1.1 Sorozatszám alapú védelem

Ha a programot sorozatszám alapú védelemmel látjuk el, a felhasználónak a program regisztrálásához be kell írnia a megfelelő sorozatszámot. Ez a sorozatszám egyes esetekben mindig ugyanaz, máskor a bekért adatok függvényében változik (pl.: név, cég neve), vagy a felhasználó számítógépétől függően változik. Újabban már lehetőség van a sorozatszámot az interneten keresztül is ellenőrizni.

1.1.1 Fix sorozatszám használata

A program felkéri a felhasználót, hogy írja be a sorozatszámot. Minden felhasználónak egymástól függetlenül ugyanazt a sorszámot kell begépelnie, ezért a cracker-nek nincs nehéz dolguk. Csak meg kell keresnie a kódban, vagy megvásárolnia a programot, és máris hozzájut a helyes számhoz.

A védelemnek mégis van egy nagy előnye. A sorozatszámot nem kell a memóriában elhelyeznünk ahhoz, hogy összehasonlítsuk a beírttal. A vizsgálat folyamán műveleteket végünk a beírt értéken (ami általában XOR művelet, és még néhány más műveletből áll), aztán a helyes sorszámon is elvégezzük ezeket, és az eredményeket összevetjük. A cracker-ek munkáját megnehezíti, ha műveletek bonyolultabbak. A program több részén is használhatjuk ezt a védelmet (pl.: ha a mentés vagy a nyomtatás lehetőségét is védjük). Ebben az esetben, ha cracker nem szerezte meg a helyes kódot, hanem patching (foltozás) módszerrel a programkódban közvetlenül átírja a feltételeket, akkor a letiltott módszerek továbbra sem működnek majd. Biztonságosabb, ha ezeket a lezárt részeket nem a helyes sorszámbeírása után tesszük elérhetővé, hanem a program indítása után, vagy még jobb, ha csak a lezárt rész első indítása előtt. Sőt, ha a használat után ismét kódoljuk, a program sosem lesz a memóriában teljesen dekódolva, ami a memória vizsgálatával próbálkozó cracker-ek munkáját lehetetleníti el.

1.1.2 Beírt adatoktól változó sorozatszám

Napjainkban ez a legelterjedtebb. A módszert használó programok bejegyzésénél meg kell adnunk néhány adatot magunkról (ez rendszerint a név és/vagy a cég neve) a helyes sorszám pedig ezen adatok függvényében változik. A cracker ebben az esetben is használhat egy más által megvásárolt kódot, de ekkor a beírt adatokat is meg kell tartania. Ha ezt nem szeretné, a program működését kell nyomon követnie. Lehet a számítási algoritmus kellően bonyolult, a végén össze kell hasonlítani a beírt kóddal, és ha ezt a cracker megtalálja a kódban, csak át kell írnia a feltételt és a program helytelen sorozatszámok estén indul csak el.

A védelmet alakíthatjuk úgy, hogy a beírt adatok minden esetben egy meghatározott eredményt adjanak, így több eredmény is megfelel, és sorozatszámot nem kell ellenőrizni, hanem egy rejtett algoritmust kell elhelyezni a kódban, ami megvizsgálja, hogy a sorozatszám valóban helyes volt-e. Hiba estén nem célszerű hibaüzenetben vagy kiugró ablakban tájékoztatni a felhasználót, elegendő csak az adatok bekérését megismételni.

1.1.3 Számítógéptől változó sorozatszám

Ez a védelem az egyik legnehezebben feltörhető, és ha még sikerül is a cracker-nek feltörnie a saját gépen, a feltört változat sok esetben másoknál nem fog működni. Ezen védelem használata esetén a sorozatszám a hardvertől függ (pl.: a merevlemez vagy a CPU gyártási számától). Az ellenőrzést gondosan el kell rejteni, mert ha a cracker megtalálja a kódban, átírhatja egy állandó értékre, és egy hozzá tartozó sorozatszámmal már bárki gépen használható. A módszert érdemes más módszerekkel együttesen alkalmazni.

1.1.4 Interneten ellenőrzött sorozatszám

Legújabban ezt a módszert választják drága programok védelmére (pl.: a Windows Vista operációs rendszer esetén). Miután a felhasználó beírja a sorozatszámot, a program az

Internet segítségével elküldi ellenőrzésre. A kiszolgáló ellenőrzi a kapott adatokat, és egy jelentést küld a programnak, melyben értesíti, a vizsgálat eredményéről. A program feldolgozza a jelentést, és ez alapján dönt a bejegyzésről. A látszat ellenére a legtöbb ilyen védelem meglehetősen egyszerű, könnyen eltávolítható.

Néhány ilyen védelemmel ellátott program véletlen ellenőrzéseket hajt végre, néha frissítésként álcázva. Ha rájön, hogy csalás történt rögön korlátozza a működését, vagy adatokat küld a felhasználóról a gyártónak az Interneten.

A hálózaton keresztül történő ellenőrzés elterjedt, de nem használható minden esetben. Ha program egyébként is hálózaton kell lennie funkciója miatt (pl.: böngésző, FTP szerver) célszerű használni, egyébként megfontolandó nem okoz-e bonyodalmakat (esetleg az eladások csökkenését).

A védelem biztonságosabb módja, amikor a kiszolgáló a helyes sorozatszám esetén létfontosságú adatokat küld vissza a programnak. Írjuk meg a programunkat úgy, hogy a regisztrálatlan változatban például a mentés lehetősége nem használható. Miután a felhasználó beírta a sorozatszámot, a program elküldi a szerverhez. Ha a megadott szám helyes, a szerver visszaküld egy rövid adatcsomagot, mely beépülve a programba lehetővé teszi a mentést. Ilyenkor a cracker-nek nem elég becsapnia a programot, és elhitetnie vele, hogy a kiszolgáló elfogadta a beírt sorozatszámot (sőt még a kiszolgálót is becsaphatja), a lezárt funkció akkor sem fog működni.

1.2 Időkorlát

A programot időkorláttal láthatjuk el. A program rendelkezik egy kipróbálási idővel, aminek a letelte után nem használható tovább. Nem tartozik a legjobb védelmek közé, mivel a cracker dolga mindössze az időkorlát eltávolítása vagy kellően nagyra módosítása, és a teljes program a birtokába kerül. Ezért jobb megoldás, hogy a regisztrálatlan változatban a program bizonyos részeit lezárjuk.

1.2.1 Időkorlát megszűnik, ha a felhasználó beírja a megfelelő sorozatszámot

Ez a megoldás az előzőekben ismertetett problémákkal rendelkezik azzal a különbséggel, hogy ha nem írjuk be a helyes sorszámot, a program regisztrálása nem lesz sikeres, és egy bizonyos idő után a program nem indul el.

Ha mégis ezt a módszert választanánk a sorozatszám előállításának megfelelően bonyolultnak kell lennie, mivel a cracker-ek nem az időkorlátra helyezik a hangsúlyt. A hátralévő használati idő ellenőrzésre elég egy egyszerű kód, ami például az első használatnál lekéri a dátumot, majd eltárolja egy file-ban vagy a regiszterben, és ha a felhasználó túllépte a korlátot, megtagadja a futást.

1.2.2 Időkorlát megszűnik, ha a megadjuk a regisztrációs (.REG) file-t

Ezt a védelmi módszert ritkán használják. A regisztrációs file-t érdemes Interneten keresztül elküldeni, ami olyan programrészletet tartalmaz, ami feloldja az időkorlátozást.

A cracker-ek arra az eljárásra összpontosítanak, mely az időkorlát leteltét ellenőrzi, így ezt hatékonyan célszerű védeni. A cracker aligha fog a helyes regisztrációs file összeállításával bajlódni, mivel ez meglehetősen nehéz feladat.

Ne használjunk olyan ellenőrző módszert, amely a regisztrációs file jelenlétét vizsgálja a program könyvtárában, továbbá ellenőrzi a tartalmát. Sokkal hatékonyabb megoldás, ha a védett alkalmazás kódját a regisztrációs file-ban helyezzük el. Számos vírusirtó program alkalmazza ezt a védelmet.

1.2.3 Időkorlátot nem lehet eltávolítani

Programok bemutató-változataiban gyakran alkalmazott módszer. Az időkorlát letelte után a programot nem lehet többé elindítani, sorozatszám beírására nincs lehetőség. A cracker ilyen védelem esetén az időkorlátot vizsgáló eljárásra összpontosít, ezért érdemes a file ellenőrző összegét (checksum) is megvizsgálnunk.

1.2.4 Az időkorlát bizonyos számú indítást engedélyez

Ez a módszer megegyezik az időkorlátossal, de itt a programindítások számát is korlátozzuk. Ezzel megnehezíthetjük a cracker-ek dolgát, mert nem kell az időkorlátot vizsgálni, eltárolni. Elegendő a programindítások számát eltárolni valahol (pl.: regiszterben, vagy egy kódolt file-ban)

1.3 Regisztrációs file alapú védelem

Ez a védelem egy regisztrációs file-t (kulcsfile-t) készít, és többnyire a program telepítési könyvtárában helyezi azt el. A program az indítás után ellenőrzi ezt a file-t, és ha megfelelő a tartalma, a program bejegyzettnek tekinthető a továbbiakban. Ha nem található vagy hibás, a program nincs bejegyezve, vagy el sem indul.

1.3.1 A regisztrációs file hiányában a program bizonyos részei nem elérhetők

Ez a módszer nagyszerű védelmet ad a programoknak. Eltávolítása meglehetősen nehéz feladat. Használata esetén a program bizonyos részeit zároljuk, ha nincs meg a helyes regisztrációs file. Amint bekerül a program könyvtárába a megfelelő file, a regisztráció megtörténik, és lezárt részek feloldódnak.

Ha ezt a módszert választjuk, a regisztrációs file elkészítésére és kódolására nagy figyelmet kell fordítani, és egyéb teszteljárásokat is beépíthetünk. Programozástechnikailag nehezebb feladat, ha a lezárt eljárás kódját a regisztrációs file-ban helyezzük el, vagy olyan állandót írunk bele, mely lehetővé teszi a feloldást. Ez a védelem a cracker számára áthatolhatatlan, de ha hozzájut egy helyes regisztrációs file-hoz, bejegyezheti a programot.

1.4 Hardverkulcsos (dongle) védelem

Ez a védelem egy ritkább és drágább megoldás programok védelmére. Működése egy I/O kapura illeszthető hardverkulcsra épül, melynek a jelenléte nélkül a program nem indul el, vagy csak korlátozott üzemeltetés lehetséges. A két legismertebb, és széles körben elterjedt hardverkulcs a HASP és a Sentinel.

1.4.1 A program nem indul hardverkulcs nélkül

A legtöbb hardverkulcs nagyon egyszerű módon működik. A program adatokat továbbit a kapura, és választ vár. Ha a válasz nem érkezik meg leáll és üzenetben értesíti a felhasználót. A fejlettebb hardverkulcsoknál a kapuhoz küldött adatok kódoltak, de az is előfordul, hogy a kulcs tartalmaz egy EPROM-ot, ami a program lényeges részeit tárolja. A cracker dolgát tovább nehezítve, mivel ha csak a programmal rendelkezik, szinte lehetetlen a védelmet eltávolítani, vagy a hiányzó kódrészletet elkészíteni.

A kulcs jelenlétét vizsgáló eljárást számos módon felkutathatjuk. Például gyakran használnak API átirányítást, ami minden hívásnál megvizsgálja, hogy jelen van-e a kulcs. Egy másik megoldásban az API hívások a kulcshoz kapcsolódó függvényekből történik. Egyéb, még fejlettebb kulcsok saját meghajtókkal rendelkeznek. A gyártó cégek folyamatosan fejlesztik védelmi rendszereiket, újabb meghajtókat adnak ki, mivel ha egy cracker megtanulja egy adott típus eltávolítását, bármely programból el tudja távolítani ezt a fajta védelmet. Sajnos még ezek ellenére is találhatunk emulátorokat, melyekkel utánozni lehet a kulcsok jelenlétét. Ezért hardverkulcsos védelem használatánál törekedjünk arra, hogy a program kódjából érjük el a kulcsot, és ne hagyatkozzunk a gyártó által biztosított meghajtókra vagy API hívásokra. Ha cracker nem talál megfelelő emulátort, megkísérli a közvetlen utánzást a program kódjában. Nem nyúl a meghajtó kódjához, mivel azt erősen védik a gyártók. Ez ellen a legegyszerűbb védelem, ha a file-on CRC-ellenőrzést hajtunk végre, ami jelzi az utólagos változtatásokat.

A hardverkulcs legnagyobb hátránya, hogy minden termékhez mellékelni kell a kulcsot. Ami az eladási árat növeli, és a terjesztést is megnehezíti, így többnyire csak igen drága szoftverek estén fordulnak elő.

1.4.2 A program bizonyos részei korlátozottak hardverkulcs nélkül

Ha nincs jelen a hardverkulcs, a program bizonyos részeit, funkcióit nem lehet elérni. Ha a kulcsot csatlakoztatjuk, a funkciók elérhetőek lesznek. Az EPROM-mal rendelkező kulcsok esetén, ezen funkciók magában a kulcsban is lehetnek, ami tovább növeli a biztonságot, vagy a kulcs olyan információkat tartalmazhat, amik szükségesek a feloldáshoz. Ha jó a kódolás szinte lehetetlen eltávolítani a védelmet.

1.5 A HASP hardverkulcs

Az Aladdin Knowledge System fejlesztése. A védelemmel ellátott program telepítésekor a HASP is telepíti saját meghajtóját, amelynek a segítségével cserél adatokat a program és a kulcs. A HASP szinte minden operációs rendszerhez elkészítette a meghajtóját (DOS, Windows 9x/NT/2000/XP/Vista, Mac OS X, Linux). A meghajtó HASP API hívásai biztosítják a kommunikációt, amit könnyen meg is lehet találni a kódban (a cmp bh,32 hívást kell keresni).

```
HASPHivas:
cmp bh, 32      ;Hasp szolgáltatás-e?
        jb jump
        mov esi, dword ptr [ebp+28]
        mov eax, dword ptr [esi]
jump: mov esi, dword [ebp+20]
mov esi, dword ptr [esi]
push ebp
call Hasp()      ;Az alap HASP szoláltatás hívása

pop ebp
mov edi, dword ptr [ebp+1C]
mov dword ptr [edi], eax        ;vissztérési érték mentése
mov edi, dword ptr [ebp+20]
mov dword ptr [edi], ebx        ;vissztérési érték mentése
mov edi, dword ptr [ebp+24]
mov dword ptr [edi], ecx        ;vissztérési érték mentése
mov edi, dword ptr [ebp+28]
mov dword ptr [edi], edx        ;vissztérési érték mentése
```

Az alap HASP szoláltatás mindig ugyanabból a hívásból áll. A program a döntéseit a híváskor átadott paraméterek alapján hozza meg, majd kiválasztja, melyik HASP szolgáltatást hívja meg.

A következő esetben a program a HASP 3. függvényét hívja meg, melynek a ReadWord( ) a neve.

```
push eax
push ecx
push 000047FE   ;jelszó1
push 000015C9   ;jelszó2
push ebx        ;nyomtatókapu (LPT)
push edi
push 00000003   ;HASP 3-as syolgáltatás
mov [esp+38], 0000001A
                ;cím
call HASPHivas  ;meghívja a ReadWord() szolgáltatást
```

A többi függvény is hasonló módon hívható meg.

1.5.1 IsHasp( ) 1. függvény

A programnak ezt a függvényt kell meghívnia elsőként, mert ez ellenőrzi a hardverkulcs jelenlétét. Mindazonáltal a visszatérési értéket átírva nem lehet kifogni a HASP-on.

Bemeneti értékek:	Visszatérési értékek:
BH = 01	EAX = 0 esetén a hardverkulcs nincs jelen
BL = LPT kapu	EAX = 1 esetén jelen van.

1.5.2 HaspCode( ) 2. függvény

Ezt a függvényt többnyire közvetlenül az IsHasp( ) után hívják meg. A jelszó1 és a jelszó2 a hardverkulccsal folytatott adatcserében használatos.

Bemeneti értékek:	Visszatérési értékek:
BH = 02	EAX = kód1
BL = LPT kapu	EBX = kód2
EAX = magkód	ECX = kód3
ECX = jelszó1	EDX = kód4
EDX = jelszó2	

1.5.3 ReadWord( ) 3. függvény

Egy duplaszót (dword-öt) olvas a HASP memóriából. Az olvasás címét az EDI-ben találhatjuk.

Bemeneti értékek:	Visszatérési értékek:
BH = 03	EBX = Beolvasott adat
BL = LPT kapu	ECX = állapot 0 – Helyes egyébként hiba történt
ECX = jelszó1	
EDX = jelszó2	
EDI = cím	

1.5.4 WriteWord ( ) 4. függvény

Egy duplaszót ír a HASP memóriába. Az írás címe az EDI-ben található.

Bemeneti értékek:	Visszatérési értékek:
BH = 04	ECX = állapot 0 – Helyes egyébként hiba történt
BL = LPT kapu	
ECX = jelszó1	
EDX = jelszó2	
EDI = cím	

A HASP ezeken kívül meg számos függvényt tartalmaz (pl.: HaspStatus( ), HaspID( ), ReadBlock( ),WriteBloc( )), amik a megfelelő kezekben nagyszerű fegyver lehet a cracker-ekkel szemben. Ha azonban a programozók egyszerűen csak e védelemre hagyatkoznak, és a függvények hívása után mindössze a visszatérési értékek helyességének ellenőrzésével foglalkoznak, nem igazán szállhatnak szembe a támadókkal. A HASP legnagyobb hátulütője a védelem többletköltsége, így használata csak olyan programoknál ajánlott, melyek költségesek, és terjesztésük nem az interneten történik.

2 Nyomkövetők és töréspontok elleni védelem

A forráskód birtokában a programok védelmének eltávolítása még a nyelv ismerete nélkül sem okoz különösebb problémát a cracker-eknek, de hibás az a feltételezés, hogy a forráskód ismerete nélkül ne lehetne a védelem működésére rájönni. Van három eszköz a cracker-ek kezében, amivel pontos képet kapnak a program működéséről. Ezek a visszafejtők, a visszafordítók és a nyomkövetők. (Disassembler, Decompiler, Debugger).

2.1 Visszafejtők

A visszafejtők feladata a lefordított programok visszafordítása Assembly nyelvre. Előnyük, hogy mindig Assembly nyelv a végeredmény, ezért használóinak nem kell más nyelvet ismerniük. Az eredmény persze függ a visszafejtő tudásától. Észlelhet speciális programrészeket, string-eket, táblázatokat, API hívásokat, amik mind megkönnyítik a munkát. Ilyen híres visszafejtő például a WinDasm.

2.1.1 Visszafejtést gátló makrók

A visszafejtés megnehezítésére használhatunk rövid kódrészleteket, makrókat, amik megnehezítik a visszafejtést, sőt néha még hibát is okozhatnak. Ráadásul még a nyomkövetést is megnehezítik.

2.1.1.1 Megtévesztő makrók a kódban

Könnyen megtéveszthetjük a hackert, ha olyan kódrészleteket helyezünk el a biztosítani kívánt kódban, amik felesleges ugrásokat tartalmaznak, vagy olyan bonyolult kódokat tartalmaznak, amik nem befolyásolják a program futását, csak megtévesztik a cracker-t. Lehet ez olyan kódrészlet is, ami úgy tűnhet, mintha itt valósítanánk meg a védelmet. Példa az ugrásra:

```
[...]
jump1:          call  jump2 ;Barmi lehet itt
                db    0ffh ;Barmi lehet itt
                inc   dword ptr [esp] ;Barmi lehet itt
                jmp   $+4 ;Barmi lehet itt
                int   20h ;Barmi lehet itt
                ret

jump2:          push  ebp ;Barmi lehet itt
                mov   ebp,esp ;Barmi lehet itt
                sub   esp, 80h ;Barmi lehet itt
                jnz   $+3 ;Barmi lehet itt
                db    0ffh ;Barmi lehet itt
                add   dword ptr [esp+84h], 01 ;Barmi lehet itt
                jnz   $+4 ;Barmi lehet itt
                db    68h, 58h ;Barmi lehet itt
                mov   esp, ebp ;Barmi lehet itt
                pop   ebp ;Barmi lehet itt
                ret

jump3:          call  jump2 ;Barmi lehet itt
                db    0ffh ;Barmi lehet itt
                sub   dword ptr [esp], -02 ;Barmi lehet itt
                jnz   $+4 ;Barmi lehet itt
                db    8dh, 87h ;Barmi lehet itt
                jmp   $+4 ;Barmi lehet itt
                int   20h ;Barmi lehet itt
                ret
[...]
```

2.1.1.2 Nyomkövetést is megnehezítő makrók a kódban

Persze minél több és minél bonyolultabb makrót helyezünk el, annál több időt vesz igénybe, de számolnunk kell azzal is, hogy ezek lassítják a programunk futását, és növelik a méretét is. A megértést nehezítő makrók kijátszhatók a nyomkövetők segítségével, ez csak idő kérdése. Van ugyan néhány módszer ezek ellen is. Egy hasznos módszer, aminek a működését nehéz észrevenni, ha jól elrejtjük: a program elején eltároljuk a rendszeridőt, és a biztosítania kívánt kódban elhelyezünk egy ellenőrzést, ami megvizsgálja mennyi idő telt el az indítás óta. Ha ez az érték több mint, ahogy egy lassú számítógépen is lefutna ez a kódrész, biztos, hogy valahol megállították a program futását.

Néhány módszer Assembly nyelven:

```
anti_dis1       macro abc
                push  offset $+8
                ret
                int   20h
                endm
```

```
anti_dis2       macro abc
                push  offset $+16
                push  offset $+8
                ret
                int   20h
                ret
                int   20h
                endm
```

```
anti_dis3       macro abc
                call  jump1
                ret
                jmp   $+6
                db    0c7h, 45h, 1ah, 64h
                push  eax
                push  ebx
                call  $+9
                db    8dh, 0b5h, 0d4h, 66h
                pop   ax
                cmp   esi,esi
                pop   bx
                jnz   $+5
                pop   eax
                jmp   $+5
                db    0e8h, 94h, 0d9h
                call  jump3
                jmp   esp
                cmp   edi,edi
                pop   ebx
                xchg  eax,ebx
                jnz   $-11
                jmp   koniec
                db    0c7h, 85h, 0efh
```

2.2 Visszafordítók

A visszafordítók eredeti nyelvre fordítják vissza az alkalmazást. A nyelvek ismerete szükséges, ezért nem olyan általános, mint a visszafejtő, de nem is minden nyelvhez lehet visszafordítót találni. A mai köztes kódra fordító nyelvek közül szinte mindegyikhez találhatunk, például Java-hoz a DJ Java Decompiler, .NET-hez a .NET Reflector remek választás. Többnyire még a megjegyzéseket is visszahozzák.

2.2.1 Védekezés a visszafordítás ellen

2.2.1.1 Védekezés a visszafordítás ellen átnevezéssel

Az egyik legegyszerűbb módszer. Találhatunk hozzá programot is, de saját magunk is írhatunk egyet, mivel nem túl bonyolult. A módszer alapja, hogy minden azonosítót átnevezzük megjegyezhetetlen, egymásra hasonlító nevekre. Ez kellően sok azonosító esetén hatalmas káosz teremt, és nagyban megnehezíti a program működésének megértését.

2.2.1.2 Védekezés a visszafordítás ellen programok segítségével

Léteznek programok, amik egyszerű és biztonságos módot ajánlanak a visszafordítás ellen. Ilyen például a .NET Reactor, ami összekavarja C#, VB.NET, Delphi.NET, J#, MSIL nyelveken írt .NET assembly-einket natív gépi kóddal, aminek eredménye képen egy natív falat épít a cracker-ek elé, aminek áttörése komoly gondot jelenthet.

2.3 Nyomkövetők

A nyomkövető lehetővé teszi a program futásának nyomon követését lépésről lépésre, és a fontosnak vélt programsornál akár meg is állíthatja azt. Manapság el sem tudnánk képzelni a munkát nyomkövetők nélkül. Nélkülük nehéz lenne a program szemantikai hibáinak felderítése. Minden magasabb szintű nyelv rendelkezik a saját nyomkövetőjével (csak néhányat említve: C++, C#, Java), viszont a forráskód hiányában a futtatható file-t (vagy akár DLL-t is) debuggolni csak Assembly nyelven lehet, ami persze nem okoz gondot a cracker-eknek. Minél magasabb szintű a nyelv annál nehezebb a nyomkövetése, de annál nehezebb hathatós védelmet is írni rajta. A legújabb nyelvek (pl.: a C# és a .NET nyelvek nagy része) már nem rendelkezik az úgynevezett inline asm-mel sem, azaz nincs lehetőség a forráskódba ágyazni Assembly kódot. És mivel az Assembly nyelv a legszerencsésebb választás (, sőt magasabb szintű nyelvek esetén többnyire nem is lehetséges), a biztonsági elemek kódolására, egyre nehezebb feladat hárul a programozókra.

A program az indítása után ellenőrizheti, hogy fut-e nyomkövető, nyomon követik-e a futását. Ha igen, felléphet ez ellen. A legegyszerűbb, ha hibaüzenetet ad és kilép, bár a hibaüzenet küldése megkönnyíti a cracker munkáját. A hibaüzenet hívására ugyanis töréspontot lehet rakni, így a kalóz megtudhatja, hogy a program mely része foglalkozik a védelemmel, és a hívását megkeresve, visszafele lépkedve pontosan meg is találhatja és eltávolíthatja azt. Ezért célszerű, ha a program jelzés nélkül lép ki, vagy valamilyen hibát generál és lefagy, vagy akár a nyomelemzőt is leállíthatja, vagy hibás működést válthat ki benne.

2.3.1 SoftICE

A Compuware fejlesztésében jelent meg. A legjobb és legkedveltebb nyomkövető DOS környezetben. Később Windows 9x és NT rendszereken futó változata is elkészült. Éppen ezért sok cracker kedvence ez a program, és széles körben elterjedt.

2.3.1.1 SoftICE beállítások

Mielőtt a SoftICE használatába kezdenénk, engedélyezni kell a Windows API hívásokat. Ezt winice.dat file-ban tehetjük meg. A legfontosabb API hívásokat tartalmazó könyvtárak a kernel32.dll és a user32.dll. A SoftICE-ból elérhetővé válik például a MessageBoxA API hívás, amire törléspontot lehet elhelyezni.

A függvényelérés engedélyezésének van egy másik lehetősége is. A SoftICE betöltő menüjében lehetőség van további könyvtárak hozzáadására és meglévők törlésére.

2.3.1.2 SoftICE Billentyűparancsok

A SoftICE eléréséhez bármikor használhatjuk a CTRL + D billentyűkombinációt. F10 billentyűvel kezdhetünk a kérdéses program nyomkövetésébe, F8 billentyűvel behatolhatunk a hívások szintjére is. Ha egy API híváshoz töréspontot rendeltünk a SoftICE megáll a hívás elején és az F11 hatására egy RET utasítást hajt végre a hívás helyett.

A BPX[API hívás vagy cím] parancs egy töréspontot helyez el a megadott API hívásnál, vagy címnél.

A BPR[cím1 cím2] kapcsoló a két cím közötti teljes memóriarészt törésponttal látja el. Ha bármilyen program megpróbál írni, vagy olvasni erről a területről a SoftICE megállítja a futását.

A BMP [cím] kapcsoló egy töréspontot rendel a megadott memóriacímhez. Működése megegyezik a BPR kapcsolóval írás olvasás esetén.

Az x kapcsoló egy nyomkövetési töréspontot helyez el közvetlenül a processzor nyomkövető regisztereibe (debug register). Ekkor INT 3h megszakítás már nem kell elhelyezni a címen . Az ilyen töréspontot sokkal nehezebb felfedezni.

2.3.1.3 SoftICE Megjelenítési parancsok

Néhány fontosabb parancs:

d[cím]: A memória tartalmát jeleníti meg DWORD egységekben a megadott címtől.
ed[cím]: A megadott címtől szerkeszthetjük a memória tartalmát
r[regiszterérték]: A regiszterek értékeinek megváltoztatására szolgál
s[cím1 cím2 string vagy byte1, byte2…]: A megadott címek között keres egy megadott karakterláncot vagy byte-sorozatot.
code on: Az utasítások gépi kódját jeleníti meg
wf: A társprocesszor regisztereit mutatja meg
exp: Megjeleníti az elérhető függvényeket
address: Programkódot illeszthetük be a megdott címtől
hboot: Újraindítás

2.3.2 Védekezés a SoftICE ellen

2.3.2.1 A SoftICE felderítése a telepítés könyvtárának keresésével

A SoftICE telepítése folyamán meg kell adnunk egy telepítési könyvtárat. Ez alapértelmezésben a [\Program Files\NuMega\SoftICE] könyvtár, aminek a meglétének ellenőrzését könnyen megtehetjük a debugger felkutatása céljából, sőt ezt minden programnyelven egyszerű kivitelezni, és minden Windows verzióban működik. Hátránya természetesen az, ha a cracker nem az alapértelmezett mappába telepíti a nyomkövetőt, a merevlemezen való softice.exe file keresése pedig sok időt venne igénybe. A következő példa C# nyelven mutatja be a könyvtár ellenőrzését, természetesen hasonlóképpen más nyomkövető alapértelmezett könyvtárát is kereshetjük.

```
public static bool SoftIceDetectDirectory()
{
    // A Program Files konyvtar helyenek beolvasasa
    String programFilesPath =
        System.Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles);

    // Ha a SoftICE telepitve van a telepitesi konyvtarai leteznek
    return System.IO.Directory.Exists(programFilesPath + @"\Numega\SoftICE") ||
        System.IO.Directory.Exists(programFilesPath + @"\SoftICE");
}
```

2.3.2.2 A SoftICE felderítése a Registry vizsgálatával

Hatékonyabban találhatunk rá a SoftICE-ra, ha Windows Registry-ben keressünk, mivel a SoftICE a többi Windows-os programhoz hasonlóan számos bejegyzést készít Registry-ben. Így hozzájuthatunk a verziószámhoz, a sorozatszámhoz, a felhasználó nevéhez, és a telepítési könyvtárhoz is.

```
public static bool SoftIceDetectRegistry()
{
    using (
        Microsoft.Win32.RegistryKey regKey =
            Microsoft.Win32.Registry.LocalMachine
            )
    {
        using (
            Microsoft.Win32.RegistryKey run =
                regKey.OpenSubKey(@"Software\NuMega\SoftICE", true)
                )
        {
            if (run != null)
            {
                // Ha run valtozoba null kerul azt jelenti,
                // hogy nincs ilyen bejegyzes -> valoszinu
                // nincs telepitve a SoftICE
                return true;
            }
            else
            {
                // Biztos telepitve van a SoftICE, igy
                // ezen a ponton kinyerheto meg nehany
                // informacio (pl.: verzoi szam)
                return false;
            }
        }
    }
}
```

Ez a megoldás is működik minden Windows változatban, és könnyedén megvalósítható számos nyelven. A információkat két helyen kereshetjük:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall\SoftICE

valamint a

HKEY_LOCAL_MACHINE\Software\NuMega\SoftICE

bejegyzéseknél. Az előbbi C# példában az utóbbit keressük.

Sajnos a felderítések nagy része nem valósítható meg magas szintű nyelvekben, ezért az előbbi példa Assembly változatát is bemutatom. Ebben a programban az ADVAPI32.DLL könyvtár 2 függvénye könnyíti meg a Registry kezelését.

```
.386
.MODEL FLAT,STDCALL
locals
jumps
UNICODE=0
includelib advapi32.lib
include w32.inc
Extrn SetUnhandledExceptionFilter        : PROC
Extrn RegOpenKeyExA                      : PROC          ;függvény az ADVAPI32.DLL-ből
Extrn RegQueryValueExA                   : PROC          ;függvény az ADVAPI32.DLL-ből

.DATA
message1        db "A SoftICE keresese a Registry-ben",0
message3        db "A SoftICE jelen van!",0
message2        db "A SoftICE nincs jelen!",0
message4        db "A SoftICE a kovetkezo mappaba telepult: ",0
message5        db "A SoftICE verzioja: ",0
delayESP        dd 0                     ;az ESP mentese
previous        dd 0                     ;az ESP regiszter ide menti
                                         ;az elozo SEH szolgaltatast
result  dd 0
size    dd 5                             ;meret a verzioszamhoz
size2   dd 80h                           ;meret a telepitesi mappahoz
subkey  db "Software\NuMega\SoftICE\",0
current_ver     db "Current Version",0
install_dir     db "InstallDir",0
data_buffer     db 200 dup (0)           ;ide ketul a verzioszam
value_buffer    db  20 dup (0)
data_buffer2    db 200 dup (0)           ;ide meg a telepitesi mappa

.CODE
Start:

;-------------------------------------------------------------------
;Hiba esetere a SEH beallitasa
;-------------------------------------------------------------------
        mov  [delayESP], esp
        push offset error
        call SetUnhandledExceptionFilter
        mov  [previous], eax
;-------------------------------------------------------------------
        push offset result               ;ide kerul az eredmeny
        push 20016h                      ;eleres tipusa
        push 0
        push offset subkey               ;string a reszkulcs nevevel
        push 80000002h                   ;HKEY_LOCAL_MACHINE = 80000002 ahol
                                         ;a reszkulcsot megnyirjuk
        call RegOpenKeyExA               ;megnyitja a kivant elerest a
                                         ;regiszterekben, es menti az eredmenyt
        test eax,eax
        jnz  notfound                    ;hiba eseten ugras

        push offset size                 ;az adattar merete
```

```
53          push offset size                ;az adattar merete
54          push offset data_buffer         ;az adattar cime
55          push offset value_buffer        ;az ertektar cime
56          push 0
57          push offset current_ver         ;az osszehasonlitasi ertek neve
58          push result                     ;ide mentjuk az eredmenyt
59          call RegQueryValueExA           ;a SoftICE verzioszamat olvassa be
60          test eax,eax
61          jnz  notfound                   ;hiba eseten ugras
62
63          push offset size2               ;az adattar merete
64          push offset data_buffer2        ;az adattar cime
65          push offset value_buffer        ;az ertektar cime
66          push 0
67          push offset install_dir         ;az osszehasonlitasi ertek neve
68          push result                     ;ide mentjuk az eredmenyt
69          call RegQueryValueExA           ;a SoftICE telepitesi mappajat olvassa be
70          test eax,eax
71          jnz  notfound                   ;hiba eseten ugras
72
73          inc  al                         ;1-el noveli az AL
74                                          ;(ha nem volt hiba -> EAX=0)
75          jmp  short ok
76
77  notfound:
78          xor eax,eax                     ;nullazza EAX-et, jelezve hogy hiba tortent
79  ok:
80          push eax                        ;menti az eredmenyt
81  ;---------------------------------------------------------------------
82  ;Visszaallitja az elozo SEH szolgaltatast
83  ;---------------------------------------------------------------------
84          push dword ptr [previous]
85          call SetUnhandledExceptionFilter
86
87          pop  eax                        ;visszanyeri az eredmenyt
88          test eax,eax
89          jnz  jump                       ;ha minden rendben ugras
90
91  continue:
92          call MessageBoxA,0, offset message2,\
93          offset message1,0               ;hibauzenet kiirasa
94          call ExitProcess, -1
95  jump:
96          call MessageBoxA,0, offset message3,\
97          offset message1,0               ;a SoftICE jelen van
98          call MessageBoxA,0, offset data_buffer,\
99          offset message5,0               ;verzioszam kiirasa
100         call MessageBoxA,0, offset data_buffer2,\
101         offset message4,0               ;a telepitesi mappa kiirasa
102         call ExitProcess, -1
103 error:          ;hiba eseten elinditja az uj SEH-t
104         mov  esp, [delayESP]
105         push offset continue
106         ret
107 ends
108 end Start
```

2.3.2.3 A SoftICE felderítése az INT 3h hívással

Ez a megoldás az egyike a legismertebb nyomkövetés elleni megoldásoknak, és külön érdekessége, hogy kihasználja a SoftICE egyik hátsó bejáratát. Használható minden Windows változatában, de az XP-ben (Sp2-tol) és Vistában az INT 3h hívás bonyodalmakat okoz, és programleálláshoz vezethet.

A módszer lényege, hogy az INT 3h hívás előtt az EAX regiszterbe 04h-t, az ESP-be pedig a 4243484Bh ("BCHK") értéket kell helyeznünk. Visszatéréskor az EAX-ben a SoftICE jelenlétekor nem a 4-es értéket találjuk.

A módszert gyakran használják különböző tömörítőprogramokban, széles körben elismert. Ezért a cracker-ek is könnyedén eltávolítják, de ha megfelelően alkalmazzuk, kifoghat a tapasztaltabb cracker-eken is.

Ezen módszer megvalósítása ugyan már csak Assembly nyelven lehetséges, van rá lehetőség, hogy olyan programokba építsük be, amelyek nem adnak lehetőséget Assembly kód beágyazására (pl.: a C#). Ebben az esetben az Assembly kódot DLL-be szervezhetjük, vagy ami még előnyösebb, a natív Assembly-ből egy C++/CLI wrapper segítségével managed DLL készíthető, ami már .NET alatt is használható. A C++ metódus megvalósítása a következő:

```
//Igaz, ha a SoftICE fut,
//Hamis, ha nem fut.
__inline bool IsSICELoaded()
{
    _asm {
        push ebp            // EBP elmentese a verembe
        mov ebp,'BCHK'      // 'BCHK' (4243484Bh) EBP-be
        mov eax,4           // 4h masolasa EAX-be
        int 3               // INT 3h hivas
        cmp al,4            // AL-be 4 van-e?
        setnz al            // ha != 0 (ZF=0)
        pop ebp             // EBP visszaallitasa
    }
}
```

2.3.2.4 A SoftICE felderítése a CreateFileA API hívással

Ez a módszer a SoftICE felderítésének legismertebb formája. Használható még VxD és Sys meghajtók felkutatására is. A módszer alapelve: meg kell próbálni megnyitni egy file-t, amelynek neve megegyezik az aktív VxD, illetve Sys file nevével. Figyelni kell, hogy a memóriában megtalálható-e, és a megnyitás típusa OPEN_EXISTING volt-e. A CreateFileA hívása után, ha a megnyitás sikeres volt, az EAX regiszterben található visszatérési érték nem 0FFFFFFFFh (-1).

Megvalósítása egyszerű magas szintű nyelveken is, ennek köszönhető ismeretsége is. Ennek köszönhetően a cracker-ek körében is igen ismert, és eltávolítása nem okoz gondot még a kezdőknek sem. Alkalmazása csak más védelemmel együtt ajánlott. A Windows 9x operációs rendszerek esetén a [\\.\SICE] file megnyitásával NT-s rendszerek esetén pedig [\\.\NTICE] file megnyitását kell tesztelnünk.

A módszer C nyelven megvalósítva:

```
#include "stdafx.h"
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
#include <iostream>

using namespace std;

bool IsSoftIce95Loaded()
{
    HANDLE hFile;

    //"\\.\SICE" Win9x-ben
    hFile = CreateFile("\\\\.\\SICE",
                       GENERIC_READ | GENERIC_WRITE,
                       FILE_SHARE_READ | FILE_SHARE_WRITE,
                       NULL,
                       OPEN_EXISTING,
                       FILE_ATTRIBUTE_NORMAL,
                       NULL);

    if( hFile != INVALID_HANDLE_VALUE )
    {
        CloseHandle(hFile);
        return true;
    }
    return false;
}
```

```
bool IsSoftIceNTLoaded()
{
    HANDLE hFile;

    //"\\.\NTICE" WinNT-ben
    hFile = CreateFile("\\\\.\\NTICE",
                       GENERIC_READ | GENERIC_WRITE,
                       FILE_SHARE_READ | FILE_SHARE_WRITE,
                       NULL,
                       OPEN_EXISTING,
                       FILE_ATTRIBUTE_NORMAL,
                       NULL);

    if( hFile != INVALID_HANDLE_VALUE )
    {
        CloseHandle(hFile);
        return true;
    }
    return false;

}

int _tmain(int argc, _TCHAR* argv[])
{
    if( IsSoftIce95Loaded() )
        cout<<"Win9x SoftICE aktiv a memoriaban!";
    else if( IsSoftIceNTLoaded() )
        cout<<"WinNT SoftICE aktiv a memoriaban!";
    else
        cout<<"SoftIce nincs jelen!";
    return 0;
}
```

2.3.2.5 A SoftICE felderítése a NmSymlsSoftICELoaded hívással

A SoftICE tartalmaz egy nmtrans.dll könyvtárat, melyben segítségükre lehet az NmSymlsSoftICELoaded függvény. Meghívva a függvényt ellenőrizhetjük, hogy a SoftICE a memóriában tartózkodik-e.

Magas szintű nyelveken sem ütközik nehézségekbe a módszer alkalmazása, mégis álljon itt most egy Assembly nyelven megvalósított példa:

```
1  [...]
2  SOFTNTsym      db "C:\NTICE\nmtrans.dll",0
3  SOFTsym        db "NmSymIsSoftICELoaded",0
4  [...]
5  call LoadLibraryA, offset SOFTNTsym          ;mntrans.dll betoltese
6  test eax,eax
7  jz   exit                                    ;hiba eseten vege
8  call GetProcAddress, eax, offset SOFTsym     ;NmSymIsSoftICELoaded betoltese
9  test eax,eax
10 jz   exit                                    ;hiba eseten vege
11 call eax                                     ;NmSymIsSoftICELoaded hivasa
12
13 exit:
14
15 test eax,eax                                 ;EAX != 0, a SoftICE aktiv
16 [...]
```

Először be kell tölteni az nmtrans.dll-t a memóriába, erre használható a LoadLibraryA API hívás. GetProcAddress API hívással megkereshetjük a memóriában a címét, majd meghívjuk a függvényt. A SoftICE fut, ha a visszatérési érték 0-tól különbözik.

2.3.2.6 A SoftICE felderítése INT 2Fh hívással

Ez egy meglehetősen ritkán alkalmazott módszer, mivel megvalósítása meglehetősen bonyolult, magas szintű nyelveken lehetetlen. Ennek köszönhetően megbízható védelmet kölcsönöz programunknak, de persze a többi védelemmel egyetemben ez sem kijátszhatatlan. Működése közben az INT 2Fh hívást és a GET DEVICE API ENTRY POINT hívást alkalmazza, és csak az ES és a DI regiszterek 0 értéke mellett használható. A BX a 0202h értéket tartalmazza, mely a VxD azonosítója a SoftICE-nál, ha a VxD működik az ES és a DI nem 0 értéket tartalmaz majd.

A megvalósítás nehézségét az adja, hogy a Windows alkalmazásokban, az INT 2Fh megszakítást nem használhatjuk anélkül, hogy hibát ne kapnánk az operációs rendszertől. DOS operációs rendszer esetén még használható volt ez a fajta megszakítás, de a 32 bites Windows-ok már nem engedélyezik ring3-ban, csak ring0-ban futó alkalmazások részére. Ennek biztonsági okai vannak, főleg a vírusok kiszorítása miatt vezette be Microsoft. És ha kiderül egy új módszer ennek kijátszására, hamarosan elkészül a javítás, és többet nem

alkalmazható, sőt programhibához vezet. Így megfelelő odafigyeléssel alkalmazható csak a SoftICE felderítésére.

Egy módszer ring0-ba váltáshoz:

A következő Assembly nyelven írt program először megkeresi a memóriában az INT 5h szolgáltatás címét, és helyettesíti egy új szolgáltatás címével, majd azonnal meg is hívja azt. Ezáltal rögtön ring0-ba kerül. Ebben az állapotban hívja meg a VxD-t, és beállítja az új INT 68h szolgáltatást. Ekkor az INT 5h szolgáltatás véget ér, és a program újra ring3-as szinten fut tovább. Az INT 68h hívásakor az AH regiszter értéke 43h, aminek a beállítása nem is lenne szükség, de remek lehetőség elhitetni a támadóval, hogy így szeretnénk felderíteni a nyomkövetőt.

Az új szolgáltatást az INT 68h-val hívhatjuk meg, és ha a program felfedezi a futó SoftICE-t a memóriában, beállít egy jelzőt. Ezután szintén meghívja a ring0-ban futó INT 5h megszakítást, és visszaállítja az INT 68h eredeti állapotát, majd visszatér ring3-ba. Legvégül a jelző vizsgálatával kapunk információt a SoftICE jelenlétéről.

```
1  .386p
2  .MODEL FLAT,STDCALL
3  locals
4  jumps
5  UNICODE=0
6  include w32.inc
7
8  Extrn SetUnhandledExceptionFilter : PROC
9  Interrupt       equ 5   ;3as megszakitas neheziti a debuggolast
10
11 .DATA
12 message1        db "Int 2Fh hivassal SoftICE felderitese",0
13
14 message2        db "Debugger nem aktiv",0
15 message3        db "Debugger aktiv",0
16 delayESP        dd 0    ;az ESP regiszter mentese
17 previous        dd 0    ;az ESP regiszter menti az elozo SEH cimet
18 mark    db 0    ;jelzo, ha = 1 -> SoftICE aktiv
19
20 .CODE
21 Start:
```

```
;---------------------------------------------------------------------
;Hiba esetere SEH beallitasa
;---------------------------------------------------------------------
mov  [delayESP], esp
push offset error
call SetUnhandledExceptionFilter
mov  [previous], eax
;---------------------------------------------------------------------
push edx
sidt [esp-2]                    ;IDT beolvasasa
pop  edx
add  edx, (Interrupt*8)+4       ;INT 5h vektoranak beolvasasa

mov  ebx,[edx]
mov  bx,word ptr [edx-4]        ;INT 5h (regi szolg) cimenek beolvasasa
lea  edi,InterruptHandler
mov  [edx-4],di
ror  edi,16                     ;beallitja INT 5h uj szolgaltatasat
mov  [edx+2],di

push ds                         ;regiszterek mentese
push es
int Interrupt                   ;ring0-ba ugras, INT 5h uj szolgaltatashoz
pop  es                         ;regiszterek visszaallitasa
pop  ds

push ds                         ;regiszterek mentese
push es
mov  ah, 43h
int  68h                        ;az uj INT68 hivasa

stc                             ;jelzo beallitasa az INT 68h hiba estere
int  Interrupt                  ;ring0-hoz ugrik (az uj INT 5h-hoz)
pop  es                         ;regiszterek visszaallitasa
pop  ds

mov  [edx-4], bx                ;eredeti INT 5h beallitasa
ror  ebx,16                     ;beallitasa
mov  [edx+2], bx
;---------------------------------------------------------------------
; elozo SEH visszaallitasa
;---------------------------------------------------------------------
push dword ptr [previous]
call SetUnhandledExceptionFilter
;---------------------------------------------------------------------
cmp  byte ptr mark, 1           ;jelzo vizsgalata
jz   jump                       ;ha SI jelen van -> vege
continue:
call MessageBoxA,0, offset message2, offset message1,0
call MessageBoxA,0, offset message2, offset message1,0
call ExitProcess, -1
jump:
call MessageBox, 0, offset message3, offset message1,0
call ExitProcess, -1
error:                                ;hiba eseten uj SEH inditasa
mov  esp, [delayESP]
push offset continue
ret
```

```
;-------------------------------------------------------------------
;Az uj INT 5h szolglatatas (ring0-ban fut)
;-------------------------------------------------------------------
InterruptHandler:
pushad                          ;regiszterek mentese
jc   uninstall                  ;jelzo vizsgalata, igen->INT68h eltavolitas
mov  eax, 68h                   ;megszakitas szama (uj szolgaltatase)
mov  esi, offset HookInt68      ;uj szolgaltatas cime
db   0cdh, 20h                  ;Hook_V86_Int_Chain meghivasa
dd   000010041H                 ;VxD fuggveny
;      VMMCall Hook_V86_Int_Chain
mov  eax, 68h                   ;megszakitasszam (uj szolgaltatase)
mov  esi, OFFSET HookInt68      ;uj szolgaltatas cime
db   0cdh, 20h                  ;VxD fuggveny
dd   000010080H                 ;Hook_PM_Fault meghivasa
;       VMMCall Hook_PM_Fault
popad                           ;regiszterek visszaallitasa
iretd                           ;visszaugras Ring3-ba
uninstall:                      ;INT 68h szolgaltatas eltavolitasa
mov  eax, 68h                   ;az uj magszakitas szama
mov  esi, OFFSET HookInt68      ;uj szolgaltatas cime
db   0cdh, 20h                  ;UnHook_V86_Int_Chain meghivasa
dd   000010118H                 ;VxD fuggveny
;        VMMCall        UnHook_V86_Int_Chain
mov  eax, 68h                   ;megszakitasszam (uj szolgaltatase)
mov  esi, OFFSET HookInt68      ;uj szolgaltatas cime
db   0cdh, 20h                  ;UnHook_PM_Fault meghivasa
dd   00001011AH                 ;VxD fuggveny
;        VMMCall UnHook_PM_Fault
popad                           ;regiszterek visszaallitasa
iretd                           ;visszaugras Ring3-ba
;-------------------------------------------------------------------
; uj INT 68h szolgaltatas
;-------------------------------------------------------------------
HookInt68:
pushfd                           ;
pushad                           ;regiszterek mentese
xor  di, di                      ;
mov  es, di                      ;es:di nullazasa
mov  bx, 202h                    ;a SICE VxD azonositoja
mov  ax, 1684h                   ;GET DEVICE API ENTRY POINT kodja
int  2Fh                         ;meghivja a DEVICE API ENTRY POINT fvg-t

mov  ax, di                      ;DI mentese
test ax, ax                      ; ax=0 ?
jz   short none                  ;ha igen akkor SI nem aktiv
mov  byte ptr mark, 1            ;jelzo beallitasa <- SI aktiv
none:
popad                            ;
popfd                            ;regiszterek visszaallitasa
ret                              ;visszater a szolgaltatasrol
ends
end Start
```

2.3.3 Egyéb neves nyomkövetők

2.3.3.1 OllyDbg

Az OllyDbg egy erős bináriskód analizáló. Kompatibilis minden 32 bites Windows verzióval, köztük a Vistával is. Nem kernelszintű, ezért lehetőségi korlátozottak, de az egyik legjobban használható debugger ebben a kategóriában. Képes a regiszterek nyomkövetésére, felismeri az eljárásokat, API hívásokat, kapcsolókat, táblázatokat, konstansokat, string-eket és táblázatokat úgy, mint a DLL könyvtárakat és függvényeiket. Fejlesztése a 2.0 verziónál jár és teljesen ingyenes.

2.3.3.2 Syser Debugger

A Syser kernel szintű nyomkövetőt szokás a SoftICE utódaként is emlegetni. Bár a SoftICE a legelterjedtebb és mai napig a legtöbb alkalommal ezt használják cracker-ek, a fejlesztése leállt, és nem mondható túl modernnek. A SoftICE első 1987-es DOS-os verziója után Windows 9x-es és Windows NT-s változatai is megjelentek, de az XP SP2 alatt már nem fut. A Syser ezzel szemben folyamatosan fejlődik jelenleg az 1.96-as verzió a legújabb, amit 2008 márciusától lehet megvásárolni. Számtalan operációs rendszer alatt fut, köztük az XP SP 2-n és Vistán is, bár még csak 32 bites változatban. Támogatja az SMP-t (Symmetric Multi Processors), ami a többprocesszoros rendszerek azon válfaja, melyek esetében több processzor osztozik ugyanazon a memórián. Továbbá rendelkezik egy modern debugger-től elvárható HyperThreading és többmagos processzortámogatással is. Kezelése, így parancsai is teljesen megegyeznek a SoftICE-szal.

Sajnos mivel fejlesztése folyamatosan folyik, nem készült még megfelelő módszer a felderítésére, csak az általános debugger-ellenes módszerek használhatók ellene.

2.3.4 Általános nyomkövető felderítése

Az x86-os processzorok nyomkövetési regisztereket tartalmaznak (DR0-DR7), melyeket felhasználhatunk a nyomkövetők felderítésre. A DR7-es regiszterben alapállapotban, ha nincs nyomkövető a memóriában 400h értéket találunk.

Sajnos a nyomkövetési regisztereket csak ring0-ás állapotban használhatjuk, így a módszer csak VxD vagy Sys meghajtókkal alkalmazhatók, vagy át kell lépnünk ring0 állapotba. A módszer erősségét mutatja, hogy előfordul a SafeDisc egy korábbi verziójában is, és mivel nem használ API hívásokat nagyon nehéz felfedezni, de persze ezt sem lehetetlen. A következő Assembly kódrészlet bemutatja a módszert.

```
1  [...]
2  Interrupt equ 5
3  [...]
4  push edx
5  sidt [esp-2]                    ;IDT a verembe
6  pop  edx
7  add   edx,(Interrupt*8)+4       ;megszakitas vektorat beolvasasa
8
9  mov  ebx,[edx]
10 mov  bx, word ptr [edx-4]       ;megszakitasi cimet beolvasasa
11 lea  edi,InterruptHandler
12 mov  [edx-4],di
13 ror  edi,16                     ;beallitja az uj megszaktas szolgaltatast
14 mov  [edx+2],di
15
16 push ds                         ;regszterek mentese
17 push es
18 int  Interrupt                  ;ring0-ba ugras (uj INT 5h-hoz)
19 pop  es                         ;regszterek visszaallitasa
20 pop  ds
21
22 mov  [edx-4],bx                 ;beallitja az erdeti szolgaltatast (INT 5h)
23 ror  ebx, 16
24 mov  [edx+2],bx
25
26 push eax                        ;eredmeny mentese
27 [...]
```

A módszert mindazonáltal a jobb cracker-ek felderíthetik, ha a DR7-es regiszter DG jelzőjét beállítják töréspont bekapcsolásához, például INT 01h-val, majd kiolvassák a DR6 regiszter BD jelzőjét, melyből kiderül a védelem.

2.3.5 Töréspontok elleni védelem

2.3.5.1 Töréspont felismerése Trap jelzővel

A módszer igen hatékony és nehezen észrevehető, mivel nem használ API hívásokat. Lényegében az Eflags regiszterben állítja be a Trap jelzőt, amivel kivált egy Exception Single Step kivételt, ami meghívja a SEH szolgáltatást. Ennek köszönhetően, ha a programot nyomkövetik a SEH szolgáltatást nem fogja meghívni, amit észrevehetünk. Ha rá is jön a cracker a módszerre, nem tudja utánozni a működést, mivel a nyomelemzők nem adnak rá lehetőséget.

```
1  [...]
2  mark            db 0                    ;volt-e xhandler szolg.
3  [...]
4  Start:
5          call real_start                 ;A program igazi kezdetere ugrik
6  ;-------------------------------------------------------------------
7  ; uj SEH hiba esetere (xhandler)
8  ;-------------------------------------------------------------------
9          inc  mark                       ;mark+1 (nincs nyomkovetes)
10         sub  eax,eax                    ;EAX nullazasa (SEH miatt)
11         ret
12 real_start:
13         xor  eax,eax                    ;EAX nullazasa
14         push dword ptr fs:[eax]         ;eredeti SEH mentese
15         mov  fs:[eax], esp              ;uj SEH beallitasa
16         pushfd                          ;flag-ek mentese
17         or byte ptr [esp+1], 1          ;Eflags Trapjelzojet beallitja
18         popfd                           ;visszaallitja a flag-eket
19         nop                             ;xhandler meghivasa
20         pop  dword ptr fs: [eax]
21         pop  ebx                        ;elozo SEH visszaallitasa
22         dec  mark                       ;mark-1==-1 ->  nomelemzo van
23 [...]
```

2.3.5.2 Töréspont felismerése CRC-ellenőrzéssel

A CRC-ellenőrzés az alkalmazás változásának kimutatására használható leghatékonyabb eszköz, mivel nem tartalmaz API hívásokat. A töréspont felismerése a kezdeti és a futás közben végzett CRC ellenőrzéssel történik, amit a program több részén is célszerű alkalmazni. Az alábbi C# osztály a CRC ellenőrzőösszeg számítását végzi. A CrcTable nevű táblázatot az 1-es számú melléklet tartalmazza.

```
using System;
using System.IO;

namespace CRC32
{
    public sealed class Crc32
    {
        private readonly static uint CrcSeed = 0xFFFFFFFF;
        public uint Crc { get; private set; }

        public static uint GetFileCRC32(string path)
        {
            if (path == null)
                throw new ArgumentNullException("path");

            Crc32 crc32 = new Crc32();
            byte[] buf = new byte[4096];
            int len = 0;
            using (FileStream fs = new FileStream(path, FileMode.Open))
            {
                while ((len = fs.Read(buf, 0, buf.Length)) != 0)
                {
                    crc32.Update(buf, 0, len);
                }
            }
            return crc32.Crc;
        }
```

```
        public void Update(byte[] buf, int off, int len)
        {
            if (buf == null)
            {
                throw new ArgumentNullException("buf");
            }

            if (off < 0 || len < 0 || off + len > buf.Length)
            {
                throw new ArgumentOutOfRangeException();
            }

            Crc ^= CrcSeed;

            while (--len >= 0)
            {
                Crc = CrcTable[(Crc ^ buf[off++]) & 0xFF] ^ (Crc >> 8);
            }

            Crc ^= CrcSeed;
        }
    }
}
```

3 CD és DVD védelem

Kész programok terjesztése többnyire valamilyen adathordozón történik, leggyakrabban CD-n vagy DVD-n, újabban pedig BD-n (Blu-ray Disc). Ezek másolás elleni védelme az eladás szempontjából nagyon fontos, mivel az illegális másolatok száma manapság meghaladja 70-80%-ot.

A legtöbb ellenőrző eljárás egy GetDriveTypeA API hívással kezdődik, hogy felkutassa az optikai meghajtót. A legegyszerűbb védelmek, csak a lemez címkéjét ellenőrzik. A modernebb változatok, már olyan címkéket adtak a lemezeknek, amik másoláskor hibát okoznak, de a mai írók már megbirkóznak ezzel a feladattal is. Szokásos fogás a lemez megvágásakor (pl.: DVD9 vágása DVD5-re) a valószínűleg eltávolított file-ok (Intók, átvezető videók) keresése.

Népszerű megoldás a futás közbeni véletlenszerű tesztek elvégzése, mivel ezek eltávolítása szinte lehetetlen. Többnyire játékoknál alkalmazzák.

3.1 Védelmi programok

3.1.1 CD-Cops / DVD-Cops

A Link Data Security kiadásában 1996-ban megjelent CD-Cops az első CD védelmi rendszer, ami a lemez geometriai adottságait használja a másolatok kiszűrésére. Könnyen felismerhető az indításkor megjelenő ablakról, a CDCOPS.DDL könyvtárról, és a lemezen található .GZ_ és .W_X kiterjesztésű file-okról.

A védelem a meghajtóban lévő lemezen megkeresi az elő és az utolsó blokkot, és kiszámolja a köztük lévő szögtartományt. Az eredeti lemez tartalmaz egy 8 byte-os kódot a helyes szögtartománnyal és összeveti a számítottal. A másolat esetén eltérnek a szögek, és a program nem indul el.

Saját védelmére időzítőt használ, mely ellenőrzi, mennyi ideje fut az ellenőrzés, és ha túl sok idő telik el programhibát vált ki. Ennek köszönhetően nagyon nehéz nyomkövetni a működését. Ezek mellett több ellenőrzőösszeget is tartalmaz, ami a kód manipulálását teszi lehetetlenné.

1998-ban megjelent DVD-re szánt változat DVD-Cops néven, ami az első DVD védelem volt a világon. Az alapelv viszont ugyanaz volt, és semmilyen jelentősebb fejlesztést nem tartalmazott.

Használata enciklopédiák, szótárak és üzleti alkalmazások esetén figyelhető meg, játékoknál nem. Ez főleg a védelemben felfedezett gyengeségének köszönheti, mivel a CD-Cops visszafejthető az eredeti lemez nélkül is. Sajnos a programozók benne hagyták a teljes kódot a programban, aminek köszönhetően a cracker-ek viszonylag hamar megírták hozzá a dekódert.

3.1.2 SafeDisc

Első változata a C-Dilla kiadásában jelent meg, később Macrovision Corporation kiadásában találkozhatunk vele. Napjaink legnépszerűbb CD/DVD védelmi módszere, amit mi sem bizonyít jobban, minthogy a játékok 45%-át ezzel a védelemmel látják el.

Ugyan a korai verziókat nem nehéz eltávolítani (a kiadás után 1 héttel már megjelent a kalózverzió), de a V2.9-es változattól kezdődően a SafeDisc-kel gyártott lemezeket sokkal nehezebb lett másolni, sőt speciális íróra is szükség van a "weak sectors" és az "odd data" formátumok írásához. Tartalmaz továbbá emulátorok (ilyenek a Daemon Tools és az Alcohol 120%) elleni védelmet, és manapság feketelistások még azok a programok is, amik ezek álcázására szolgálnak (pl. a CureROM). Komoly gondot fordítottak a nyomkövetők felismerésére, a SoftICE ellen több fogást is beépítettek. Bár az újabb változatokból ki kellett venni néhány módszert, ugyanis nem működtek volna XP SP2 és Vista rendszereken.

Mindezek ellenére léteznek programok, melyek lehetővé teszik a SafeDisc-kel védett programok futtatását az eredeti lemez pontos másolatáról. Sőt az UnSafeDisk nevű program segítségével az eredeti lemez nélkül is lehetséges a dekódolás.

Folyamatos fejlesztés alatt áll, manapság a 4.81 verziónál tart.

3.1.3 SecuROM

A SecuROM is egy igen széleskörűen használt CD/DVD másolásvédelmi program, a Sony DADC fejlesztése. Megakadályozza az általa védett lemez otthoni és profi másolását és véd a visszafejtés ellen is. Nagy a hasonlóság a SafeDisc-kel a korai változatokban, amikor még a védelmét könnyen ki lehetett játszani, mivel az adatok csak részben voltak kódolva, és egy memóriakiíró programmal ki lehetett menteni dekódolás után az .EXE file-t. A SecuROM a 4.7-es verziótól lett híres, komoly ellenfele lett a SafeDisc-nek. Felszerelték emulátorok elleni védelemmel, a 4.84-es verziótól úgynevezett trigger függvényeket is tartalmaz, melyek lehetővé teszik a fejlesztőnek, hogy testre szabják a hitelesítési eljárást, a védelem is átkerült az alkalmazás kódja és az operációs rendszer közé, és a rendszerhívásokhoz hasonló működést tanúsít.

```
if (GetCurrentDate() == '13-32-2999') then
    WorkCorrectly()
else
    PreventProgramUse()
end if
```

Nyilvánvaló, hogy a normál GetCurrentDate() függvény sosem tér vissza ’13-32-2999’ értékkel. Azonban mivel a SecuROM módosította a hívás eredményét, és az alkalmazás ellenőrizheti ez idő alatt a vedelem jelenlétét. Ha védelem el lett távolítva, a hívás egy hasonló értékkel tér vissza, lehetőséget adva az alkalmazásnak, hogy hibaüzenetet adjon, vagy leállítsa a működést. A trigger függvények elhelyezésére a kódban számtalan lehetőség adott, ezzel is nehezítve, hogy a cracker-ek rábukkanjanak.

A 7.x verzió sajátossága, hogy telepít egy saját ring3-ban futó szolgáltatást, UAService7.exe néven a SecuROM eltávolításra szakosodott programok kivédése céljából.

4 A Vista memória- és programvédelme

2003-ban a Microsoft elindított egy Security Development Lifecycle (SDL) kezdeményezést, melynek célja olyan programok fejlesztése, melyek a biztonsági kockázatok csökkentését hivatottak növelni. Az SDL szigorú és formális tervezési, fejlesztési, tesztelési, bevizsgálási és visszajelzési szabályokat ír elő a sérülékenységek száma és a támadási felület nagyságának csökkentése érdekében. Már korábbi rendszereken látszik a SDL megközelítés, de a Vista az első, amelyet teljesen a SDL szempontok alapján készítettek. A memória és a programok védelme egyébként is sarkalatos pontja egy operációs rendszernek, így a Vista külön hangsúlyt is helyez erre a két területre. Kombinálva ezeket a védelmeket, mindenidők legbiztonságosabb windows operációs rendszerét hozta létre a Microsoft.

4.1 DEP

Komoly támadást jelent, amikor egy program egy nem futtatható kódot erővel futtathatóvá akar tenni. Ehhez kihasznál egy puffer-túlcsordulást és futtatható kódként értelmezhető adatokat ír a verem-, adat-, heap-területként kijelölt memóriába, majd az így betöltött kódnak adja át a vezérlést.

A Data Execution Prevention (adatfuttatás-megelőzés) vagy NoExecute (ne futtasd) egy olyan biztonsági technológia, amely a programok memóriahasználatának helyességét felügyeli. A DEP figyeli a memóriát, és észleli, ha egy program helytelenül akarja használni azt, illegális helyen akar kódot futtatni. A DEP az ilyen helyzeteket nem tudja kijavítani, csak megakadályozza a program futását. A program adatainak eltárolását nem akadályozza, de az illegális helyen tárolt kód futtatásakor közbeavatkozik.

4.1.1 Hardveres DEP

A hardverrel érvényesített DEP olyan architektúrában lehetséges, amelyben a processzor támogatja azt. Szegmens leíró tulajdonságként ehhez hasonló védelem van az x86-os processzorokban is a 80286-os óta. Ez a típusú védelem csak akkor működik, ha egy egész

szegmensre vonatkoztatjuk. Az AMD volt az első, aki az x86-os processzorokban használta a NoExecute bitet (NX) a lineáris címzési módnál. Ez a tulajdonság elérhető az AMD64 processzorokban is emulált üzemmódban, és a jelenlegi Intel x86 processzorokban is, ha a PAE használatban van. Az NX az eredeti AMD-s elnevezés, az Intel terminológiájában XD bitről (eXecute Disable) beszélünk.

Előfordulhat az is, hogy egy legitim program adatterületen futtatna gépi kódot, de a DEP megakadályozná. Az ilyen eseteken, a biztosan nem ártó szándékú programokat meg lehet jelölni, fel lehet ruházni olyan jogosultsággal, hogy az eredetileg nem futtatható kódot is futtassák.

A Windows XP (SP2-től), Windows Server 2003 és Windows Vista változatok elég okosak ahhoz, hogy felismerjék a processzor ezen képességét és kihasználják a hardveres DEP előnyeit, sőt még ennél is okosabbak, mert ha a processzor nem támogatja, akkor emulálják.

4.1.2 Szoftveres DEP

Az NX vagy XD bittől független szoftveres DEP, azaz Safe Structured Exception Handling (SafeSEH, biztonságos struktúrájú kivételkezelés) esetében kicsit bonyolultabban valósítható meg – de nem lehetetlen – a verem, a heap vagy az adatszegmens rendszerszintű figyelése. A Vista esetében a szoftveres DEP a kivételkezelő mechanizmusokon keresztül érvényesül. A SafeSEH követelményeinek eleget tevő programoknak rendelkezniük kell ilyen funkciókkal, és futtatáskor regisztrálniuk kell a saját kivételkezelő eljárásukat. Nem a SafeSEH-megfelelőség szerint készült programok esetén a szoftveres DEP megvizsgálja, hogy maga a kivételkezelő kód futtatható memóriaterületen van-e. Végső soron a hardveres DEP is a kivételkezelésbe torkollik bele, mert az adatterületen történő kódfuttatási kísérlet kivételt generál, amit az operációs rendszer kezel le. Akár hardveres, akár szoftveres DEP-ről van szó, a rendszer nem javítja ki a puffer-túlcsordulásos sérülékenységeket, és nem akadályozza meg memóriaterületek felülírását. Amikor azonban a vezérlés adatterületre kerülne, akkor hardveres esetben könyörtelenül hardveres kivétel generálódik, amelynek értelmezésével le lehet fülelni a goromba ágenseket. Szoftveres DEP esetén a kivételgenerálást az operációs rendszer szoftveresen végzi. Ennek a könyörtelensége, azaz a „jósága” attól függ, mennyire érzékenyen figyel a rendszer a helytelen memóriahasználatra.

4.1.3 DEP opciók a Vista-ban

A Windows Vistának nincs boot.ini állománya, ellenben a korábbi Windows operációs rendszerekkel, ahol használni lehetne a /noexecute kapcsolót, mint például az XP-ben. A betöltés alatt beállítandó opciók a BCD (Boot Configuration Data) nevű struktúrába kerülnek, ahol az adminisztrátorok más lehetőségek mellett a DEP-et, azaz a Vista esetében az NX-házirendet is állíthatják a bcdedit.exe paranccsal. A paraméterek nélkül futtatott bcdedit parancs megmutatja az alapvető beállításokat. A „Boot Loader" szakaszban alapesetben azt látjuk, hogy az NX OptIn-re van állítva. Ez a 32 bites Vistánál azt jelenti, hogy alapesetben a korábbi platformokhoz hasonlóan csak a futtatható rendszerfájlokra érvényes a védelem. A 64 bites változatra igaz, hogy minden 64 bites alkalmazás NX-védett, a 64 bites rendszeren futó 32 bites programok védelme pedig megfelel a 32 bites platformének. Az NX-házirend beállításánál a következő paraméterek használhatók: OptIn, OptOut, AlwaysOn, AlwaysOff. OptOut esetben a DEP minden futó folyamatra érvényes lesz, de a felhasználók a Vezérlőpult Rendszer beállításának segítségével létrehozhatnak kivétellistákat, és az ezekben felsorolt programokra az operációs rendszer nem alkalmazza a memóriavédelmet. Az AlwaysOn esetén a DEP ennél az opciónál is minden folyamatra kiterjed, de a rendszer itt nem enged meg semmilyen felhasználói/fejlesztői kivételezést, azaz semmilyen listával vagy programozói fogással sem lehet kivonni semmilyen folyamatot a DEP védelme alól. AlwaysOff esetén az operációs rendszer semmilyen programot sem lát el DEP-védelemmel. A DEP ugyanúgy nem érvényesül a futtatható rendszerállományokon, ahogyan a felhasználói programokon sem.

4.1.4 NX-es programok írása

NX-es programok írása esetén legtöbbször nem kell különösebben figyelni a kód NX-es vonatkozásaira, a 32 bites alkalmazások az érvényes beállítások szerint viselkednek majd, a 64 bitesek meg úgyis NX-védettek, amiről a fordítóprogram és a Vista gondoskodik. Az alacsony szintű nyelven írt alkalmazásoknál, vagy magas szintű nyelvbe ágyazott alacsony

szintű kód esetén az NX-megfelelőségre is figyelni kell. A Microsoft Linker /NXCOMPAT kapcsolójának használatával kijelentjük az alkalmazásról, hogy NX-szempontból megfelelő. A Vista alá fejlesztett alkalmazásoknál általában célszerű használni ezt a kapcsolót, így a Vista is NX-védelemben futtatja majd a kódot. Ha speciálisan a Vista alá fejlesztett programoknál /SUBSYSTEM:6.0 linkelési opciót használunk, akkor az /NXCOMPAT-ot nem kell külön megadni, mert automatikus a beállítása. Ha Vista alatt valamilyen okból ki akarunk vonni egy alkalmazást az NX-védelem alól, akkor ezt a /NXCOMPAT:NO opcióval érhetjük el. Ha alkalmazásunk dinamikusan lefoglalt memóriaterületen akar kódot futtatni, akkor a VirtualAlloc() szabványos API-hívással kell lefoglalnia a tárterületet oly módon, hogy a Vista számára futtathatónak jelöli meg azt a memóriát, amire a foglalási igény vonatkozik. A Vistában semmilyen más memóriafoglalási lehetőség, sem függvény sem rendszerhívás nem alkalmas futtatható tárterület lefoglalására!

4.2 ASLR

Az Address Space Layout Randomization (ASLR) véletlenszerű címterület-kiosztás jelent, ami abban nyilvánul meg, hogy a betöltött alkalmazás, program vagy folyamat kulcsterületei (futó kódja, könyvtárai, heap, verem stb.) véletlenszerű helyre kerülnek a memóriában. Minden egyes modulhoz külön generálódik egy random érték. Ez a technika megnehezíti egy lehetséges külső támadó dolgát, mert egyrészt nem jósolható meg előre a konkrét memóriacím, másrészt a kiszámítása (helyesebben próbálgatása) is vesződséges.

A nyílt kódú ASLR-technológiát először Linux-rendszereken használták (OpenBSD, linuxos PaX és Exec Shield rendszerek), a Vista-tesztváltozatok közül a második bétánál került be először. Az ASLR jósága attól függ, hány bitet, azaz mekkora entrópiafaktort használunk a memória randomizálásához, konkrétan egy folyamat kezdőcímének a kitalálási esélye

$$\frac{1}{2^n}$$

, ahol n a bitek száma.

Ha a támadónak esetleg több modulra is szüksége van, akkor tovább romlik a találati arány:

$$\frac{1}{m * 2^n}$$

, ahol m a célmodulok száma.

Bizonyos ASLR-implementációknál a betöltés alatt még az egyes modulok betöltési sorrendjét is külön randomizálják (Library Load Order Randomization). Ebben az esetben a támadónak még a sorrendet is ki kell találnia, amire

$$\frac{1}{k}$$

az esélye, ahol k a betöltendő modulok száma.

A puffer-túlcsordulásokat kihasználó rosszindulatú programoktól az ASLR sem véd meg, de megnehezíti a támadó dolgát. Önmagában még nem erős védelem, de más technológiákkal kombinálva hasznos lehet, mert a támadó szempontjából a Vista nem az, aminek látszik. A Vista-ban nemcsak jelen van az ASLR, hanem alapértelmezésben be is van kapcsolva. Az ASLR kiterjed a visszatérési veremre heap-memóriára és az operációs rendszer részeként települő összes bináris állományra. Minden más .exe vagy .dll állománynak kifejezetten kérnie kell az ASLR-lehetőséget a megfelelő PE-fejlécmutató (Portable Executable Header) beállításával.

Azt is meg kell jegyezni, hogy a Vista-ban az n értéke túlságosan kicsi, így gyorsan ki lehet találni a belépési pontokat, mindössze 256 lehetőség. Viszont a technológiák összessége erős védelmet jelent.

Összefoglalás

A programvédelem megalkotása egy meglehetősen nehéz feladat, de nem lehetetlen. A bemutatott módszereket kombinálva erős védelem építhető, de kerülni kell a példákban használt egyszerűséget, mivel minél egyszerűbb, vagy ha sablonos rendszert állítunk össze, a támadó annál könnyebben távolítja el. A cracker-ek jól bevált módszereket és programokat használnak egy adott típusú védelem feltörésére, ami 80-90%-os sikerrel működik is. Ezért ajánlott jól elrejteni a védelmet, és több módszert együttesen használva megalkotni a saját egyedi védelmünket.

A következők betartása növeli a feltörés esélyét. A fontosabb részeknél kerülni kell a beszédes nevek használatát (főleg a DLL-ekben), kerüljük a hibaüzeneteket a törés észlelésénél, mert az csak segítség a nyomkövetésnél, végül használjunk hathatós kódolást a regisztrációs file-oknál, és ahol azt megköveteli a program (pl.: RSA 1024-et).

Ha azt tapasztaljuk, hogy feltörték a programunkat a következő verzióban cseréljük a védelmet. Ezt egyébként minden verzióváltásnál célszerű megtenni, függetlenül a feltörés észlelésétől.

Amennyire lehetséges mindig maradjunk naprakészek, a Windows is legyen a lehető legfrissebb, mivel sok nyomkövető program nehezen működik a modern biztonsági rendszerek mellett. Az újabb nyomkövetők, mint például a Syser debugger, még annyira új, hogy nem találtam speciálisan ellene kifejlesztett védelmi módszert, bár a cracker-ek se rendelkeznek még kellő tapasztalattal a használatához, a jövőben fel kell készíteni a programunkat ezek elleni védelemre is.

Mindezek mellett kellő időt kell fordítani a programvédelem tesztelésére, minden lehetséges futtatási környezetben. Mivel sok esetben, az operációs rendszerek fejlődése és fejlett memória-, és regisztervédelme miatt, modernebb rendszerek alatt nem, vagy csak hibásan működnek.

Remélem dolgozatommal sikerült átfogó képet alkotnom, és segítségére lennem azoknak, akik programjuk védelmén gondolkoznak.

Irodalomjegyzék

Könyvek:

Pavol Červeň: Programvédelem fejlesztőknek (2003) Budapest, Kiskapu

Internetes adatgyűjtés:

http://66.102.9.104/search?q=cache:XxS-9H27WZgJ:www.woodmann.com/RCE-CD-SITES/Library/Manuels%2520%26%2520Misc/Anti_debug/SoftIce%25203.xx%2520detection.rtf+c%2B%2B+CreateFileA+detect+softice&hl=hu&ct=clnk&cd=1&gl=hu&client=firefox-a

http://www.msportal.hu/k12/vistakepzes/Resources/DEPEsASLR/Default.htm

http://techies.teamlupus.hu/2007/06/23/pinvoke-code-access-security/

http://www.codeproject.com/KB/cs/unmanage.aspx

http://www.codexonline.hu/CodeX8/alap/assembly/GebeiJanos/asm_direkt_reg.htm

http://w3.to/protools

http://msdn2.microsoft.com/en-us/library/45yd4tzz(VS.80).aspx

http://members.fortunecity.com/blackfenix/sicedete.html

http://members.fortunecity.com/blackfenix/isbpxede.html

http://mirrors.sinuspl.net/www.crackstore.com/003.htm

http://engstromfamily.com/bookshelf/books/SecProgCnC++/0596003943_secureprgckbk-chp-12-sect-15.html

http://pcforum.hu/hirek/9858/Kijatszhato+a+processzorok+hardveres+toresvedelme.html

http://www.woodmann.com/crackz/Tutorials/Dotnet.htm

http://mirror.sweon.net/madchat/windoz/win32inc/

http://programujte.com/index.php?akce=clanek&cl=2006082907-cracking-%E2%80%93-3-cast

http://www.codeproject.com/KB/mcpp/quickcppcli.aspx

http://en.wikipedia.org/wiki/.NET_Reflector

http://en.wikipedia.org/wiki/SoftICE

http://en.wikipedia.org/wiki/Syser

http://www.compuware.com/products/devpartner/studio.htm

http://www.sysersoft.com/products.htm

http://en.wikipedia.org/wiki/Symmetric_multiprocessing

http://www.eziriz.com/

http://www.securom.com/

http://en.wikipedia.org/wiki/SafeDisc

http://www.macrovision.com/products/activereach_games/safedisc/index.shtml

http://en.wikipedia.org/wiki/CD-Cops

http://www.ollydbg.de/

http://www.aladdin.com/hasp/default.aspx

http://www.safenet-inc.com/products/sentinel/hardware_keys.asp

http://en.wikipedia.org/wiki/Data_Execution_Prevention

http://support.microsoft.com/KB/875352

http://www.updatexp.com/data-execution-prevention.html

http://en.wikipedia.org/wiki/Address_space_layout_randomization

http://technet.microsoft.com/en-us/magazine/cc162458.aspx

http://hardware.earthweb.com/chips/article.php/3358421

http://en.wikipedia.org/wiki/NX_bit

Függelék

1-es számú melléklet a CRC számításhoz használatos táblázat (C# tömb):

```
    private readonly static uint[] CrcTable = new uint[] {
0x00000000, 0x77073096, 0xEE0E612C, 0x990951BA, 0x076DC419,
0x706AF48F, 0xE963A535, 0x9E6495A3, 0x0EDB8832, 0x79DCB8A4,
0xE0D5E91E, 0x97D2D988, 0x09B64C2B, 0x7EB17CBD, 0xE7B82D07,
0x90BF1D91, 0x1DB71064, 0x6AB020F2, 0xF3B97148, 0x84BE41DE,
0x1ADAD47D, 0x6DDDE4EB, 0xF4D4B551, 0x83D385C7, 0x136C9856,
0x646BA8C0, 0xFD62F97A, 0x8A65C9EC, 0x14015C4F, 0x63066CD9,
0xFA0F3D63, 0x8D080DF5, 0x3B6E20C8, 0x4C69105E, 0xD56041E4,
0xA2677172, 0x3C03E4D1, 0x4B04D447, 0xD20D85FD, 0xA50AB56B,
0x35B5A8FA, 0x42B2986C, 0xDBBBC9D6, 0xACBCF940, 0x32D86CE3,
0x45DF5C75, 0xDCD60DCF, 0xABD13D59, 0x26D930AC, 0x51DE003A,
0xC8D75180, 0xBFD06116, 0x21B4F4B5, 0x56B3C423, 0xCFBA9599,
0xB8BDA50F, 0x2802B89E, 0x5F058808, 0xC60CD9B2, 0xB10BE924,
0x2F6F7C87, 0x58684C11, 0xC1611DAB, 0xB6662D3D, 0x76DC4190,
0x01DB7106, 0x98D220BC, 0xEFD5102A, 0x71B18589, 0x06B6B51F,
0x9FBFE4A5, 0xE8B8D433, 0x7807C9A2, 0x0F00F934, 0x9609A88E,
0xE10E9818, 0x7F6A0DBB, 0x086D3D2D, 0x91646C97, 0xE6635C01,
0x6B6B51F4, 0x1C6C6162, 0x856530D8, 0xF262004E, 0x6C0695ED,
0x1B01A57B, 0x8208F4C1, 0xF50FC457, 0x65B0D9C6, 0x12B7E950,
0x8BBEB8EA, 0xFCB9887C, 0x62DD1DDF, 0x15DA2D49, 0x8CD37CF3,
0xFBD44C65, 0x4DB26158, 0x3AB551CE, 0xA3BC0074, 0xD4BB30E2,
0x4ADFA541, 0x3DD895D7, 0xA4D1C46D, 0xD3D6F4FB, 0x4369E96A,
0x346ED9FC, 0xAD678846, 0xDA60B8D0, 0x44042D73, 0x33031DE5,
0xAA0A4C5F, 0xDD0D7CC9, 0x5005713C, 0x270241AA, 0xBE0B1010,
0xC90C2086, 0x5768B525, 0x206F85B3, 0xB966D409, 0xCE61E49F,
0x5EDEF90E, 0x29D9C998, 0xB0D09822, 0xC7D7A8B4, 0x59B33D17,
0x2EB40D81, 0xB7BD5C3B, 0xC0BA6CAD, 0xEDB88320, 0x9ABFB3B6,
0x03B6E20C, 0x74B1D29A, 0xEAD54739, 0x9DD277AF, 0x04DB2615,
0x73DC1683, 0xE3630B12, 0x94643B84, 0x0D6D6A3E, 0x7A6A5AA8,
0xE40ECF0B, 0x9309FF9D, 0x0A00AE27, 0x7D079EB1, 0xF00F9344,
0x8708A3D2, 0x1E01F268, 0x6906C2FE, 0xF762575D, 0x806567CB,
0x196C3671, 0x6E6B06E7, 0xFED41B76, 0x89D32BE0, 0x10DA7A5A,
0x67DD4ACC, 0xF9B9DF6F, 0x8EBEEFF9, 0x17B7BE43, 0x60B08ED5,
0xD6D6A3E8, 0xA1D1937E, 0x38D8C2C4, 0x4FDFF252, 0xD1BB67F1,
0xA6BC5767, 0x3FB506DD, 0x48B2364B, 0xD80D2BDA, 0xAF0A1B4C,
0x36034AF6, 0x41047A60, 0xDF60EFC3, 0xA867DF55, 0x316E8EEF,
0x4669BE79, 0xCB61B38C, 0xBC66831A, 0x256FD2A0, 0x5268E236,
0xCC0C7795, 0xBB0B4703, 0x220216B9, 0x5505262F, 0xC5BA3BBE,
0xB2BD0B28, 0x2BB45A92, 0x5CB36A04, 0xC2D7FFA7, 0xB5D0CF31,
0x2CD99E8B, 0x5BDEAE1D, 0x9B64C2B0, 0xEC63F226, 0x756AA39C,
0x026D930A, 0x9C0906A9, 0xEB0E363F, 0x72076785, 0x05005713,
0x95BF4A82, 0xE2B87A14, 0x7BB12BAE, 0x0CB61B38, 0x92D28E9B,
0xE5D5BE0D, 0x7CDCEFB7, 0x0BDBDF21, 0x86D3D2D4, 0xF1D4E242,
0x68DDB3F8, 0x1FDA836E, 0x81BE16CD, 0xF6B9265B, 0x6FB077E1,
0x18B74777, 0x88085AE6, 0xFF0F6A70, 0x66063BCA, 0x11010B5C,
```

```
 0x8F659EFF, 0xF862AE69, 0x616BFFD3, 0x166CCF45, 0xA00AE278,
 0xD70DD2EE, 0x4E048354, 0x3903B3C2, 0xA7672661, 0xD06016F7,
 0x4969474D, 0x3E6E77DB, 0xAED16A4A, 0xD9D65ADC, 0x40DF0B66,
 0x37D83BF0, 0xA9BCAE53, 0xDEBB9EC5, 0x47B2CF7F, 0x30B5FFE9,
 0xBDBDF21C, 0xCABAC28A, 0x53B39330, 0x24B4A3A6, 0xBAD03605,
 0xCDD70693, 0x54DE5729, 0x23D967BF, 0xB3667A2E, 0xC4614AB8,
 0x5D681B02, 0x2A6F2B94, 0xB40BBE37, 0xC30C8EA1, 0x5A05DF1B,
 0x2D02EF8D
};
```

Köszönetnyilvánítás:

Ezúton szeretnék köszönetet mondani Juhász István tanár úrnak, hogy szakmai tapasztalataival segítette munkámat.