SZAKDOLGOZAT

Keresztes Zoltán Róbert

Debrecen

2010

Debreceni Egyetem
Informatika Kar

KÉMPROGRAM ÍRÁSA C# NYELVEN

Témavezető:
Dr. Végh János
DSc, egyetemi tanár

Készítette:
Keresztes Zoltán Róbert
Mérnök Informatikus

Debrecen
2010

TARTALOMJEGYZÉK

Tartalomjegyzék ... 1
1. Köszönetnyilvánítás ... 1
2. Bevezetés ... 2
3. A programmal szemben támasztott követelmények ... 2
3.1 Funkcionális követelmények: ... 2
3.2 Nem funkcionális követelmények: ... 2
3.3 A fejlesztés hardver és szoftver környezete: ... 3
4. Az alkalmazott technológiák ismertetése ... 3
4.1 A COM technológia ... 3
4.2 Rejtőzködési technikák bemutatása ... 5
4.3 Architektúra megtervezése és tesztelése ... 12
4.4 Program védelemi módszerek ismertetése ... 23
4.5 Programfeltörési technikák megismerése ... 26
5. A megvalósításra került program bemutatása ... 35
5.1 A szoftver felhasználói felülete ... 35
5.2 Az alkalmazás architekturája ... 38
5.3 Jogosultságok kezelése ... 40
5.4 A program védelmi mechanizmusa ... 41
5.5 Az alkalmazás által használt rejtőzködő funkció ismertetése ... 42
5.6 Az alkalmazás digitális aláírása ... 46
6. Konkurencia értékelése ... 51
7. Összefoglalás ... 53
8. Irodalomjegyzék ... 54

1. KÖSZÖNETNYILVÁNÍTÁS

Ezúton szeretném megköszönni mindenkinek, aki szakdolgozatom megírásához valamilyen módon hozzájárult. Köszönetet mondok Dr. Végh Jánosnak, a témavezetőmnek, aki véleményeivel, javaslataival ösztönzött arra, hogy minél jobb munka kerüljön ki a kezeim közül. Hálával tartozom családtagjaimnak, akik szakdolgozatom megírása során kellő megértést és türelmet tanúsítottak irántam, mellyel elősegítették gondolataim összegzését, szakdolgozatom megírásának sikerességét.

2. BEVEZETÉS

Az embert már ősidők óta foglalkoztatja, hogy fényt derítsen mások titkaira, ez mind igaz nemzeti szinten (különböző kémműholdak, felderítő repülőgépek) és a magánéletben is (magánnyomozó), természetesen a kémkedés a számítástechnikában is jelen van. Számítógépünkre védelemi szoftverek nélkül különböző kémprogramok települhetnek, melyek információkat gyűjthetnek rólunk, amelyet később rossz célokra is felhasználhatnak. Ezért az emberek különböző védelmi szoftvereket használnak rendszerük védelmére, de ezek se védenek minden ellen. A védelmi szoftverek gyártói és a kémprogramok írói között egy örökös harc folyik, hol egyik (biztonsági rés), hol a másik kerül ki ideiglenes győztesen (foltozás). Természetesen egy kémprogram használatának jó szándékú oka is lehet pl.: bűnözők lenyomozása vagy, ha a magánéletbeli dolgokat nézzük, akkor a kémprogram használatának oka lehet a féltékenység, ezért a feleség/férj ellenőrzése, alkalmazott ellenőrzése, hogy megtudjuk, hogy mit csinál munkaidőben vagy akár a gyermek felügyelet alatt tartása, hogy nehogy, olyan dolgokra vegyék rá, amit magától nem tenne. Az általam készített szoftver ezekre a problémára nyújt megfelelő megoldást, teljesen láthatatlan módon.

3. A PROGRAMMAL SZEMBEN TÁMASZTOTT KÖVETELMÉNYEK

3.1 FUNKCIONÁLIS KÖVETELMÉNYEK:

Tegye lehetővé a Windows Live Messengeres beszélgetések naplózását, akár akkor is, ha a felhasználó kikapcsolta a fiókjában a naplózást. A program rejtőzködjön, a megfigyelt személy ne szerezhessen tudomást a program létéről. A naplózott beszélgetéseket tárolása lehetséges legyen a számítógépen vagy akár egy előre magadott email címre történő továbbítása is.

3.2 NEM FUNKCIONÁLIS KÖVETELMÉNYEK:

A kezelőfelület legyen felhasználóbarát, ne igényeljen különösebb informatikai előismereteket a program használata.

3.3 *A FEJLESZTÉS HARDVER ÉS SZOFTVER KÖRNYEZETE:*

Hardver:

- Processzor: 2 GHz Mobil Sempron 3500+
- Memória: 1 GB
- Winchester: 120 GB

Szoftver:

- Operációs rendszer: Microsoft Windows XP, Windows 7
- Fejlesztő eszköz: Visual Studio 2008 Professional
- Programozási nyelv: C#, C
- Tesztelő eszközök: NCover, VS Unit Test, Rhino Mock Framework

4. AZ ALKALMAZOTT TECHNOLÓGIÁK ISMERTETÉSE

4.1 *A COM TECHNOLÓGIA*

A COM (Component Object Model) elnevezés azt a nyelv független szoftvermodellt jelöli, amely lehetővé teszi, hogy a Windows programok használják az alkalmazástól független, lefordított programelemeket (komponenseket). A szoftvermodell kialakításának alapvető jellegzetessége az a bináris szabvány, amely lehetővé teszi, hogy bármely nyelv, bármely fejlesztőeszköze alkalmas legyen a programelemek létrehozására. A COM-alapú elemeket a nélkül használhatjuk, hogy tudnánk, hol helyezkednek el fizikailag a háttértáron. A hálózati elérési lehetőségeket is tartalmazó COM technológiát DCOM-nak (Distributed COM) nevezzük. A COM komponensek nyelv független megvalósítására jellemző, hogy akár script nyelvekből is elérhetjük.

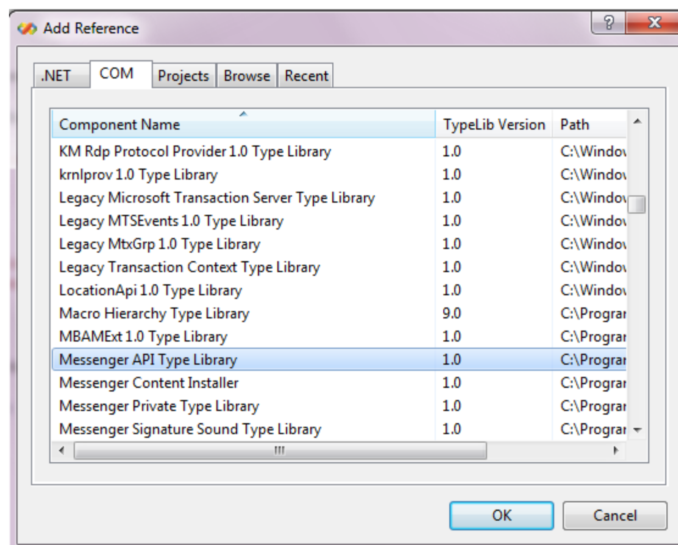
A COM komponensek olyan önálló egységek, melyek bináris formában jelennek meg, rendszerint egy DLL vagy egy EXE fájl formájában.

A COM-komponensek elérését a Windows operációs rendszerek a regisztrációs adatbázis segítségével támogatják. Minden egyes felhasználható osztálynak és kapcsolati felületnek egyedi azonosítóval ellátott bejegyzése van az adatbázisban. Ezt a 128-bites azonosítót GUID-nak (Global Unique Identifier - globális egyedi azonosítónak) hívjuk. Ilyen azonosítókat gyárthatunk a CoCreateGuid() API-hívással, vagy akár a GUID generátor

program alkalmazásával is, mely megtalálható a Microsoft Windows SDK-ban. A generálás során annyira sok összetevő játszik szerepet, hogy szinte lehetetlen két objektumnak azonos GUID azonosítót létrehozni. A COM technológia lehetővé teszi, hogy két különálló alkalmazásmodul egy speciálisan kialakított kapcsolódási felületen keresztül kommunikáljon egymással. A két alkalmazás a kommunikáció szempontjából egyenértékű, azonban azt az alkalmazást, amelyik megteremti a kommunikáció alapfeltételeit kiszolgálónak (server) nevezzük, a másikat pedig ügyfélnek (client).

Ezeken keresztül az ügyfélalkalmazásokból elérhetjük a kiszolgáló lehetőségeit. A COM definiálja a kapcsolódási felület kialakításának és elérésnek szabályait és módjait.

Amikor COM komponenseket és .NET objektumokat használunk egy alkalmazáson belül, a legfontosabb különbség a két objektum-típus között a memóriahasználatban van. A .NET objektumok a menedzselt memóriában találhatóak, melyet a Common Language Runtime (CLR) tart felügyelete alatt. Szükség esetén el is távolítja az objektumokat a memóriából. A COM objektumok számára az úgynevezett nem-menedzselt memóriában foglalódik le a terület, és várhatóan nem kerülnek át másik memóriaterületre életük során. Amikor .NET alkalmazás egy COM objektumot hív meg, akkor egy úgynevezett burkoló osztály jön létre. Ahhoz, hogy a két memóriaterület között szükséges kapcsolat megvalósuljon, rengeteg Windows-os alkalmazás rendelkezik COM –os kapcsolófelülettel. Ezen kapcsolófelületeken keresztül hozzáférhetünk más alkalmazások funkcióihoz, értesítéseket kaphatunk más alkalmazásban bekövetkezett eseményekről. Ezen COM felületek használatával például a saját programunkból létrehozhatunk Word, Excel, Access dokumentumot, hozzáférhetünk a Windows Live Messengerhez, vagy akár a Skype, Adobe Reader-hez is, hogy ne csak a Microsoft által készített program kerüljön említésre.



1. *ábra: COM referencia hozzáadása a projekthez*

A Visual Studio-ban egyszerűen csak hozzá kell adni a kiválasztott COM objektumot a referenciákhoz és ezek után ugyanúgy dolgozhatunk vele, mint bármelyik másik referenciával.

4.2 *REJTŐZKÖDÉSI TECHNIKÁK BEMUTATÁSA*

A "rootkit" fogalom egy olyan technika vagy módszer leírására szolgál, amikor is egy rosszindulatú program (pl. vírus, kémszoftver, trójai) megpróbálja magát eltüntetni a védelmet biztosító szoftverek elől, tehát például a vírusölő, a kémszoftvereket blokkoló vagy egyéb rendszerkezelő programok elől. Az ezután felsorolásra kerülő technikák alkalmazásához mindenképpen szükséges valamilyen natív kódra forduló programozási nyelv használata. „ lehetőség közül választhatunk vagy az egész rootkit natív nyelven kerül megírásra vagy csak egy bootstrappert írunk meg natív nyelven a többi funkciót már managelt nyelven implementáljuk.

A rootkiteket működésük szerint az alábbiak szerint csoportosítják:

4.2.1 *ÁLLANDÓ ROOTKIT (PERSISTENT ROOTKIT):*

Az állandó rootkit fogalma arra utal, hogy a rosszindulatú program minden rendszerindítás után újraindul. Tehát ennek a kódsorozatnak állandóan a rendszerben kell tárolódnia, például a registry-ben vagy a fájlrendszerben. Valamint léteznie kell egy olyan eljárásnak is, amely a

felhasználó beavatkozása nélkül gondoskodik ennek a kódsorozatnak a boot utáni elindításáról.

4.2.2 Memóriában lévő rootkit (Memory-based rootkit):

A memóriában található rootkiteknél a kódsorozat nem kerül a háttértárba, ezért ez a fajta rootkit nem éli túl a rendszerindítást.

4.2.3 Kernel módú rootkit (Kernel-mode rootkit):

A kernel módú rootkit nagyon hatékony, mivel kernel módban nemcsak az eredeti API-t tudja feltartóztatni, hanem közvetlenül manipulálhat a kernel módú adatstruktúrákkal. A legtöbb operációs rendszer támogatja a kernel módú drivereket, melyek ugyanolyan jogosultságokkal futnak, mint az operációs rendszer, ezen okból kifolyólag számos kernel módú rootkit született, melynek az alapja egy betöltő vagy egy driver. A kernel módú rootkiteket különösen nehéz felismerni és eltávolítani. Mivel ugyanolyan jogosultságokkal rendelkezik, mint az operációs rendszer, így módosíthatja a kimenő adatokat.

4.2.4 Felhasználó módú rootkit (User-mode rootkit)

Sokfajta eljárás létezik arra, hogy hogyan próbálja meg magát eltüntetni egy rosszindulatú kód. Például egy felhasználói rootkit a Windowsban minden FindFirstFile/FindNextFile API hívásnál közbelép (ezeket a hívásokat a fájlrendszert lekérdező programok használják arra, hogy egy könyvtár tartalmát számba vegyék, ezt használja a Windows Explorer és ez működik parancsmódú fájllekérdezésnél is.) Ha egy alkalmazás egy könyvtár tartalmára kíváncsi, normális esetben visszakapná a könyvtár tartalmát, de a rootkit ilyenkor közbelép, és módosítja a választ, hogy a rootkitre utaló bejegyzések ne jelenjenek meg.

A Windows eredeti API-ja arra szolgál, hogy a felhasználói kliensek és a kernel módú szolgáltatások (services) között illesztőfelületet, azaz interfészt biztosítson. A felhasználó módú rootkit pedig az API működésébe lép közbe akkor, amikor az a fájlrendszer, a registry vagy a processzekkel kapcsolatos teendőit végzi. Így védi magát a rootkit attól, hogy észleljék. Az API hooking első lépése a DLL injektálás.

DLL injektálás révén betölthető a rootkit kódja egy másik folyamat memóriaterületére.

A DLL injektálásra különböző módszerek használatosak:

1) **Rendszerleíró adatbázis (Registry):**

Ez a technika azon alkalmazásoknál működik, melyek működésük során használják a user32.dll–t. A Windows rendszerszintű szolgáltatásait a programok nem közvetlenül hívják meg, hanem alrendszereken keresztül, melyeket DLL ként implementáltak a rendszerben. A programok által leggyakrabban használt DLL-ek közé tartozik a user32.dll is. Ezt a DLL –t a legtöbb grafikus felülettel rendelkező alkalmazás használja. A user32.dll inicializáló része megnézi a rendszerleíróadatbázis **HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Windows\AppInit_DLL** kulcsában található értékét és inicializáló során ezen DLL–t/DLL-eket is betölti a felhasználói módú folyamat memória területére. Az érték állhat egyetlen DLL nevéből, vagy akár DLL -ek csoportjából is, amely vesszővel vagy szóközzel vannak elválasztva. A fentebb említett regisztrációs kulcsot módosítva tetszőleges DLL tölthető be az alkalmazással együtt.

Technika előnyei:

- A számítógép újraindítása után is a rendszerben marad a rootkit.

Technika hátrányai:

- A konzolos alkalmazásokon ez a technika nem működik, mivel nincs olyan funkciójuk, melyet a user32.dll-ból importálnának.
- Nincs beleszólásunk abba, hogy mely alkalmazásba injektálódjon a DLL -ünk. Minden GUI–val (mely használja a user32.dll-t) rendelkező alkalmazás memóriaterületére injektálódni fog a DLL -ünk.

A Windows Vista -tól kezdve az AppInit_DLL alapértelmezetten le van tiltva.

Érték	Leírás	Minta érték
LoadAppInit_DLL (REG_DWORD)	Az érték globálisan engedélyezi/letiltja az AppInit_DLL –t	0x0 – Az AppInit_DLLs le van tilva. 0x1 –Az AppInit_DLLs engedélyezve van.
AppInit_DLL (REG_SZ)	Szóközzel vagy vesszővel vannak elválasztva a DLL-ek útvonala melyek betöltésre kerülnek. Rövidített útvonal is megadható.	C:\PROGRA~1\Test\Test.dll
RequireSignedAppInit_DLL (REG_DWORD)	Digitálisan aláírt DLL –t igényel.	0x0 – Minden DLL betöltésre kerül. 0x1 – Csak digitálisan aláírt DLL ek kerülnek betöltésre.

A DLL betöltődéséhez kapcsolódó registry értékek

4.2.5 DRM VÉDETT FOLYAMATOK

A Windows Vista által bevezetésre került egy új típusú folyamat, amely védett folyamat néven ismert, mely támogatja a digitális aláírást. Az AppInit DLL -ek nem lesznek betöltve a DRM védett folyamatokba. Ezt a működést nem lehet beállítani. Az elsődleges különbség a normál folyamatok és a védett folyamatok között, a hozzáférési szint, amely szabályozza, hogy más folyamatok a rendszerben, hogyan férhetnek hozzá a védett folyamatokhoz. A Windows Vista előtt, ha a felhasználó elindított egy folyamatot, akkor ez a folyamat hozzáférhetett a rendszer összes folyamatához. Ez továbbra is igaz a normál folyamatokra, de lényegesen más a hozzáférés módja a védett folyamatokhoz.

A normál folyamatok nem hajthatják végre a következő műveleteket a védett folyamatokon:

- Szál beszúrás a védett folyamatba
- Hozzáférés a védett folyamat virtuális memóriájához
- A munkaterület kvótájának megváltoztatása
- Debuggolás

4.2.6 IAT

Amikor egy fájl betöltésre kerül, akkor a betöltő automatikusan átnézi a fájl azon részét, ahol az IAT (Importing Address Table) található, amelyben megtalálható a DLL-ek listája, valamint azok a függvények, melyeket a program ezen DLL -ekből használni fog. Bináris fájlok IAT-jaiban található címek módosításával egy program magára irányíthatja a végrehajtást, s befolyásolhatja az eredeti függvény futását: egy program egy könyvtár listázását végzi, és néhány műveletet hajt végre rajtuk. Tételezzük fel, hogy felhasználói módban fut, és Win32-es alkalmazás. Ekkor a Kernel32, a User32 és a Gui32 DLL-eket szinte biztosan használni fogja használni az adott alkalmazás. A könyvtár listázásához a FindFirstFile és amennyiben az sikeres, a FindNextFile API függvényeket fogja meghívni. Amikor az alkalmazás meghívja a függvényt, akkor az import táblából kikeresi a címet, s a megfelelő függvényre ugrik a Kernel32.dll-ben. A rootkit az IAT címet átírva a saját függvényére irányíthatja a hívást, és tulajdonképpen bármilyen utasítást végrehajthat: meghívhatja az eredeti függvényt, s annak visszatérési adatait szűrheti stb. Fontos tudnunk, hogy amikor átírja ezt a címet, a rootkit átlépi a folyamat virtuális címtartományát, s ezzel detektálhatóvá válik.

4.2.7 RENDSZER SZINTŰ WINDOWS HOOK

Ez a leggyakrabban használt technika a user szintű rootkiteknél, Ez a technika a rendszerszintű üzeneteket kapja el. A rootkit egy szűrőt telepít a rendszerbe mellyel, figyelemmel kísérheti az alkalmazásoknak küldött üzeneteket, mielőtt azok célba érnének. Ehhez a SetWindowsHookEx API függvény használatára van szükség, mellyel más folyamatok eseményeihez férhetünk hozzá, s tetszőleges DLL –t tölthetünk be ezen folyamatok memória területére, legalábbis úgy látszik, a Windows memóriakezelőjének egyik alapvető szolgáltatása a megosztott memória támogatása. A háttérben minden egyes natív

DLL használatakor ez teszi lehetővé, hogy a sok folyamat által használt DLL-ek kódja és bizonyos részben adatai is csak egyszer legyenek a fizikai RAM-ban, mégis minden folyamat úgy láthatja, mintha a saját címterében lenne. Így memóriát és lemezterületet takarítva meg, valamint a programok készítése is egyszerűsödik, mivel ugyanazt a rutineljárást csak egyszer kell elkészíteni. Azonban, ha egy új program úgy telepít egy DLL-t, hogy felülírja annak régebbi változatát, ez eredményezheti, hogy régebb telepített programok (amelyek a régi DLL-t használták) többet nem fognak futni. Ezt hívjuk DLL Hell-nek (DLL pokolnak). Fontos még megjegyezni, hogy egy 32 bites DLL csak 32 bites folyamatok, egy 64 bites DLL csak 64 bites folyamatokhoz férhet hozzá. Érdekességképpen még megemlíthető, hogyha egy .NET-ben íródott DLL –t töltünk be a programunkba, akkor az minden esetben újra betöltődik a memóriába azaz, hogyha 10 program használja egyszerre ugyanazt a DLL –t, akkor az 10 szer fog betöltődni a memóriába és újrafordulni futás közben. Ezért a .NET fejlesztői kitalálták, hogy a lehetőség legyen arra, hogy az alkalmazások, DLL –ek natív képe a számítógépen tárolódjon. Ehhez az ngen program használatára van szükség. Az ngen segítségével az alkalmazásunk, vagy a DLL -ünknek készít egy native image-t, amely a native image cache-ben a GAC-on belül egy elkülönített részen lesz tárolva. A program futtatásakor vagy a DLL betöltődésekor a JIT fordító megnézi, hogy szerepel-e a cache-ben, ha igen, akkor nem fogja futási időben a szükséges részeket lefordítani, hanem a native image-t tölti be a memóriában. Ekkor ez úgy működik, mint egy natív DLL esetében, azaz a Windows a DLL –t megosztja az alkalmazások között. A ngen segítségével a program betöltődési ideje is csökkenthető (mivel nem kell a szükséges részeket lefordítania a JIT- nek).

SetWindowHookEx függvény paraméterei.

HHOOK SetWindowsHookEx(

int	***idHook***,
HOOKPROC	***lpfn***,
HINSTANCE	***hMod***,
DWORD	***dwThreadId***
);	

);

Az idHook paraméter specifikálja a telepíteni kívánt hook típusát, ezek közül most néhány fontosabb felsorolásra kerül:

- WH_CALLPROC: Egy olyan hook –ot telepít, mely figyeli az üzeneteket, mielőtt azt a rendszer elküldené a címzett ablak eljárásnak.
- WH_CALLWNDPROCRET: Egy olyan hook –ot telepít, mely figyeli az üzeneteket, miután azt feldolgozta a címzett ablak eljárás.
- WH_KEYBOARD: Egy olyan hook –ot telepít, mely figyeli a billentyű leütések üzeneteit
- WH_MOUSE: Egy olyan hook- ot telepít, mely figyeli az egér üzeneteket.

Az lpfn paraméter HOOKPROC típusú, és egy mutatót definiál a visszahívó funkcióra, melynek végrehajtására akkor kerül sor, amikor bekövetkezik az első paraméterként megadott típusú esemény.

A hMod paraméter HINSTANCE típusú, és annak a hozzákapcsolt eljárást tartalmazó DLL-nek a példányleírója, amelyre a hHookProc mutat. Ha a dwThreadID egy, az aktuális alkalmazás által létrehozott szálat jelöl ki, és a hozzákapcsolt feldolgozás az aktuális alkalmazáshoz tartozó kódban található, a hMod paraméter értéke kötelezően NULL.

A dwThreadID a hozzákapcsolt eljáráshoz tartozó szál DWORD típusú azonosítója. Ha értéke nulla, a hozzákapcsolt feldolgozás az összes létező szálhoz tartozó lesz és egy DLL-ben adottnak kell lennie

A DLL –nek van egy opcionális belépési pontja, azért opcionális mert a megírása nem kötelező. Amikor a rendszer elindít vagy megszűntet egy folyamatot, vagy egy szálat, akkor meghívásra kerül, a folyamat virtuális memóriaterületére betöltött DLL. A belépési pont paraméterei:

```
BOOL APIENTRY DllMain (
HINSTANCE hInstance,
DWORD dwReason,
LPVOID lParam
);
```

A hInstance paraméter tartalmazza a DLL modul leíróját.

A dwReason paraméter jelzi azt, hogy miért lett meghívva a DLL belépési pontja az okok a következőek lehetnek:

- DLL_PROCESS_ATTACH: A DLL első alkalommal töltődik be a hozzáadott folyamatba. A DLL csak akkor kapcsolódik hozzá az aktív folyamat címterületéhez, ha az TRUE-val tér vissza, különben az alkalmazás által használ LoadLibrary függvény NULL értéket ad vissza.

- DLL_PROCESS_DETACH: A DLL lekapcsolódik a hívó folyamat címterületéről. A visszatérési érték figyelmen kívül van hagyva.

- DLL_THREAD_ATTACH: Az aktuális folyamat új szálat hoz létre. A visszatérési érték figyelmen kívül van hagyva.

- DLL_THREAD_DETACH: A szál befejezi a működését az aktuális folyamatban.

Ha az dwReason értéke DLL_PROCESS_ATTACH, akkor az lParam értéke NULL, ha a DLL dinamikusan lett betöltve, és nem NULL statikus betöltésnél

Ha a dwReason értéke DLL_PROCESS_DETACH, akkor az lParam értéke NULL lesz, ha a DLL –t a FreeLibrary függvény hívta meg, vagy a DLL betöltése meghiúsult, minden más esetben pedig nem NULL lesz az értéke.

ARCHITEKTÚRA MEGTERVEZÉSE ÉS TESZTELÉSE

Egy megfelelő program architektúra megtervezése igen időigényes feladat, de ez a későbbiekben igencsak megtérül. Kezdő programozók gyakran beleesnek abba a hibába, hogy mindent „beledrótoznak" a kódba, és/vagy a felhasználói felületbe ágyazzák be az üzleti logikát. Lehet, hogy az adott alkalmazás igen gyorsan elkészül, de az eredmény „spagettikód" lesz, melynek későbbiekben történő módosítása igencsak fájdalmas lehet, és a tesztelése is igen nagy kihívás. Célszerű programunkat úgy megtervezni, hogy az könnyen bővíthető, tesztelhető alkalmazás legyen. Az idők során jó néhány tervezési minta megszületett, melyek megkönnyítik, vagy éppen megnehezítik (a design pattern-ekkel csak ismerkedőknél

túlbonyolított kódot eredményezhet) egy jól megtervezett architektúra felépítését. A programtervezési minták külön csoportokba sorolhatóak:

- létrehozási minták
- strukturális minták
- viselkedési minták
- architekturális minták
- stb.

Vegyük például az architekturális mintákat. Ezen minták segítségünkre lehet egy alkalmazás „vázának" megtervezésében. Több ismertebb architekturális tervezési minta létezik:

- MVC (Model-View-Controller)
- MVP (Model-View-Presenter)
- MVVM (Model-View-ViewModel ez WPF specifikus).

A közös ezekben a mintákban, hogy az alkalmazás felépítését legalább 3 rétegre választják szét:

- megjelenítési réteg
- köztes réteg
- üzleti logika/ adatelérési réteg

Megjelenítési réteg:

A megjelenítési réteg az üzleti logika megjelenítését végzi. Egy modellhez többféle nézet is tartozhat.

Köztes réteg:

A megjelenítési rétegtől érkező eseményeket közvetíti a modell felé. A modelltől érkező választ megjeleníti a megfelelő formában a felhasználói felületen vagy egy eseményt közvetít a megjelenítési rétegnek, hogy olvassa fel a változásokat a modelltől (ez a rész mintánként eltérhet). Általánosan elmondható, hogy az egyetlen feladata a vizuális felület állapotának tárolása, esetleg módosítása, adatok típuskonvertálása (mintafüggő).

Modell (üzleti logika/adatelérési réteg):

Itt foglal helyet, az a kód, amely meghatározza az alkalmazás működésének logikáját, és az adatokkal végzett számítások kódja is itt található. Egyes modellekben az adatelérési réteget

(DAL) is a modellhez sorolják (de ettől még 2 független jól elkülönülő réteg marad). Az adatelérési réteg feladata az adattárolás módjának leválasztása az alkalmazástól. Más rétegek egy felületen keresztül kommunikálnak az adatelérési réteggel. Így az adatok tárolási módját bármikor lecserélhetjük pl.: eddig fájlban tároltuk az adatokat, most már adatbázisban szeretnénk, ha az adatelérési felületen nem változtatunk, akkor a többi rétegben semmilyen változtatást nem kell alkalmaznunk.

Lényegében a fentebb felsorolt architekturális minták is ezeket a rétegeket valósítják meg, csak más-más elnevezést használunk. A lényegi különbség a rétegek közötti viszonyban nyilvánul meg.

Nézzük meg közelebbről MVP –t mintát. Ennek a mintának 2 változata van.

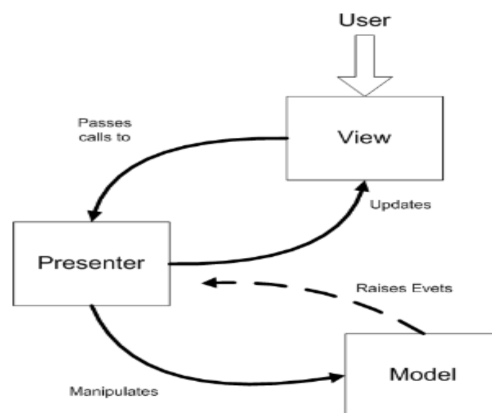
- supervising controller
- passive view

A mintában a köztes réteg, azaz a MVP –ben használatos nevén presenter egy – egy referenciát fog tartalmazni, mind a megjelenítési rétegről (view), mind a modellről (model). A következő példában egy MVP minta gyakorlati alkalmazását láthatjuk.

```
public partial class View : Form
{
    Presenter presenter;
    public View()
    {
        InitializeComponent();
        presenter = new Presenter(this);

    }

    public string PersonName
    {
        get
        {
            return txtBoxPersonName.Text;
        }
        set
        {
            txtBoxPersonName.Text = value;
        }
    }
```

```
    public string PersonAge
    {
        get
        {
            return txtBoxPersonAge.Text;
        }
        set
        {
            txtBoxPersonAge.Text = value;
        }
    }

    private void SaveClick(object sender, EventArgs e)
    {
        presenter.SavePerson();
    }
}

public class Presenter
{
    View view;
    PersonModel model;
    Service service;

    public Presenter(View view)
    {
        this.view = view;
        service = new Service();
        model = service.LoadModel();

        SetupView();
    }


    void SetupView()
    {
        view.PersonAge = model.Age.ToString();
        view.PersonName = model.Name;

    }

    public void SavePerson()
    {
        int age;

        if(Int32.TryParse(view.PersonAge, out age))
        {
            model.Age = age;
            model.Name = view.PersonName;
            service.Save(model);
        }
    }
}
```

```
    class PersonModel
    {
        DateTime birthDate;
        public PersonModel(string name, DateTime birthDate)
        {
            Name = name;
            this.birthDate = birthDate;
        }

        public string Name{ get; set; }
        public int Age
        {
            get
            {
                return DateTime.Now.Year - birthDate.Year;
            }
            set
            {
                birthDate = birthDate.AddYears(birthDate.Year - value);
            }
        }
    }

    class Service
    {
        public void Save(PersonModel m)
        {
            // Elmenti egy adatbázisba
        }

        public PersonModel LoadPersonModel()
        {
            // Egy adatbázisból történő betöltést szimulál
            return new PersonModel("Keresztes Zoltán", new DateTime(1986,
12, 16));
        }
    }
```



2. ábra MVP minta szemléltetése[1]

Mint az látható, a presenter egy referenciát tart számon a view-ról és a model-ről és, ha felhasználói interakció történik, akkor a view a kérést átadja a presenter-nek, a presenter

[1] http://www.silverlightshow.net/storage/userfiles/MVC%20common(1).png

kérést intéz a model –hez, a model-ből felolvassa az adatokat, a model eseményeinek hatására változást eszközöl a view-ban.

A rétegekre történő bontással az alkalmazás logikáját teljesen leválasztottuk a felhasználói felülettől.

A fentebb megírt példaalkalmazás minden szempontból megfelel a MVP mintának ezzel elérve azt, hogy a felhasználói felületet igény szerint bármikor testre szabhatjuk anélkül, hogy az alkalmazás üzleti logikáján változtatni kellene, de persze ettől még az alkalmazásunk nem tesztelhető. Tegyük fel, hogy tesztelni szeretnénk azt, hogy a view-ban megfelelően jelennek-e meg az adatok, persze ezt lehet próbálgatással is csinálni, lefordítjuk az alkalmazást, próbálgatással teszteljük az alkalmazást. Mini alkalmazásoknál ez még járható út lenne, de ha ennél nagyobb programot írunk érdemes mindenre unit (egység) teszteket írni. A fentebb megírt alkalmazás nem megfelelő unit tesztek írásához. A fő problémát az jelenti, hogy a presenter két osztálytól is függ. A View–tól és a Service-től ez azért probléma, mert a presenter tesztelése magával rántja a többi objektumot is. Ezért a jól tesztelhető alkalmazás létrehozása érdekében érdemes még egy igen közkedvelt módszert is megemlítenem. Ez a dependency injection (függőségi befecskendezés). Egy adott osztály, annál kevésbé tesztelhető minél több függősége van, azaz más osztályoktól függ. A fenti példában pl.: a service egy adatbázishoz kapcsolódik. A presenter egy referenciát hordoz magában erről, és egy megjelenítendő felületről. Azaz, hogyha tesztelni akarjuk a presentert kapcsolódnunk kell egy adatbázishoz, vagy hogyha módosítani szeretnénk a felhasználói felületen megjelenítendő adatokat, akkor az adatbázisban kell módosításokat végezni. A függőségi befecskendezés segítségével az eddigi problémák megoldhatóak. A refaktorált kód ezt fogja bemutatni:

```
public interface IView
{
    string PersonName { get; set; }

    string PersonAge { get; set; }
}
public partial class View : Form, IView
{
    Presenter presenter;

    public View()
    {
        InitializeComponent();
        presenter = new Presenter(this, new Service());

    }
```

```
        public string PersonName{...}

        public string PersonAge{...}

        private void SaveClick (object sender, EventArgs e){...}
    }
    public class Presenter
    {
        IView view;
        IPersonModel model;
        IService service;

        public Presenter(IView view, IService service)
        {
            this.view = view;
            this.service = service;
            model = service.LoadPersonModel();

            SetupView();
        }

        void SetupView(){...}

        public void SavePerson(){...}
    }
    public interface IService
    {
        void Save(IPersonModel m);
        IPersonModel LoadPersonModel();
    }

    public class Service : IService
    {
        public void Save(IPersonModel m){...}

        public IPersonModel LoadPersonModel(){...}
    }
```

Mint látható, ebben a példában a presenter a konstruktorában 1-1 interfészt (felületet) vár. Így már teljesen mindegy, hogy mit kap meg, csak az adott interfész legyen megvalósítva az osztályban. Ezt hívják konstruktor befecskendezésnek.

Tegyük fel, hogy egy olyan osztályt akarunk megszabadítani a függőségeitől, amely nagyon sok mindentől függ és azokban az osztályokban, amelyekkel függésben áll semmiféle kapcsolat nincs, hogy egy közös felületet adva nekik csökkentsük a paraméterek számát. Ebben az esetben konstruktor befecskendezést használva a konstruktorunk paraméterlistája nagyon sok paraméterből állna.

```
class Something

{
    public IA a;
    public IB b;
    public IC c;
    public ID d;
    public IE e;
    public IF f;
    public IG g;
    public IH h;

    public Something(IA a, IB b, IC c, ID d, IE e, IF f, IG g, IH h)
    {
        this.a = a;
        this.b = b;
        this.c = c;
        this.d = d;
        this.e = e;
        this.f = f;
        this.g = g;
    }

    // a többi rész elhagyva
}
```

Az ismertebb tervezési mintákat használva 2 lehetőség közül választhatunk:

- Az egyik a setter injection. A konstruktor létrehozása után egyesével állítjuk be a függőségeket.

```
class Something
{
    public IA A { get; set; }
    public IB B { get; set; }
    public IC C { get; set; }
    public ID D { get; set; }
    public IE E { get; set; }
    public IF F { get; set; }
    public IG G { get; set; }

    // a többi rész elhagyva
}
```

- A másik lehetőség az Abstract Factory használata.

```
interface IAbstractFactory
{
    IA CreateA();
    IB CreateB();
    IC CreateC();
    ID CreateD();
    IE CreateE();
    IF CreateF();
    IG CreateG();
    IH CreateH();

}
```

```
class AbstractFactory : IAbstractFactory
{
    #region Implementation of IAbstractFactory

    public IA CreateA()
    {
        return new A();
    }
    public IB CreateB()
    {
        return new B();
    }
    // a többi rész elhagyva
    #endregion
    }

class Something
{
    public IA a;
    public IB b;
    public IC c;
    public ID d;
    public IE e;
    public IF f;
    public IG g;
    public IH h;
    IAbstractFactory factory;

    public Something(IAbstractFactory factory)
    {
        a = factory.CreateA();
        b = factory.CreateB();
        c = factory.CreateC();
        // stb...
    }
}
```

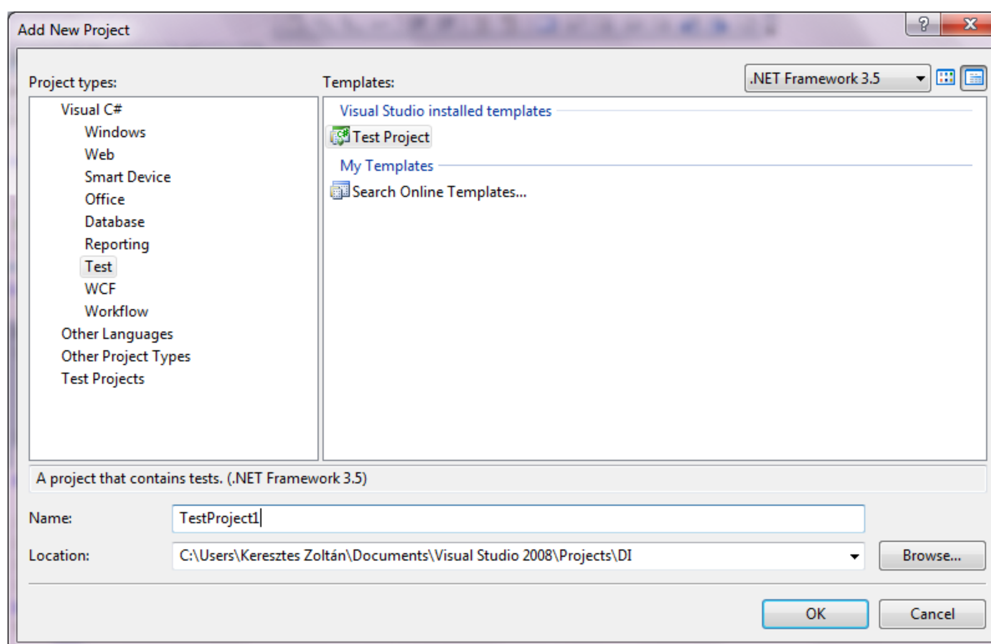
Abstract Factory minta használatának még egy hatalmas nagy előnye, hogy futási időben is létrehozhatunk újabb példányt az adott függőségből, ezt eddig a többi minta használatával nem tehettük volna meg. Ezért ha kevés függőség van, de valamelyiknél szükséges futási időben újabb példányt létrehozni, bátran alkalmazzunk az adott függőségre Abstract Factory mintát.

Ezután már könnyedén tesztelhetővé vált az alkalmazás, persze ennek is ára van a nem tesztelhető alkalmazásokhoz képest, többletkód keletkezik, egy nagyobb alkalmazásnál rengeteg interfészt, absztrakt osztályt (helyzetfüggő) kell létrehozni, ezért ajánlatos valamilyen dependency injection framework-öt használni, mely megkönnyíti egy jól tesztelhető alkalmazás létrehozását.

Tesztelésnél 4 különböző kifejezés használatos. Sokan rosszul használják a kifejezéseket, de minden a helyére kerül, hogyha elolvassuk Martin Flower nagyszerű cikkét [38].

- **Dummy:** Az adott objektumtesztelésnél át lesz adva a tesztelendő osztálynak (pl.: a konstruktor befecskendezés miatt), de használatára az adott tesztesetben nem kerül sor
- **Fake:** Nagyon jó példa erre, hogyha egy adatbázis kapcsolódás helyett egy listából nyerjük az információkat.
- **Mock:** Egy másik objektum interfészét utánzó csak nyomkövetési és tesztelési célokra szánt objektum, értékeit nem őrzi meg az egész teszt során.
- **Stub:** A stub egy olyan mock objektum, mely képes megőrizni a beállított értékeket az egész teszt alatt, de az értékeket csak a property-k, és publikus adattagokra őrzi meg.

Elérkeztünk a tesztelési fázisba. Tesztek írásához a Visual Studio 2008 Professional változata beépített támogatással rendelkezik. Adjunk hozzá a tesztelendő projekthez egy teszt projektet.



3. ábra: Teszt projekt hozzáadása a solution-höz

A teszt projektben referenciaként vegyük fel referenciaként a tesztelendő projektet. Így már hozzáférhetünk a tesztelendő projekt osztályaihoz. A legnagyobb probléma, az hogy minden egyes teszthez létre kellene hozni újabb mock, stub, fake, dummy objektumokat, hogyha a korábbi teszteseteket is meg szeretnénk őrizni. Ez rengeteg munkával járna. Ezért célszerű valamilyen mock framework –öt használnunk, pl. a Rhino mock framework -öt. Ezzel a tesztesetekben dinamikusan hozhatjuk létre ezen objektumokat, így nem kell

mindenhez külön-külön legyártani. Adjuk hozzá projektünkhöz a Rhino mock framework referenciáját is.

```
 [TestMethod]
public void TestMethod1()
{
    // létrehozunk egy mockrepository-t
    var mockRepository = new MockRepository();
    // létrehozunk dinamikusan egy olyan objektumot, amely
implementálja az IView -t
    //mivel a stub az egész teszt alatt megőrzi a nyílvános
   // property értékeit ezt kell használnunk, hogy később le tudjuk
   // kérni az értékeit
    var view = mockRepository.Stub<IView>();
    // létrehozunk dinamikusan egy olyan objektumot, amely
implementálja az IService -t
    var service = mockRepository.StrictMock<IService>();
    var name = "Nagy Gábor";
    var person = new PersonModel(name, new DateTime(1972, 11, 10));
    // beállítjuk, hogy a service LoadPersonModel metódusa mivel
térjen vissza
    // ezt rögzíti a mockrepository ez csak egyszeri hívásra szól!
   // ha többször is szükségünk van rá akkor azt külön rögzíteni
   // kell különben hibát dob a teszteset
    Expect.Call(service.LoadPersonModel()).Return(person);
    // a rögzített állapot(ok) visszajátszása történik meg
    mockRepository.ReplayAll();
    // létrehozzuk a presentert
    var presenter = new Presenter(view, service);
    // ellenőrizzük az értéket
   // az assert osztály rengeteg statikus metódussal rendelkezik,
   //amellyel tesztelhetjük az értékeket
// hogyha egy Assert vizsgálat nem a várt értéket adja vissza, akkor a
tesztesetünk elbukik
    Assert.AreEqual(view.PersonName, name);
    var age = Int32.Parse(view.PersonAge);
    Assert.AreEqual(age, person.Age);        }
```

Mint látható a presenter osztályunkat anélkül tudtuk letesztelni, hogy létre kellett volna hoznunk egy formot, hozzá kellett volna kapcsolódnunk egy adatbázishoz, nem kellett

sajátkezűleg mock és stub objektumokat gyártanunk, ebben rejlik a mock framework és a unit tesz együttes használatának előnye.

Ebben a fejezetben szerettem volna betekintést nyújtani a tervezési minták használatának előnyeibe. Célszerű alkalmazásainkat úgy megtervezni, hogy a benne lehető osztályok a lehető legkevésbé függjenek egymástól. Általánosan igaz az, hogyha egy alkalmazás jól tesztelhető, akkor könnyen módosítható, karbantartható kódot írtunk. A tervezési minták nem szentírások, csak általános ajánlások, ha egy minta nem teljesen felel meg az adott körülmények között nyugodtan alakítsuk át igényeink szerint, vagy használjuk mást helyette.

4.4 PROGRAM VÉDELEMI MÓDSZEREK ISMERTETÉSE

Ha szeretnék, hogy programunkat csak azon személyek használhassák, akik erre jogosultak, azaz, akik ezen alkalmazást megvásárolták célszerű valamilyen védelmet építenünk az alkalmazásba.

Védelmi módszerek:

- Fix kulcs használat: Nagyon gyenge védelem, ha valaki tudomást szerez erről a kulcsról, ezt közzétéve a védelem már mit sem ér.

- A beírt adatoktól függő kulcs generálása: Bonyolult algoritmus segítségével a beírt adatokhoz generálódik a kulcs, melyet aktiváláskor meg kell adni. Ezzel is ugyanaz a probléma, mint az előbb említett védelemmel.

- Internetes aktiválás: A program aktiválása során egy kiszolgálóhoz fordul a kiszolgáló, ellenőrizheti a felhasználó nevét, aktiváló kulcsát, nyilvántarthatja, hogy ezzel a felhasználó névvel, kulccsal, már hányszor lett regisztrálva a program, vagy hogy egyáltalán érvényes-e ez a felhasználónév kulcs páros. Az eredményt pedig visszaküldi az alkalmazásnak.

- Véletlenszerű kulcs generálás: Az aktiválás során véletlenszerűen generálódik egy kulcs, és ezen kulcshoz kell megadni az aktiváló kulcsot, mely csak egy bonyolult függvény alkalmazásával állítható elő.

- Hardverkulcs: Egy speciális eszköz szükséges a program használatához, ha ezen eszköz a számítógéphez van csatlakoztatva, akkor a program fut, máskülönben

befejezi a működését. A speciális eszköz lehet akár egy usb kulcs is melyen egy program fut, amely kommunikál a védelem alatt álló szoftverrel, ha a kommunikáció megszűnik, az azt jelenti, hogy az eszköz nincs a géphez csatlakoztatva. Csak drágább programoknál érdemes használni, mert egy kulcs ára körülbelül 8 -10 ezer forint körül mozog és az alkalmazás minden egyes példányához külön hardver kulcsra van szükség.

- A számítógéptől függő kulcsgenerálás: A program a számítógépben található hardverinformációk által generálja a kulcsot (pl.: CPU ID, MAC ID, Merevlemez ID). Ez a kulcs minden számítógépen egyedi, aktiválás során a hardveres ujjlenyomathoz tartozó aktiváló kulcsot kell megadni.

A programokból általában készíteni szoktak korlátozott verziót is:

- Időkorlátos: Programaktiválás nélkül csak bizonyos ideig futhat. Ennek több variációja is lehet. Egyes programok azt tartják számon, hogy az adott program mennyi ideig futhat aktiválás nélkül. Más alkalmazások pedig egy időintervallumot határoznak meg, mely során az alkalmazás használható megvásárlás nélkül is.

- Az indítások számában korlátozott: Aktiválás nélkül hányszor indítható el az alkalmazás.

- Demó: A szoftverbe nem került beleépítésre az összes funkció. Aktiválásra nincs lehetőség.

Ezek a verziók azt a célt szolgálják, hogy a felhasználók a program megvásárlása előtt kipróbálhassák az adott alkalmazást. Az eddig felsorolt védelmek egy igen magas absztrakciós szinten állnak, amelyek önmagában szinte semmit se érnek. A .NET – ben írt alkalmazás fordítása után egy köztes nyelvre fordul, gyakran jelentős mennyiségű metaadatot tartalmazva. A metaadat információt ad az alkalmazás függvényeiről, osztályairól, változóiról és még sok egyéb dologról. Ezért programunk további védelem nélkül egyszerűen visszafejthető. Nagyon sok programozónak nincs ideje elmélyülni a programvédelmi módszerekben, ezért valamilyen mások által megírt védelmi szoftverre bízza a programja védelmét.

Általában a fizetős védelmi szoftverek többféle módszert kínálnak szoftvereink védelmére:

- Kód összezavarás:
    - Az alkalmazás változóit, névtereit nem nyomtatható karakterekre, rövid nem beszédes nevekre nevezi át, így megnehezítve az alkalmazás forráskódjának értelmezését.
    - Az szoftver forráskódjába olyan érvénytelen utasításokat injektál, melyek megnehezítik a kód visszafejtését, mivel a decompiler nem tudja teljes mértékben értelmezni az adott kódot.
    - A program kódjába plusz utasítás blokkokat szúr be, melyek a szoftver futás során sose futnak le, egyetlen célja visszafejtés során a kód átláthatóságának megnehezítése.
- String titkosítás: Az alkalmazásban található karakterláncok titkosítva kerülnek tárolása.
- Ildasm elnyomása: Az ildasm programnak megtiltja a program betöltését.
- Érvénytelen metaadatok beszúrása: Olyan metaadatokat szúr be a programba, mellyel a legtöbb kód visszafejtő nem tudja betölteni az adott programot.
- Natív wrapper használata: Az alkalmazás egy natív csomagolást kap csak futási időben épül fel újra az eredeti alkalmazás a memóriába, e formában visszafejthetetlen a .NET es dissamblerekkel.

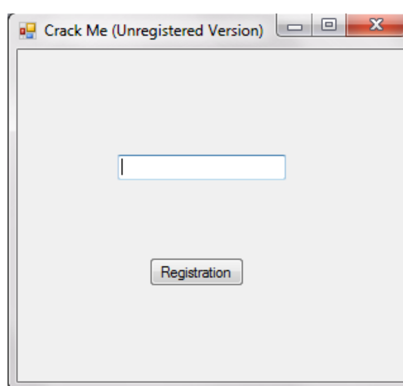
Ezen felül a legtöbb védelmi eszköz különböző plusz szolgáltatásokat nyújthatnak. Ilyen például a memória optimalizáció vagy a szoftver, különböző korlátos (időben vagy elindítások számában korlátozott) verzióinak elkészítésében nyújtott segítség.

4.5 PROGRAMFELTÖRÉSI TECHNIKÁK MEGISMERÉSE

Amennyiben már megismertük a programvédelmi mechanizmusokat célszerű időt szánni a programtörési módszerek megismerésére is, hogy legalább fogalmunk legyen arról, hogy mennyire biztonságos az általunk használt védelmi mechanizmus és köztudott, hogy a legprofibb védelmi mechanizmusok tudói maguk a crackerek. Azt természetesen kijelenthetjük, hogy **feltörhetetlen program nem létezik**. A crackerek dolgát maximum csak megnehezíteni lehet. Van egy szabály mely szerint, **csak olyan programot érdemes feltörni, melynek a feltörésére fordított költség kisebb, mint az egész program újraírása**.

Vegyük most azt az alap esetet, hogy programunk csak magas absztrakciós szintű védelmet tartalmaz, az alkalmazás forráskód terén nincs védve. A feltörés nagyon leegyszerűsödik, egyszerűen csak vissza kell fejteni a program forráskódját, és azt a megfelelő helyeken módosítva máris sikerült a feltörés. Lássunk erre most egy egyszerű példát.

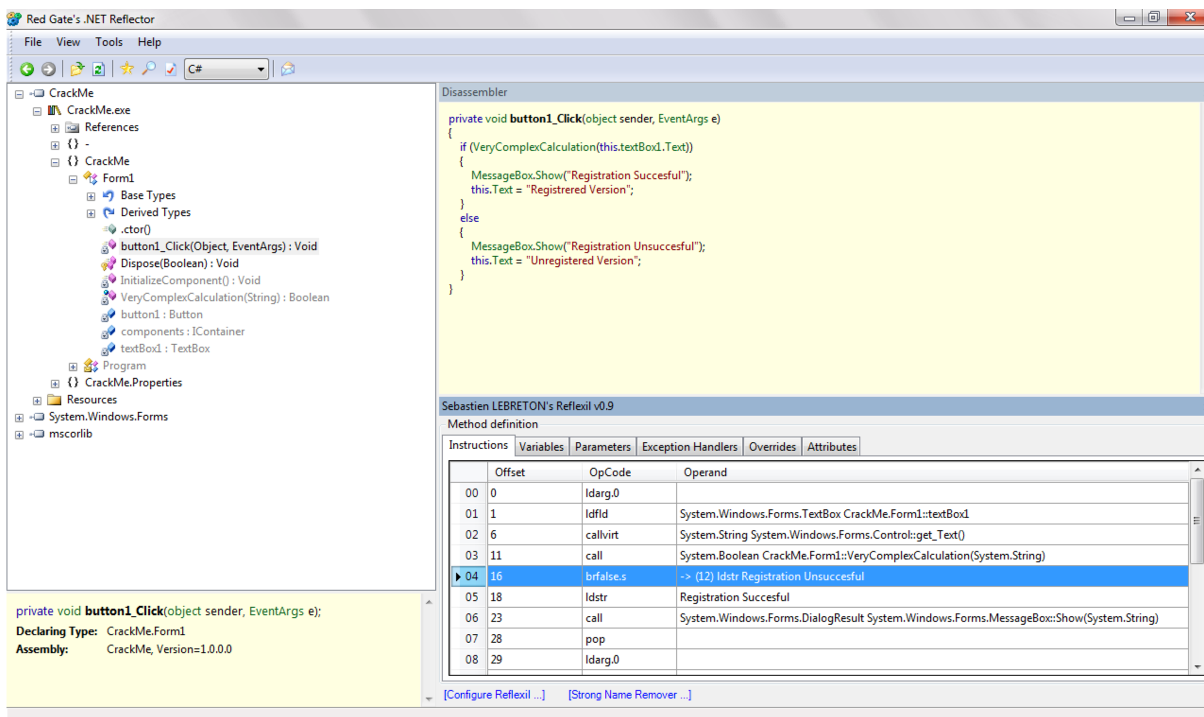
Ezen alkalmazás regisztrációja során egy „nagyon bonyolult" metódus végzi a regisztrációt. Ha a regisztráció során helyes kódot adunk meg, akkor a regisztráció sikeres, különben a regisztráció sikertelen.



4. ábra: Az általunk feltörésre kerülő program

Ha a program semmilyen védelemmel nem rendelkezik, akkor ezen alkalmazás aktiválásának kijátszása nagyon egyszerű. A visszafejtésre több program is a rendelkezésünkre áll. Ezek közül az egyik a Lutz Roeder által fejlesztett .NET Reflector. Ezen alkalmazással legtöbb esetben szinte ugyanazt a forráskódot kaphatjuk vissza, amely lefordításra került. A futtatható állományt a Reflectorba betöltve visszakapjuk az alkalmazás forráskódját. Az szoftver forráskódját különböző nyelveken is megtekinthetjük, mivel több

nyelv és létezik melyen .NET es alkalmazást fejleszthetünk és e nyelveken írt kódok fordítás során ugyanarra a köztes nyelvre fordulnak.



5. ábra: A Reflector program használat közben

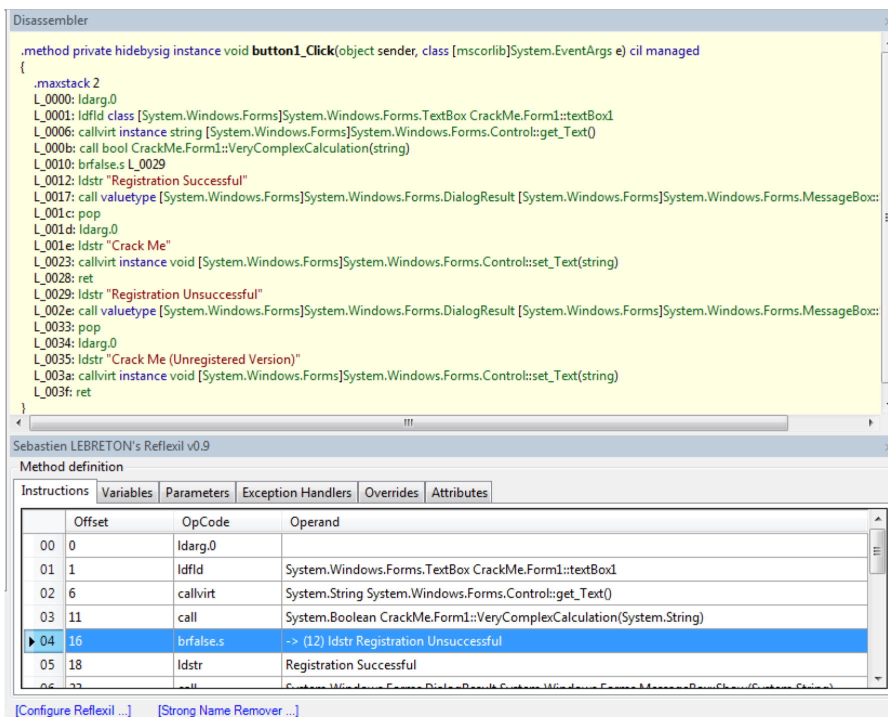
Mint látható az alkalmazás során a textbox tartalma átadódik egy „bonyolult” számításokat végző függvénynek, ha a metódus igaz értéket ad vissza, akkor a regisztráció sikeres ellenben sikertelen. A crackerek általában 2 lehetőség közül választhatnak:

- Az aktiválást végző metódus forráskódját megnézve előállítanak egy sorozatszám generátort (serial code generator).
- Az alkalmazás kódját módosítva megváltoztatják annak működését (patching).

Tegyük fel, hogy az alkalmazás a már korábban említett hardveres ujjlenyomat digitálisan aláírt változatával lehet aktiválni. Ebben az esetben a sorozatszám generátor előállítása szinte lehetetlen, mivel ekkor az alkalmazás nem tartalmaz minden információt, mely ehhez szükséges (privát kulcs). Ekkor a crackereknek csak a második lehetőségre nyílik módjuk.

A reflektorhoz rengeteg plugin áll a rendelkezésünkre. Ezek közül az egyik a Reflexil, mellyel az alkalmazáshoz tartozó IL kód módosítható, így megváltoztatva az annak

működését szerencsére a plugin használatához nem kell ismernünk az összes IL utasítást, mivel a funkciójukat könnyedén megismerhetjük, hogyha az egeret az utasításra visszük.

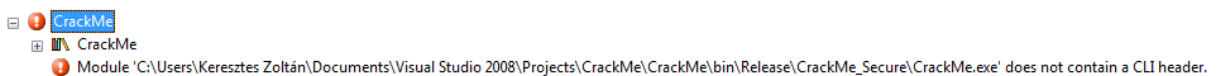


6. ábra: A visszafejtett kód IL nyelven

Az alkalmazás által tárolt egyszerű típusok a veremben (stack), míg a referencia típusok a halomban (heap) tárolódnak. A veremben tárolt értékek közül az alkalmazás mindig a verem tetején található értékkel tud dolgozni. A legutoljára betöltődő érték mindig a verem tetejére kerül. Ezen metódus betöltése során először a beállítódik a maximális verem méret, ami jelen esetben kettő. Ezután a metódus paraméterének helyére fog mutatni (a paraméter referencia típus azaz a heap-en tárolódik) a verem tetején elhelyezkedő érték. Ezt követően a textbox tartalma a verembe töltődik. Ezután a VeryComplexCalculation függvény a veremben legfelül lévő értéket megkapja paraméterül (jelen pillanaban a textbox tartalmát) és a visszatérési értékét a verem tetejére helyezi el. A brfalse.s utasítás false értéket kap, akkor a vezérlést átadja az utasítás után szereplő címnek. Mint látható, ekkor a nem jó aktiváló kulcshoz tartozó utasítások futnak le, majd azután a függvény a ret paranccsal visszatér a hívó környezethez. Ebben a kódban egyetlen egy utasítását kell módosítani ahhoz, hogy a program működését megváltoztatassuk, brfalse.s utasítást lecseréljük brtrue.s re, ekkor a rossz aktiváló kód megadása esetén VeryComplexCalculation függvény false értéket ad vissza és mivel a brtue.s false nem igaz, így nem ugrik az utasítás után megadott címre. Ekkor az utasítást

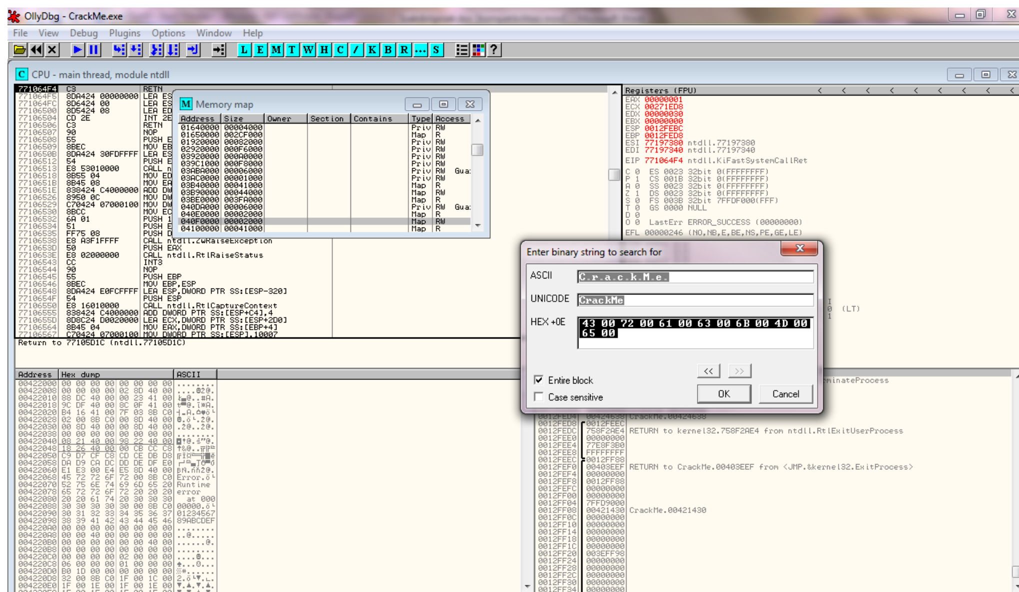
követő, a program aktiválását végző parancsok futnak le és ezután a ret utasítással visszatér a hívó környezethez. Ha a módosítást elvégezzük az alkalmazáson és újrafordítjuk, akkor az alkalmazást anélkül aktiválhatjuk, hogy tudnánk a megfelelő aktiváló kulcsot. A programot akkor tudjuk aktiválni, ha nem megfelelő kulcsot adunk meg, kizárólag jó aktiváló kulcs megadása esetén nem tudjuk aktiválni a programot.

Ezek után most nézzük meg, hogy mi történik akkor, amikor a programozó egy védelmi szoftverre bízza alkalmazását. Ebben az esetben lényegében fogalma sincs, hogy mi történik a háttérben, milyen mechanizmussal történik valójában a programjának a védelme. A crackerek szeretnek „nagy halakra vadászni" ezért kifejezetten ingerli őket ez a fajta univerzális rendszer. Onnan kezdve, hogy egy alkalmazást sikerült feltörni az adott verziójú, licenc menedzselt programok halmazából elméletileg az összes több program feltörhető ugyanazon (de maximum paraméterekben különböző) módszerrel az univerzális védelmi mechanizmusok leírásának (cracking tutorials) megjelenése több olvasótábor tud magáénak, ezért több ember ismeri meg a védelmi rendszer működését rendszerközeli szinten. Az alkalmazás a jobb védelmi szoftverek közé tartozó Eziriz cég által fejlesztett .NET Reactor. A szoftver tartalmazza az összes, védelmi funkciót mely korábban felsorolásra került. Ha alkalmazásunkat e szoftverrel védjük le, a következőt tapasztalhatjuk, ha megpróbáljuk a Reflectorral megnyitni az alkalmazást:



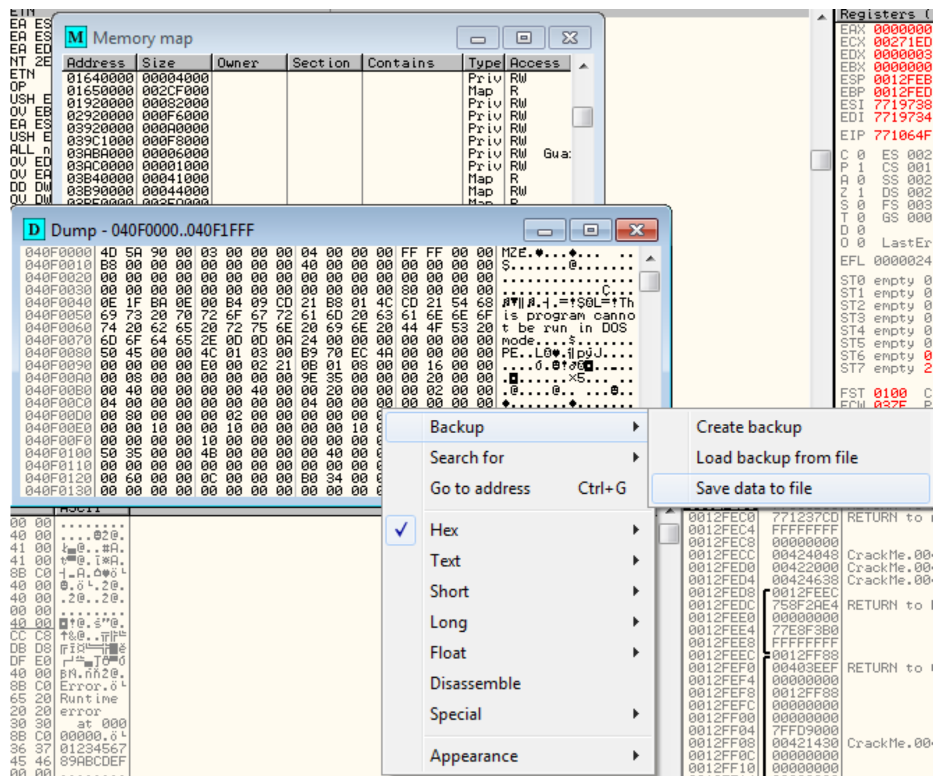
7. ábra: A Reflector program nem képes betölteni a szerelvényt

Mint látható az alkalmazást nem sikerült visszafejteni, mivel az alkalmazás egy natív csomagolást kapott. A legnagyobb probléma ezzel a fajta védelemmel, hogy a program futtatása során a becsomagolt alkalmazás eredeti formájában a memóriába kerül (mivel a JIT-nek valahogyan futási időben értelmeznie kell). Különböző debugger programokkal (pl OllyDbg, SoftIce) futási időben megkereshetjük a memóriában az eredeti alkalmazást, ezt kimentve pedig máris kijátszottuk ezt a védelmet. Természetesen az alkalmazás észlelheti, hogy debuggolni próbálják, erre vannak különböző WINAPI hívások, és ekkor leállíthatja önmagát, persze e fajta védelemnek is megvan az ellenszere, ami például az OllyDbg esetén egy plugin formájában elérhető, mellyel elrejthető az alkalmazás.



8. ábra: Az Ollydbg program használata

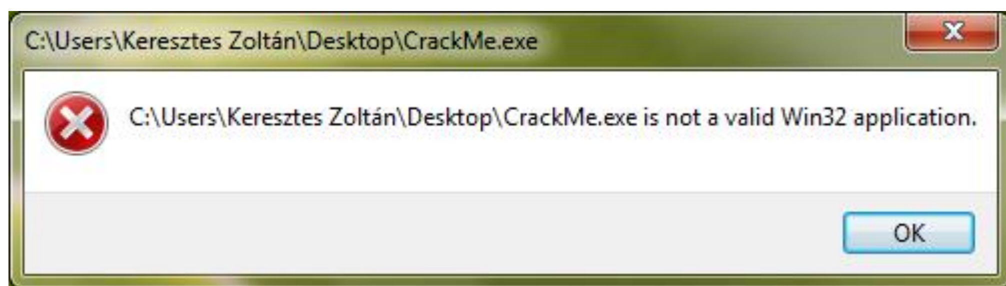
A memóriában megkeressük, az alkalmazásban található valamelyik szót. Mivel a memóriában a szövegtitkosítással ellátott alkalmazás is eredeti formájában található ez se okozhat problémát. Minden futtatható alkalmazás első 2 byte-jának értéke MZ ez Mark Zbikowski nevének rövidítése, aki az MS-DOS egyik egykori fejlesztője volt. Természetesen napjainkban már a legtöbb alkalmazás, nem DOS alapú rendszeren fut, de ez a rész kompatibilitási okokból benne maradt a futtatható állományokban. A .NET Reactor által védett szoftverek futási időben több fájlt is létrehoznak a memóriában, ezért különböző memóriaterületeken is megtalálható a keresett szó. Ezért nekünk azt kell keresni, ahol az adott memóriaterület az MZ értékkel kezdődik.



9. ábra: Az Ollydbg program használta

Ezt a memóriarészt egy fájlba kimentve máris megkapjuk az szoftver csomagolatlan verzióját.

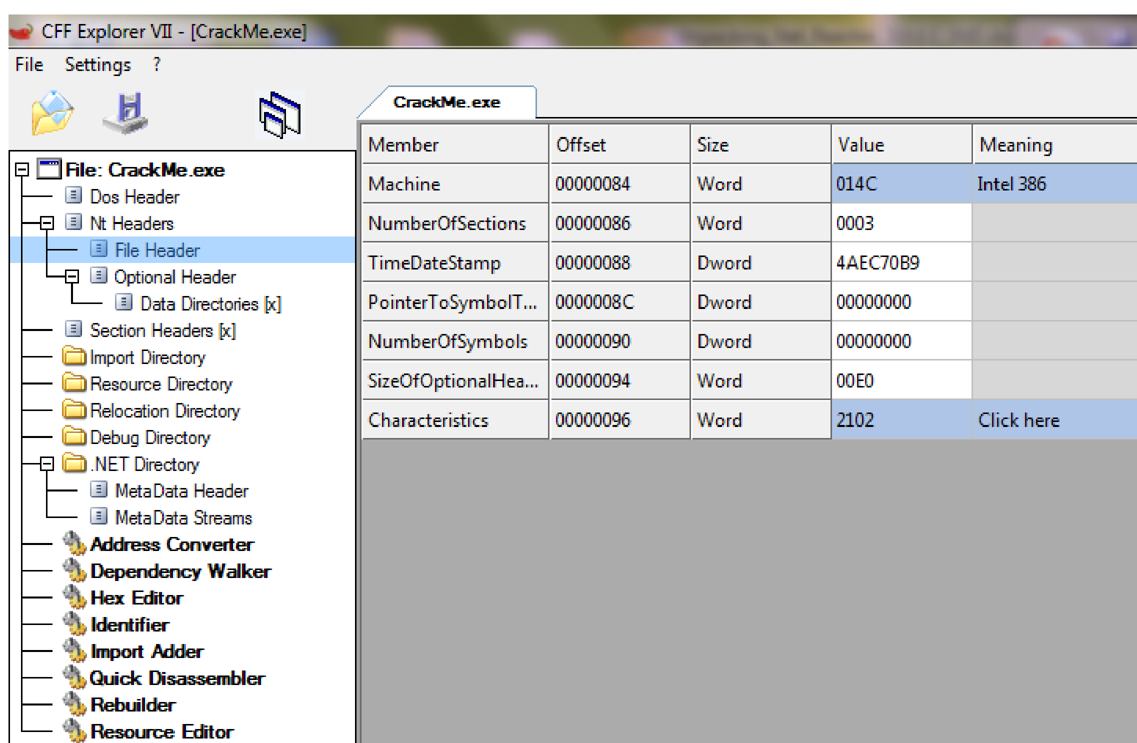
Ha a memóriából kimentett változatot megpróbáljuk elindítani, akkor a következőt tapasztaljuk:



10. ábra: Hibaüzenet érvénytelen PE esetén

Mivel az védett alkalmazás érvénytelen adatokat tartalmaz a fejlécében. A legkönnyebb dolgunk akkor van, ha rendelkezésünkre áll az eredeti alkalmazás. Ebben az esetben jóval könnyebb az érvénytelen adatok felfedezése. Ha a saját alkalmazásunkat sikerül helyreállítani, akkor ezek után bármelyik ezt a védelmet használó szoftver feltörésére képesek leszünk.

A program fejlécének módosítása a CFF Explorer nevű alkalmazással történt. Ez az első olyan PE szerkesztő, mely teljes mértékben támogatja a .NET file formátumot is.



Member	Offset	Size	Value	Meaning
Machine	00000084	Word	014C	Intel 386
NumberOfSections	00000086	Word	0003	
TimeDateStamp	00000088	Dword	4AEC70B9	
PointerToSymbolT...	0000008C	Dword	00000000	
NumberOfSymbols	00000090	Dword	00000000	
SizeOfOptionalHea...	00000094	Word	00E0	
Characteristics	00000096	Word	2102	Click here

11. ábra: A CFF program használat közben

A memóriából kimentett képfájl a következő érvénytelen adatokat tartalmazza. A alkalmazás file fejlécben a karakterisztika DLL –re van állítva.

Érvénytelen metaadat RVA(Relative Virtual Addess) kezdő címet tartalmaz a fejlécben. A tényleges cím a BSJB karaktersorozattól kezdődik. A BSJB azon néhány fejlesztők nevének kezdőbetűjéből áll össze, akik részt vettek a .NET metaadat engine kifejlesztésében. Az cím konvertálóban megkeresve ezt a karaktersorozatot megkapjuk az érvényes kezdőcímet.

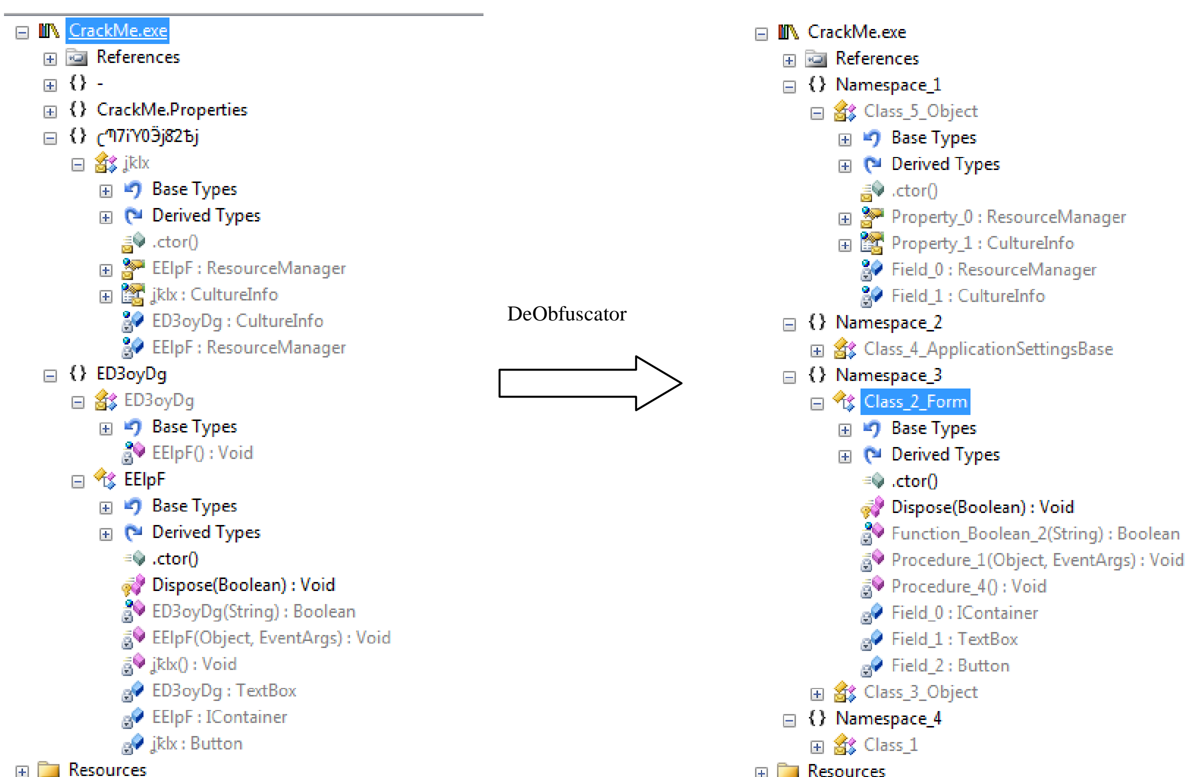
Metaadat méret = Import Directory RVA – Metaadat RVA

Érvénytelen major és minor verziók a fejlécben.

Az érvényes adatok:

- MajorRuntimeVersion = 2 and MinorRuntimeVersion = 5 .NET Framework 2.0 -ban
- MajorRuntimeVersion = 2 and MinorRuntimeVersion = 0 .NET Framework 1.1 –ban

Ezek után még néhány módosítást elvégezve, már betölthető lesz az alkalmazás a Reflectorba és ehhez hasonló kép fogad minket:



12. ábra: Az eredeti és a DeObfuscatorral visszafejtett kód közötti különbség

Mint látható sokkal olvashatatlanabb a kód, mint mielőtt még levédtük ez főleg nagyméretű programoknál jelenthet problémát, de persze erre is van megoldás, léteznek olyan alkalmazások melyek az összezavart kódot megpróbálják visszaalakítani értelmezhetőbb formába ezzel megkönnyítve a visszafejtők munkáját. Jelen pillanatban is fejlesztés alatt áll, egy olyan szoftver, mellyel a védett alkalmazásokból könnyedén eltávolíthatjuk azon kódrészleteket, melyet a védelmi szoftver helyezett el az alkalmazásunkban, annak

reményében, hogy a kódvisszafejtőket a többletkód összezavarja a program neve Native Blocks.

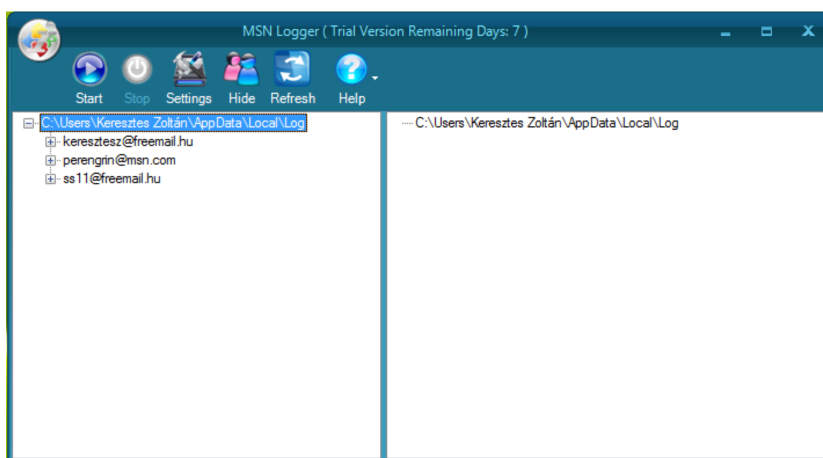
A programunkat feltörni akaró személyeknek, természetesen nem kell az egész programot átlátniuk ahhoz, hogy azt feltörjék, a crackerek olyan kódrészleteket keresnek, amely a program regisztrációjához kapcsolódik. Tegyük fel, hogy ha regisztráció során rossz kulcsot adunk meg, akkor felugrik egy ablak, mely visszajelzést ad a felhasználó számára, de ez nem csak a felhasználónak ad visszajelzést, hanem a cracker-nek is, hogy hol kell keresni a regisztrációs folyamatot, ezért érdemes elkerülni regisztrációs résznél a visszajelzés alkalmazását.

Természetesen nem csak a .NET Reflector visszafejtéséhez léteznek leírások , hanem az összes nevesebb védelmi eszközhöz, ha szeretnénk alkalmazásunkat levédeni célszerű valamely nevesebb protektorok, packer közül a legújabb verziószámút választani, melyhez programunk kiadásának pillanatában még valószínűleg nem létezik kicsomagolási leírás.

5. A MEGVALÓSÍTÁSRA KERÜLT PROGRAM BEMUTATÁSA

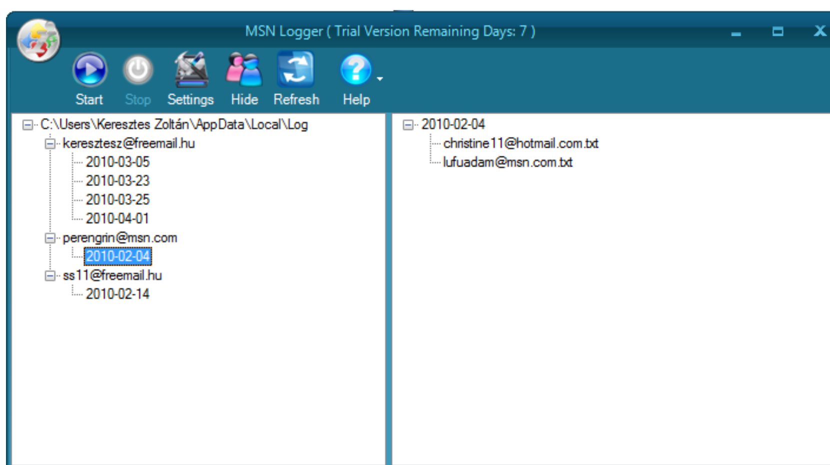
5.1 A SZOFTVER FELHASZNÁLÓI FELÜLETE

Alapesetben a felhasználót a program elindítása után a következő kép fogadja:



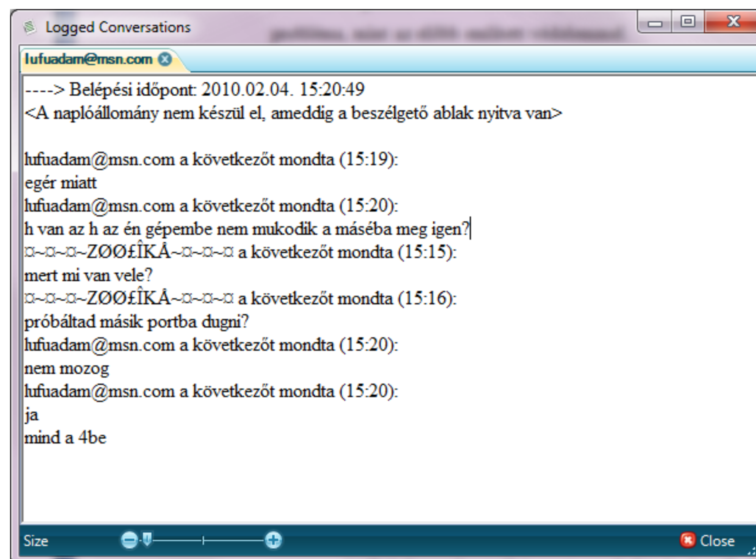
13. ábra: Az MSN Logger indítás utáni képe

Az alkalmazás jelen pillanatban 2 nyelvet támogat az angolt és a magyart. A bal oldali faszerkezetben a felhasználók nevei láthatóak, akik bejelentkeztek az MSN Messengerbe, miközben a program futott. Ezt lenyitva tekinthetőek meg a naplózott beszélgetések dátum szerint kategorizálva.

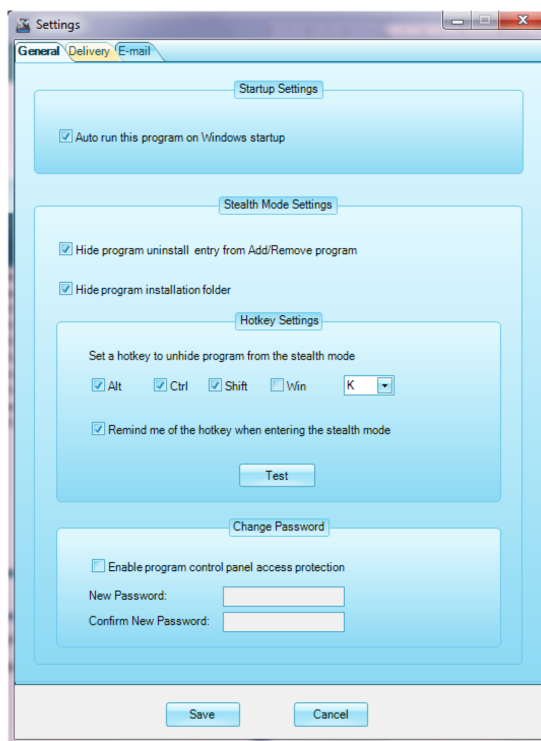


14. ábra: Az MSN Logger használat közben

A jobb oldali faszerkezetben, azt tekinthetjük meg, hogy az adott napon, a felhasználó kikkel beszélgetett. Valamelyiket kiválasztva elolvashatjuk az adott felhasználóval történő beszélgetést.



15. ábra: Az MSN Loggerben lévő naplóállományok megtekintése



16. ábra: Az MSN Logger beállításai

A programhoz tartozik egy beállítások menüpont, melybe a programot igényeinkre szabhatjuk.

A beállítható funkciók a teljesség igénye nélkül:

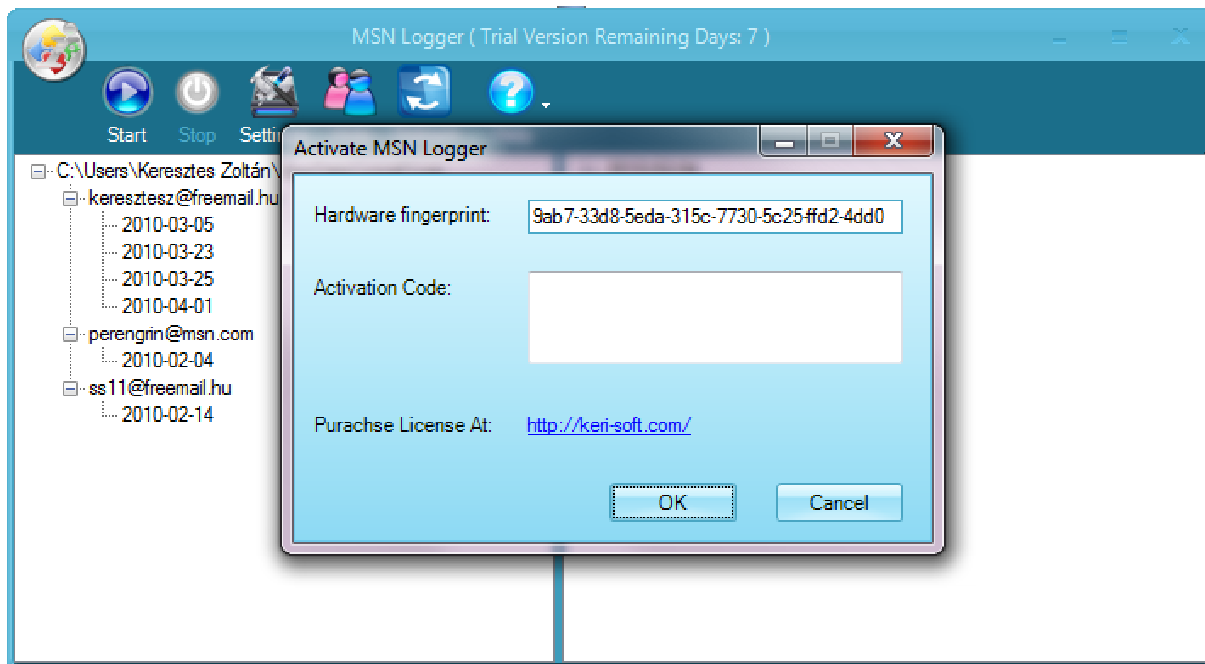
- A program a Windows rendszerbe történő bejelentkezés után automatikusan induljon el.
- A rejtett módból történő előhozáshoz milyen billentyűkombináció tartozzon.
- Ha előhozzuk a rejtett módból az alkalmazást az ablakot tartozzon e hozzá jelszó bekérése.
- Hová mentse a beszélgetéseket
- Küldjön-e emailt, és ha igen, akkor milyen időközönként

A program alapesetben 7 napig használható regisztráció nélkül, de ekkor se teljes funkcionalitással. Ha a felhasználó el szeretné rejteni a programot, hogy megfigyelje mások beszélgetését, akkor néhány percenként felugrik egy értesítés, mely figyelmeztet arról, hogy az MSN jelen pillanatban naplózva van, ilyen formában nem történhet meg semmilyen személy titokban történő lehallgatása.

17. ábra: Értesítési üzenet az MSN Logger által történő naplózásról

Mivel Windows Visa –tól kezdve bevezették, hogy az értesítéseket el lehet tűntetni, ezért kénytelen voltam egy saját értesítőt írni és nem a beépített támogatást alkalmazni. Ha valaki szeretné megvásárolni az alkalmazást, akkor azt a súgó menüponton belül a program aktiválása menüpontban teheti meg.

Amely így néz ki:



18. ábra: Az MSN Logger aktiválása

Az aktiválás után a programunk teljes funkcionalitással fog működni.

5.2 *AZ ALKALMAZÁS ARCHITEKTURÁJA*

Az alkalmazás vázaként a már korábban említésre kerülő MVP mintát alkalmaztam. A program megírása során a programozás egyik legalapvetőbb elvét a DRY –t (Do not repeat yourself, azaz ne ismételd önmagad) is próbáltam szem előtt tartani. A programban minden felhasználói felülete MVP alapokon lett megvalósítva, egyes felhasználói felület a komplexitása miatt, több MVP alapokon megvalósított komponenst is tartalmaz. A felhasználói felületekhez tartozó presenterekben jó néhány közös tulajdonság, metódus megtalálható, ezért az ismétlések elkerülése érdekében az objektumorientált szoftverfejlesztés során használatos származtatással éltem. A presenterbe kerülő alkalmazáslogikát pedig természetesen függőségi befecskendezést használva oldottam meg, amely inkább a komponens orientált szoftverfejlesztésre jellemző és, amelynek nagy előnye az objektumorientáltsággal szemben a nagyobb rugalmasság, amely tesztelésnél elengedhetetlen, és amelynek során nem kell ismerni az ősosztály felépítését, mivel általában interfész alapokon nyugszik.

Mint már korábban említésre került a program jelen pillanatban 2 nyelvet támogat, de további nyelvekkel történő bővítése se okozna gondot a jól megtervezett architektúra miatt,

lényegében csak a kulcsokhoz tartozó szavakat, mondatokat kellene felvenni egy erőforrás állományba. Nyelvváltoztatásról az egész rendszer azonnal értesül, mivel a fordítást a különböző nyelvekre egy olyan osztály végzi, amelyből futási időben kizárólag egy példány létezhet(Singleton pattern) és, ha ebben az osztályban nyelvet változtatunk, akkor a nyelvváltoztatásra feliratkozott összes osztály erről értesülni fog. Függőségi befecskendezés során a singleton pattern alkalmazása miatt minden presenter osztály ugyanazt a fordítást végző objektumot kapja meg, így ha valamely osztály a nyelvváltást igényeli, akkor arról az összes presenter osztály értesülni fog és ekkor elkéri a fordító osztálytól a megfelelő szövegeket és végrehajtja a módosításokat a view-on.
Megjegyzés: A eseményekre történő feliratkozással vigyázni kell, ha egy olyan osztály eseményére iratkozunk fel, amely létrejötte után a program futásának végéig létezik (pl. singlenton pattern alkalmazásával könnyen kerülhetünk ilyen helyzetbe). Akkor az annak az osztálynak az eseményeire feliratkozott osztályok soha nem fognak felszabadulni a program futása alatt még, hogyha már nincs is rájuk szükség, mert marad egy erős referencia az objektumra a rendszerben lévő statikus osztály miatt és így a Garbage Collector röviden GC nem szabadítja fel futási időben. Mint tudjuk a .NET es programokban nincs normál értelemben vett „Memory Leak”, de ha nem vigyázunk könnyen érhetünk el olyan állapotot, hogy futás közben iszonyatosan sok memóriát foglaljon a program. A megoldás weakreferece használata, mellyel lényegében egy proxy –t csinálunk a statikus osztály és a feliratkozandó osztály közé. A feliratkozandó osztály a weakrefencen keresztül fog feliratkozni. A weakreference nem képez erős referenciát az objektumra, így figyelni tudja, hogy mikor szűnik meg az objektum, ekkor leiratkozik az eseményről és így a weakreference is megszűnik.

A beállítások menüpontban a tab fülek esetleges későbbi bővítéséhez szinte, semmit se kellene módosítni, a kódon, mivel úgy van megoldva, hogy a fő rész az egy shell (burok), amelyhez csak olyan usercontrolok adódhatnak hozzá(hozzáadáskor létrejön egy új tab fül), melyek megvalósítanak egy bizonyos interfészt. A shellnek semmit se kell tudnia a usercontrolokon megvalósított funkciókról(a usercontrolok is MVP minta alapján készültek el). Az interfészt implementáló usercontrolloknak egy Save és egy Validation metódussal kell rendelkeznie és még egy névvel ami a tabfülön fog megjelenni. Hogyha rákattintunk a mentés gombra, akkor a shell az összes usercontroltól lekéri, hogy érvényes-e a rajta lévő információ,

hogyha ez mindegyikre teljesül, akkor kiadja a Save parancsot az összes usercontrolnak, hogy hajtsák végre a mentést.

5.3 JOGOSULTSÁGOK KEZELÉSE

Elmúltak azok az idők (XP és azelőtti rendszerek), amikor is a futtatott programok mindenhez hozzáférhetett, amelyhez a felhasználó is hozzáfér. Gondos figyelmet kell fordítani arra, hogy az általunk késztett program lekezelje azokat a szituációkat, melyek során a rendszer megtagadja egyes erőforrásokhoz történő hozzáférést. Ha valamely erőforrás használatára mindenképpen szükségünk van, akkor előre intézkedjünk arról, hogy az alkalmazás biztosan hozzáférhessen az adott erőforráshoz. Az olyan programozók, vagy inkább mondjuk úgy, hogy magukat programozónak nevező személyek, akik úgy gondolják, hogy mindenki rendszere teljesen azonos az övékével, amelyen a fejlesztést végezték, programjuk kiadása után nagy meglepetésben lehet részük. Sok felhasználó fog arra panaszkodni, hogy az adott alkalmazás nem működik megfelelően, pl. már indulásnál kritikus hibával leáll. Az ok egyszerű: nem megfelelő jogosultság kezelés. Windows Vista-tól kezdve, mint az már korábban is említésre került egyes funkciók eléréséhez megemelt jogosultság szükségeltetik. A megírásra került alkalmazásban is akad néhány olyan funkció, melyhez elengedhetetlen az olyan erőforrásokhoz történő hozzáférés, amelyet alapértelmezetten a rendszer nem engedélyez számunkra.

Pl.: A felhasználó az alkalmazásban szeretné beállítani azt, hogy az alkalmazás automatikusan induljon el a rendszer indításakor ehhez a registry-ben egy új kulcsot kell felvenni a megfelelő helyre. A normál módú folyamatok férhetnek hozzá a registry-hez. Ehhez megemelt jogosultsági szint szükséges.

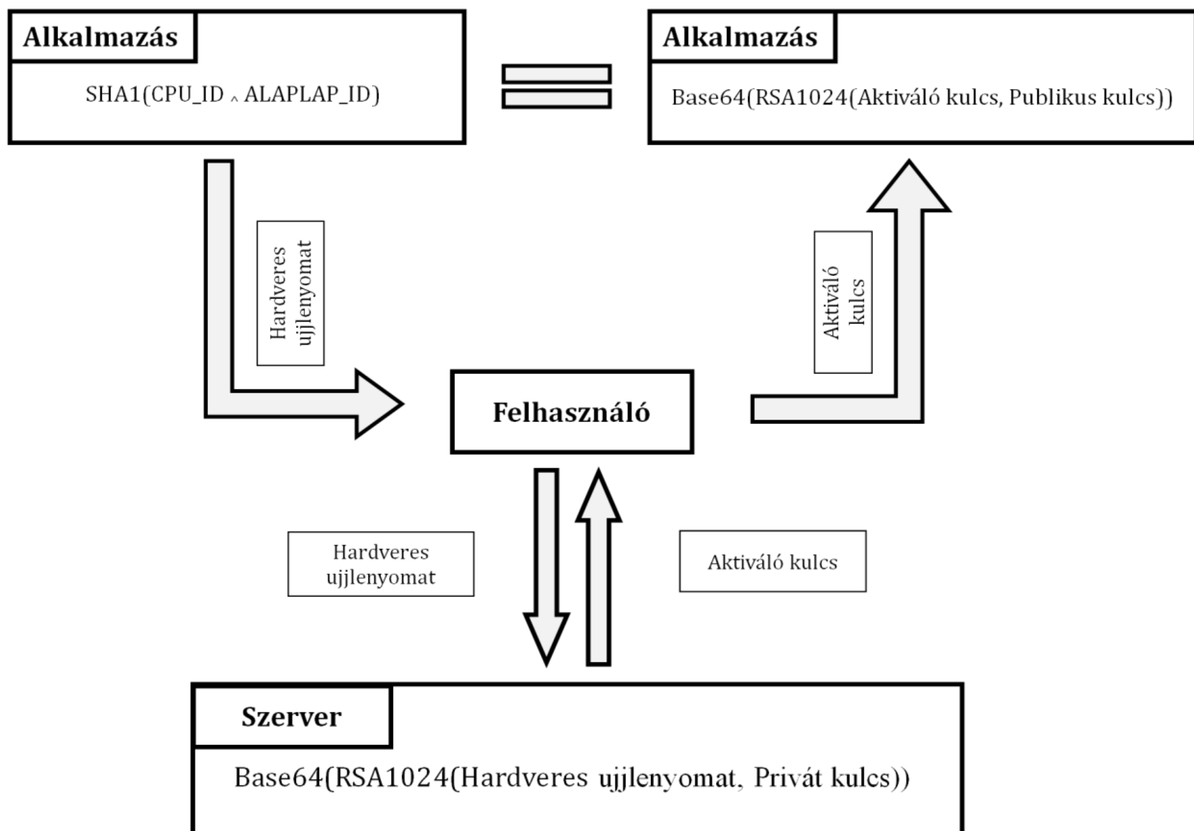
Megemelt szint kérésének 3 lehetséges módja van:

- A programot leállítjuk és megemelt jogosultsági szintet kérve újra elindítjuk.
- Egy másik programba kitesszük ezeket a funkciókat, és ha szükséges azt elindítva megemelt jogosultsági szintet kérve beállítjuk az adott funkciót.
- A program telepítésekor, olyan COM objektumot(kat) regisztrálunk a rendszerbe, amelyet rá lehet venni megemelt jogosultsági szint kérésére. Talán ez a legelegánsabb módja megemelt jogosultsági szint igénylésének.

Eddig csak arról esett szó, hogy más program COM objektumait hogyan lehet meghívni, de nem csak más programok COM objektumait használhatjuk, hanem sajátot is készíthetünk,

akár C#-ban is. Ha valamely program ezt meghívja, akkor a runtime létrehoz egy burkoló objektumot, hogy képes legyen a külvilág felé úgy kommunikálni, mint egy rendes COM objektum. Ez lényegében annak a burkolóosztálynak az ellentéte, amikor mi hívunk meg a programunkban egy COM objektumot. Az általam írt alkalmazás ezt a módszert használja. Hogy, hogyan hozhatunk létre COM objektumokat C# nyelven? Kérdésünkre az alábbi cikkből választ kapunk [27].

5.4 A PROGRAM VÉDELMI MECHANIZMUSA



19. ábra Az MSN Logger védelmi mechanizmusa

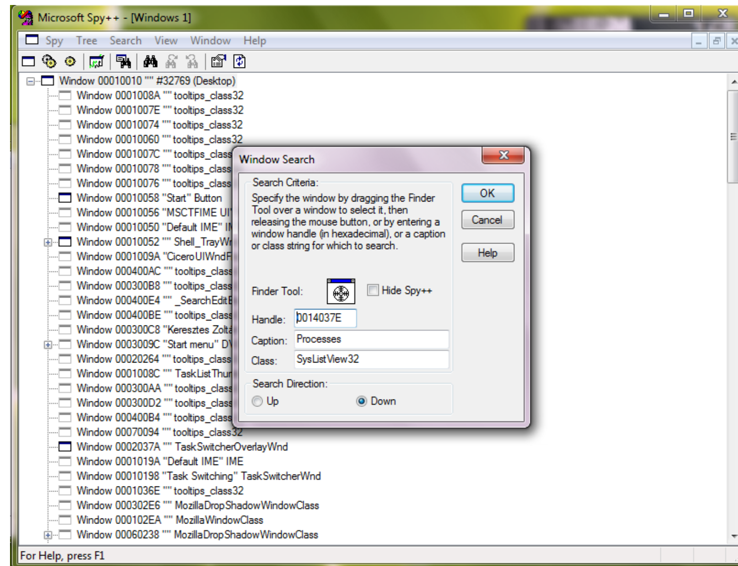
A program által alkalmazott védelem a már korábban említésre kerülő hardveres ujjlenyomat generálást alkalmazza. Amikor a felhasználó aktiválni szeretné a programot, akkor legenerálódik az általa használt számítógép hardveres ujjlenyomata. Ezt a felhasználó a program megvásárlása során megadja a weboldalon (ahol a program árulása történik) és ekkor birtokába jut a hardveres ujjlenyomat egy kódolt (privát kulccsal) változatához. Ez lesz a programaktiváló kulcsa. Ezt a programban megadva újabb kódolásra kerül a publikus

kulccsal. Ekkor, hogyha az eredeti hardveres ujjlenyomatot kapjuk vissza, akkor valóban a weboldal által lett megadva a kulcs.

Az aszimmetrikus kódolás során 2 kulccsal rendelkezünk. Egy privát kulccsal és egy publikus kulccsal. A publikus kulcsunk bárki számára hozzáférhető ez a programban lesz megtalálható. A privát kulcs kizárólag a vásárlást lebonyolító cég szerverén található meg. Jelenlegi matematikai ismereteink szerint egy megfelelő gondossággal kivitelezett RSA - titkosítás eredménye számításelméleti okok miatt nem fejthető vissza olyan gyorsan, hogy érdemes legyen megpróbálni. Azonban matematikailag nem bizonyított, hogy a titkosított adat visszafejtésére nem létezik kellő gyorsaságú algoritmus, ezért a jövőben ilyen algoritmus felfedezése lehetséges. Mivel az alkalmazás ilyen módon védett, így nincs lehetőségük sorozatszám generátor előállítására a forráskód ismeretében sem. Maximum a patching módszerrel élhetnek.

5.5 AZ ALKALMAZÁS ÁLTAL HASZNÁLT REJTŐZKÖDŐ FUNKCIÓ ISMERTETÉSE

Az alkalmazás a már korábban ismertetésre kerülő rendszer szintű hook-ot használja. Általában, ha a felhasználó tudni szeretné, hogy milyen folyamatok futnak, indulnak el a számítógépen, leggyakrabban a folyamatkezelőt, rendszerkonfigurációs segédprogramot (msconfig), rendszerleíróadatbázis szerkesztőt (regedit) használják erre a célra. A közös ezen segédprogramokban, az hogy az adatok megjelenítésére egy Windows-os syslistview32 komponenst használnak. Az, hogy egy felhasználói felület milyen osztályokból áll az egyszerűen használható Spy++ segédprogram segítségével könnyen kideríthető, amely a Visual Studio Professional eszközei között is megtalálható.



20. ábra: Spy++ program használata

Ha a célkeresztet rávisszük egy ablakra, kiírja az adott komponens leíróját, feliratát és a komponens osztályát.

Amikor a DLL-ben meghívjuk a ***SetWindowsHookEx* függvényt, akkor a DLL-ünk betöltődik minden grafikus felülettel rendelkező folyamat virtuális memóriájába és mivel a DLL belépési pontját is megírtuk, így lefut a** DLL_PROCESS_ATTACH rész. Létrehozunk egy új objektumot az adott folyamathoz, és ezen keresztül módosítjuk az ablakeljárásának a címét, amely a Windows- os üzenetek feldolgozását végzi.

A Windows Vista–ban sok folyamatot, segédprogramot csak megemelt biztonsági szinten lehetett futtatni, ehhez adminisztrátori módban kellett futtatni a programot. Ha adminisztrátori módban akarunk futtatni egy programot mindig, felugrik egy ablak, hogy megbízunk-e a program gyártójában. Ezt rengeteg felhasználó megelégelte, ezért kikapcsolta a UAC –t ezért a user módú rootkitek nagy többségben ugyanúgy hozzáférhettek a folyamatokhoz, mint a Vista előtti rendszerekben. A Windows 7-ben ezért módosították a UAC beállításokat. A Windows 7 fehér listán kezel alapvetően megbízhatónak tekintett folyamatokat. Ezen fehér listás folyamatoknak a rendszer alapértelmezett UAC beállítás mellett, rákérdezés nélkül engedélyezi magasabb privilégiumú kódok futtatását. A felhasználó nem módosíthatja, hogy mely folyamatok kerüljenek ezen listára, csak a Microsoft által meghatározott folyamatok, lehetnek ezen a listán. Biztonsági megerősítés nélkül azon folyamatok indulhatnak el, adminisztrátori módban, melyeknek a konfigurációs állományában szerepel ezen bejegyzés

```
<asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">
   <autoElevate>true</autoElevate>
</asmv3:windowsSettings>
```

21. ábra: autoElevate tulajdonság a manifest állományban

Ez a Microsoft által nem dokumentált jelzőbit ahhoz, hogy ezen jelzőbitet a rendszer kizárólag a Microsoft által aláírt folyamatoknál veszi figyelembe. A Windows 7 béta verziójában, a fehér listás folyamatok között (a jelenlegi verziókban már nem szerepel) szerepelt a RunDll32.exe ezt a folyamatot könnyedén rávehettük, hogy a paraméterként átadott DLL adminisztrátori módba kerüljön és ezután a rendszer folyamatain olyan kódot futtathattunk le amilyet csak akartunk. Léteznek ugyanis olyan folyamatok, amelyek ugyan nem kapnak automatikusan megemelt jogkört, de létre tudnak hozni bizonyos típusú emelt jogosultsági szintű COM objektumokat, melyeket aztán meg lehet kérni bizonyos feladatok végrehajtására. Egy ügyes programozó ezt kihasználva készített egy olyan példaalkalmazást, melynek felhasználásával a normál privilégiumú folyamat képes magasabb privilégiumot szintet elérni, a felhasználó beleegyezése nélkül, persze ez csak Windows 7 alatt működik. A forráskód és a részletes leírása a következő oldalon megtekinthető [21] . Mivel az általam készített rootkitnek hozzá kell férnie minden rendszer (Windows XP/Vista/7) alatt is egyaránt az összes folyamathoz (főleg a registry –hez és a msconfig segédprogramhoz), ezért más utat kellett választani

A futtatható állomány és a DLL tartalmazhat egy vagy több xml konfigurációs fájlt (mainfest). A fájl lehet beágyazva is, vagy az állomány mellett, melyben a konfigurációs beállításokat lehet tárolni. Az állomány egyaránt megtalálható a natív illetve a managelt exe-ékben DLL-ek ben. Az alábbi képen egy konfigurációs állomány látható.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="asInvoker" uiAccess="false"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

22. ábra: Példa egy manifest állományra

A programot vagy a DLL –t a rendszer az állomány betöltésekor megvizsgálja, ha talál manifest állományt, akkor a konfigurációt a szerint végzi el, ha nem talál, akkor az alapértelmezett beállítások szerint történik a betöltődés. A Windows Vista előtt a konfigurációs állományt arra használták, hogy megállapítsák például, hogy mi a DLL verziószáma. A Windows Vista óta a Microsoft kibővítette a konfigurációs állományt egy trustInfo résszel, amely valahogy így néz ki:

```xml
<requestedExecutionLevel
     level="asInvoker|highestAvailable|requireAdministrator"
     uiAccess="true|false"/>
```

23. ábra: Manifest állomány level, uiAccess tulajdonságainak lehetséges értékei

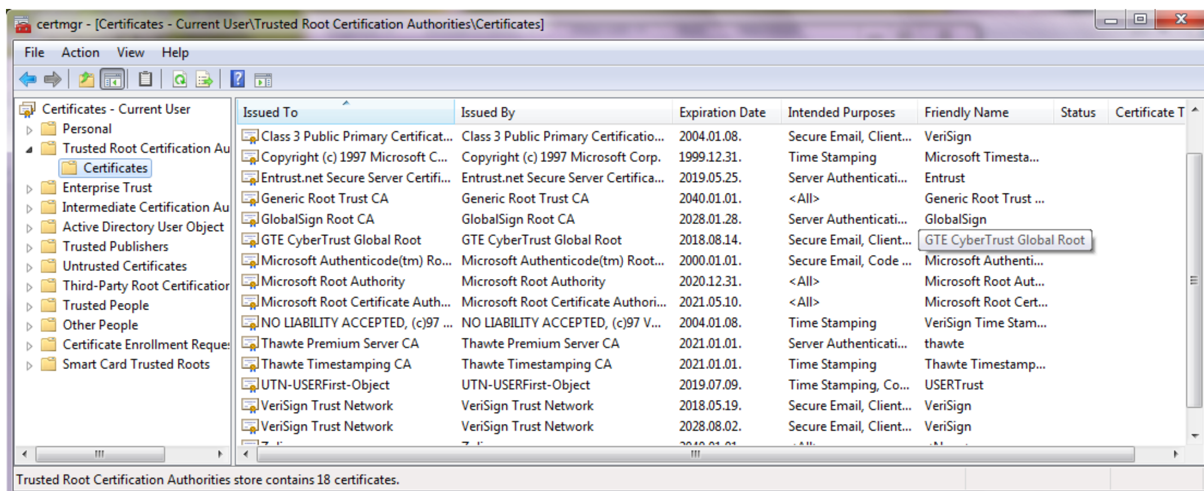
- **level**
  - asInvoker – Az alkalmazás ugyanolyan jogosultságokkal fut, mint a szülő folyamat.
  - highestAvailable – Az alkalmazás a legmagasabb jogosultággal fut, amit a felhasználó meg tud szerezni.
  - requireAdministrator – Az alkalmazás kizárólag adminisztrátori módban futhat.

- **uiAccess**
  - false -. Az alkalmazásnak nincs szüksége arra, hogy hozzáférjen más alkalmazások felhasználói felületéhez.
  - true – Az alkalmazás számára lehetővé teszi, hogy hozzáférjen magasabb privilégiumú szinten lévő folyamat felhasználói felületéhez (pl. üzenetküldés stb.).

Például, ha beállítjuk a manifest állományban a level–t requireAdministrator-re és ekkor elindítjuk az alkalmazást, már egyből igényli az adminisztrátori szintet a program nem kell külön adminisztrátori módban futtatni, de természetesen ettől még az UAC rákérdez arra, hogy megbízik-e az alkalmazás készítőjében. Ahhoz, hogy hozzáférjünk, magasabb privilégiumú szinten elhelyezkedő alkalmazásokhoz, nem elég, ha beállítjuk a manifest állományban az uiAccess –t true-ra. Az alkalmazásnak érvényes digitális aláírással kell rendelkeznie és az alkalmazásnak valamely rendszerkönyvtárban kell helyet foglalnia.

5.6 AZ ALKALMAZÁS DIGITÁLIS ALÁÍRÁSA

Az alkalmazásunk digitális aláírással kell rendelkeznie és az alkalmazásunkat aláírt hitelesítő szolgáltatónak (CA) szerepelnie kell a Windows gyökérhitelesítő szolgáltatói között. CA-ból még mindig elég sok lehet, **és mindegyikük** nyilvános kulcsának hiteles beszerzése még mindig gondot jelenthet. Ezért találták ki, hogy a *CA-k egymás nyilvános kulcsait is hitelesíthetik* – erre a célra szintén tanúsítványokat adnak ki. Ez egyben azt is jelenti, hogy ha ismerjük egy CA nyilvános kulcsát, akkor az alapján az általa hitelesített CA-két is le tudjuk ellenőrizni, ezekkel pedig azokét, akik számára ők bocsájtottak ki tanúsítványt, és így tovább. Ennek megfelelően *hitelesítési útvonalak* alakulhatnak ki közöttük. Vannak kitűntetett, „a hierarchia tetején álló" CA-k, ezeket hívjuk *Root-CA*-knak. Ezek már alapértelmezetten szerepelnek a rendszerben, ezen tanúsítványok önmagukat írják alá. Érdemes ilyen szolgáltatók által aláírni alkalmazásunkat, mivel így már alapértelmezetten megbízhatónak véli szoftverünket a rendszer. A rendszerben megtalálható tanúsítványokat a **start/run/certmgr.msc** paranccsal tudjuk megnézni.



24. ábra: Gyökér hitelesítő szolgáltatók tanúsítványai

Ha az alkalmazásunkat szeretnénk aláíratni, az elég sokba kerül. Vegyük például a VeriSign hitelesítési szolgáltatót. ennél a szolgáltatónál egy 2 évig hiteles aláírás 899 dollárba kerül jelen pillanatban, amely nem olcsó. Persze van egy másik járható út, amely egy fillérbe se kerül, de ehhez egy kis trükkre van szükség.

A .Net eszközei között van egy segédprogram, mellyel hitelesítő aláírást hozhatunk létre. A neve makecert.exe. Vegyük az alábbi parancsot. A további parancsokhoz a Visual Studio 2008 Command Promt-ot használom, mely induláskor beállítja a környezeti változókat mellyel bármelyik könyvtárból indíthatóak az általam használt segédprogramok

```
D:\>makecert -sv privateKey.pvk -r -n "CN=VeriSign Trust Network" -b 01/08/2003
-e 01/08/2018 CA.cer
```

25. ábra: Példa a makecert segédprogram használatára

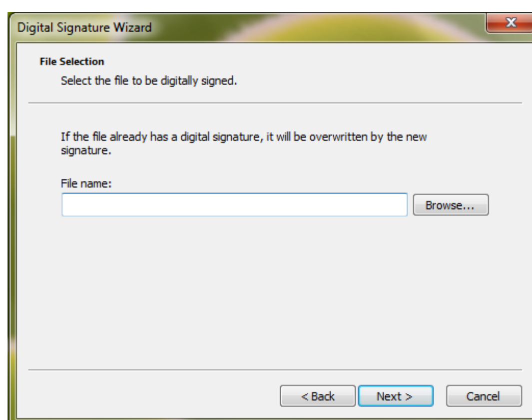
A makecert program paraméterei:

- -sv ha nem létezik az adott privát kulcs generál egyet
- -r önaláíró tanúsítványt hoz létre
- -n megadjuk a tanúsítvány nevét
- -b érvényesség kezdete
- -e érvényesség vége

Ha most generáltunk egy privát kulcsot meg kell adni a jelszót, amit használni szeretnénk a privát kulcsunknál. Ezután az assemblyinket kell megadni, melyet szeretnénk aláírni. Ehhez a

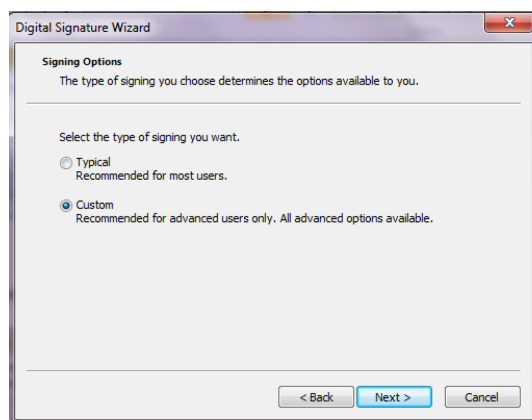
signtool nevű segédeszközt kell használni. A konfigurálás a legegyszerűbb elvégezni a –signwizart kapcsolóval mivel így egy grafikus felületet kapunk.

Első lépésben a fájl nevét kell megadni, melyet szeretnénk aláíratni.



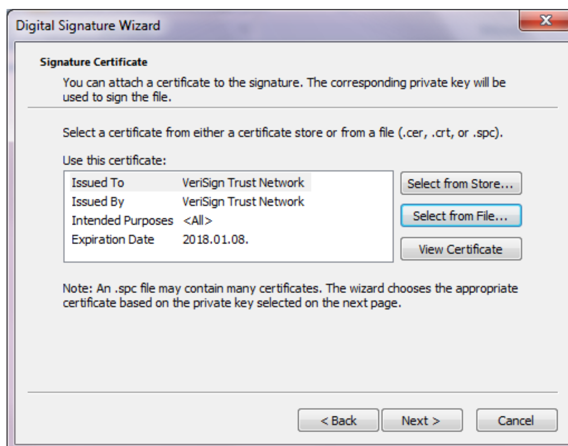
26. ábra: Tanúsítvány létrehozásának 1. lépése

Ezután válasszuk a Custom módot



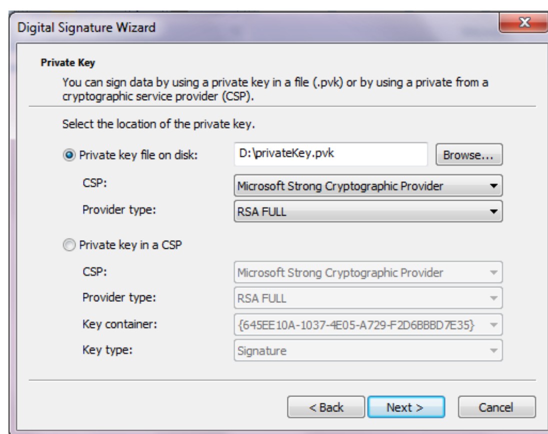
27. ábra: Tanúsítvány létrehozásának 2. lépése

Válasszuk ki a CA –t, amit az előbb létrehoztunk



28. ábra Tanúsítvány létrehozásának 3. lépése

Válasszuk ki a privát kulcsot, amit az előbb generáltunk

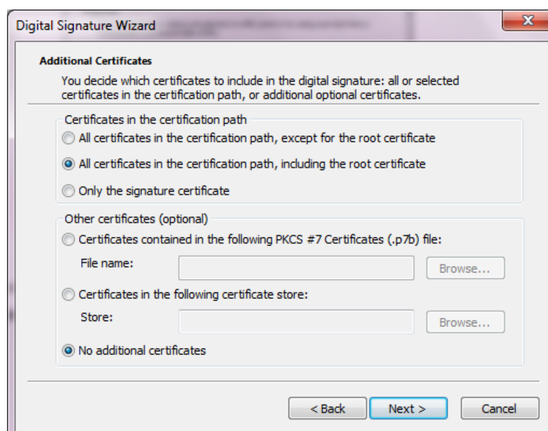


29. ábra Tanúsítvány létrehozásának 4. lépése

Ezután kérni fogja a privát kulcshoz tartozó jelszót, ezután meg kell adni a hash függvényt, amelyet használni fogunk. Ez lehet: md5, sha1. Ajánlott az sha1-et választani.

Végezetül meg kell adni, hogy melyik tanúsítvány legyen benne a digitális aláírásban.

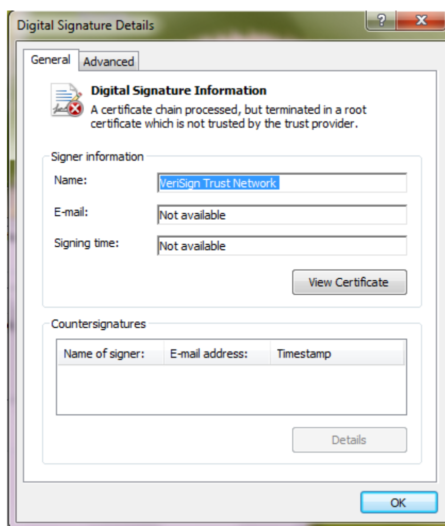
Ez maradhat alapértelmezetten:



30. ábra: Tanúsítvány létrehozásának 5. lépése

Ezután még van pár mező, amit opcionálisan kitölthetünk. Végezetül meg kell adnunk a privát kulcsunkhoz tartozó jelszót és el is készült a digitálisan aláírt alkalmazásunk.

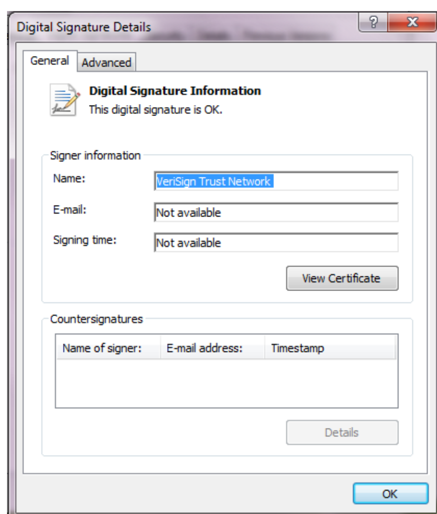
Ha ezután megnézzük az általunk aláírt alkalmazást vagy DLL –t, ezt tapasztaljuk:



31. ábra: Érvénytelen tanúsítvánnyal aláírt alkalmazás/ DLL

Ez azt jelenti, hogy a rendszer nem tudja ellenőrizni a tanúsítványunk hitelességét, mivel az aláíró nem szerepel a megbízható szolgáltatók között, de ezen könnyen lehet segíteni. A programhoz készül egy telepítő, amelyhez a felhasználónak szükséges engedélyezni a megemelt jogosultsági szintet. A telepítés közben a telepítő elindítja a hitelesítés injektáló alkalmazást, amely telepíti a rendszerbe a hitelesítő kulcsot. A technika előnye a felhasználó,

aki telepíti a számítógépre az alkalmazást, semmit nem kell pluszban csinálnia a telepítőkben szokásos NEXT->NEXT->NEXT -> OK mellett. Mégis ezután minden folyamathoz hozzá fog férni, a rootkit. A tanúsítvány telepítése után az alkalmazásunkat biztonságosnak minősíti a rendszer.



32. ábra: Érvényes tanúsítvánnyal rendelkező alkalmazás/ DLL

Ezek után már a rendszer összes folyamatához hozzáférhet az alkalmazás különösebb probléma nélkül.

6. KONKURENCIA ÉRTÉKELÉSE

Jelenleg a szoftverpiacon rengeteg kémprogram található, ha kizárólag csak az MSN Messenger naplózására specializált szoftvert nézzük, akkor is létezik 1-2 hasonló alkalmazás. Ha az általam készített program ki szeretne tűnni a tömegből, akkor a szem előtt tartandó legfontosabb funkciók a rejtőzködés és az igényesség lehet az, amivel kitűnhet. Úgy érzem, hogy ezt olyan szinten sikerült megvalósítanom, amellyel a konkurencia nem rendelkezik. Általánosságban elmondható, hogy a kémprogramok nagy többsége nagyon amatőr szinten rejtőzködik a rendszerben. Nem 1 bites userek könnyedén kiszúrhatják akár a folyamatok között is a kémprogramokat. Nagyon sok program egyszerűen csak úgy álcázza magát, hogy valamely rendszerfolyamat nevét leutánozzák, és ha a felhasználó rákeres erre a neten, akkor azt látja, hogy ez egy rendszerfolyamat, de a trükk ott bukik el, hogy a rendszerfolyamatok tulajdonosa a system nem a felhasználó. Vegyük például az MSN SPY Monitor programot. Ez az egyetlen igazán komoly versenytársa az általam készített programnak. Az én

programommal ellentétben a naplózást nem a programhoz történő kapcsolódással valósítja meg, hanem mivel az MSN programban történő beszélgetések során a számítógépet az adatforgalom titkosítatlanul hagyja el, így ezt monitorozva visszanyerhetőek a beszélgetések. A kimenő, bejövő forgalom monitorozásához a WinPcap library-t használja az alkalmazás. A felhasználói felülete egész kellemesnek mondható.

33. ábra: MSN SPY Monitor működés közben[2]

A neten úgy került meghirdetésre, mint amely program teljesen kompatibilis a legújabb operációs rendszerekkel is. Ezzel szemben nekem telepítés után azonnal kiakadt Windows 7 operációs rendszer alatt. Véleményem szerinte a program fejlesztője kizárólag kikapcsolt UAC mellett tesztelte a programot. Természetesen szinte, mint minden kémkedésre szakosodott program, itt is fel van sorolva a funkcióismertetőnél, hogy a rendszerben teljesen láthatatlan. Ezzel szemben, nagyon is látható, semmilyen különleges programot nem igényel a felfedezése. Összesen 2 folyamatot indít el, melyek egymást figyelik, ha az egyik leállításra kerülne, akkor azt a másik folyamat azonnal újra elindítja. Egyszerre pedig a felhasználó nehezen tud kilőni 2 folyamatot. Ha az általános kémprogramokat nézem, akkor sem találtam olyat, amely valóban eltűntette volna nyomait a felhasználó szeme elől. Egyes programok különböző service-ket regisztráltak a rendszerben és azon keresztül naplózták a felhasználó tevékenységeit. Az XP –s feladatkezelőben még nem mutatta a futó serviceke-t a feladatkezelő, ezért használták régebben ezt a módszert, de mára már ez se járható út.

[2] Letölthető: http://www.ematrixsoft.com/download.php?file=ms

7. Összefoglalás

Az általam megírásra került program jelenleg az MSN Messengerben történő beszélgetések naplózására képes, akkor is, ha az MSN Messengerben a naplózás ki van kapcsolva. A program teljesen láthatatlan, tapasztaltabb felhasználók is csak speciális programok használatával szerezhetnek tudomást a naplózó program futásáról. Rejtőzködés alatt azt értem, hogy a program nem látszik a feladatkezelőben, telepített programok között, az operációs rendszeren automatikusan elinduló programok listáján se szerepel, a registry-ben sem találhatók meg az MSN Logger-re (a program neve) utaló bejegyzések. A rejtőzködés alapjául egy saját kezűleg írt rootkit szolgál, mely egyaránt működik Windows XP/Vista/7 operációs rendszerek alatt is. Az általam készített rootkit bármi mást is el tudna rejteni az operációs rendszerből, ezért felhasználási lehetőségeinek csak a képzelet szab határt, jó és rossz célokra egyaránt felhasználható. A programra tekinthetünk úgy, mint egy keretrendszerre, melynek a későbbi bővítése minimális befektetést igényel. Az alkalmazás megtervezés során a tesztelhetőséget és a későbbi bővíthetőség lehetőségét tartottam szem előtt. Ehhez a jól ismert tervezési mintákat alkalmaztam. Az alkalmazás a későbbiekben a következő funkciókkal fog kiegészülni: billentyűzetnaplózás, file és könyvtár figyelés, aktív ablak monitorozás(milyen alkalmazást használt a felhasználó), képlopó funkció. A felsorolt funkciók megvalósításához szükséges tudás birtokában vagyok, most már csak a kivitelezés van hátra. A szoftver licenc mechanizmusának alapjául a számítógép hardverinformációiból képzett hardveres ujjlenyomat és a RSA algoritmus szolgál. A szoftver védelmi módszerének kidolgozása után megismerkedtem a szoftverfeltörési technikákkal is, mert mint tudjuk a crackerek a legjobb védelmi módszerek tudói. Jövőbeni terveim között szerepel a rootkitek minél mélyebb tanulmányozása (user módú és kernel módú egyaránt). Szándékaim között szerepel még az MS Detours library megismerése, mely módot nyújt arra, hogy az általánosan használt módszereknél ellentétben sokkal könnyebben képezhessünk hook-ot a rendszerben és a jövőbeni terveim között szerepel még a driver írás is, mellyel teljes mértékben az uralmam alá hajthatom az operációs rendszert.

8. Irodalomjegyzék

[1] http://msdn.microsoft.com/en-us/magazine/cc163610.aspx#S1

[2] http://people.inf.elte.hu/z_farkas/OS/Index_DLLs.html

[3] http://www.codeproject.com/KB/threads/APIHooking.aspx

[4] http://www.codeproject.com/KB/threads/winspy.aspx

[5] http://www.codeproject.com/KB/system/Hack_Windows_Task_Manager.aspx

[6] http://www.codexonline.hu/CodeX8/alap/cpp/MagyarAttila/KeyBoard%20Hook.htm

[7] http://zoell.hu/2009/08/22/c-rendszer-menu-manipulalasa/

[8] http://www.microsoft.com/hun/technet/archive/?type=author&id=d3574925-3169-42af-8667-7bb9bbf880d1

[9] http://www.flounder.com/hooks.htm

[10] http://www.rohitab.com/discuss/index.php?s=61bc3b1c83264a2cb4e6b89bdb90edd8&showtopic=34305

[11] http://www.codeproject.com/KB/system/hooksys.aspx

[12] http://www.codeproject.com/KB/DLL/hooks.aspx

[13] http://www.softwareonline.hu/Article/View.aspx?id=3903

[14] http://www.dnjonline.com/article.aspx?id=jun07_access3264

[15] http://www.tar.hu/ozikorn/data/borland/fejezetek/f_6_1_s.html

[16] http://www.microsoft.com/whdc/driver/install/AppInit-Win7.mspx

[17] http://www.withinwindows.com/2009/02/04/windows-7-auto-elevation-mistake-lets-malware-elevate-freely-easily/

[18] http://www.pretentiousname.com/misc/win7_uac_whitelist2.html

[19] http://www.codeproject.com/KB/dotnet/insidedontnet.aspx

[20] http://www.symantec.com/connect/articles/windows-anti-debug-reference

[21] http://www.codeproject.com/KB/system/inject2exe.aspx

[22] http://udooz.net/blog/2009/03/ngen-helpful-or-harmful/

[23] http://docs.google.com/viewer?a=v&q=cache:oQuJRltNxYUJ:www.flexerasoftware.com/webdocuments/PDF/is_Digital_Signing_and_Security.pdf+installshield+create+certificate&hl=hu&gl=hu&pid=bl&srcid=ADGEESi5bKE0Nkokez2EaiNiUv8Lf09HBrGdASPhHiyDJEiJZ3CK6X4hUq-

6_NkEoVB6z2B3uiUqG1kCJdAfQQ9aVXu456iLiDa8dcF1S5KTGmlURMBdAWW
Mwl0l3p0b25GhH1FZSACK&sig=AHIEtbRGuEa6rDMzQuM7Buu66d85plBpZA

[24] http://www.codeproject.com/KB/cs/testnonpublicmembers.aspx

[25] http://www.blackwasp.co.uk/ElevatedButton.aspx

[26] http://chrison.net/UACElevationInManagedCodeGuidanceForImplementingCOMElevation.aspx

[27] http://sohotechnology.wordpress.com/2009/08/23/3/

[28] http://comsci.liu.edu/~murali/win32gui/CallBackFun.htm

[29] http://portal.b-at-s.info/download.php?list.2 (a listában szereplő összes dokumentumot elolvastam kb. 60 db)

[30] http://tuts4you.com/

[31] http://ntcore.com/files/rebelnet.htm

[32] http://codebetter.com/blogs/jeremy.miller/archive/2005/10/06/132825.aspx

[33] http://springtips.blogspot.com/2007/06/configuration-hell-remedy-with.html

[34] http://www.dofactory.com/Framework/Framework.aspx (Design pattern framework tanulmányozása)

[35] http://buildingsecurecode.blogspot.com/2007/06/how-to-display-uac-shield-icon-inside.html

[36] http://www.restuner.com/howto-insert-trust-info-manifest.htm

[37] http://martinfowler.com/articles/mocksArentStubs.html

[38] http://www.codeproject.com/KB/cs/weakeventhandlerfactory.aspx

[39] http://www.codeproject.com/KB/cs/WeakEvents.aspx

[40] http://msdn.microsoft.com/en-us/library/ms756482%28VS.90%29.aspx (felhasznált library COM objektum megemelt jogosultsági szintjének igényléséhez)

[41] http://www.icsharpcode.net/OpenSource/SharpZipLib/ (a program által használt log fájlok tömörítéséhez használt library)

[42] Greg Hoglund, Jamie Butler : *Rootkits: Subverting the Windows Kernel*

[43] Trey Nesh: C# 2008

[44] Christian Nagel, Bill Evjen, Jay Glynn, Morgan Skinner, Karli Watson: *Professional C# 2008*