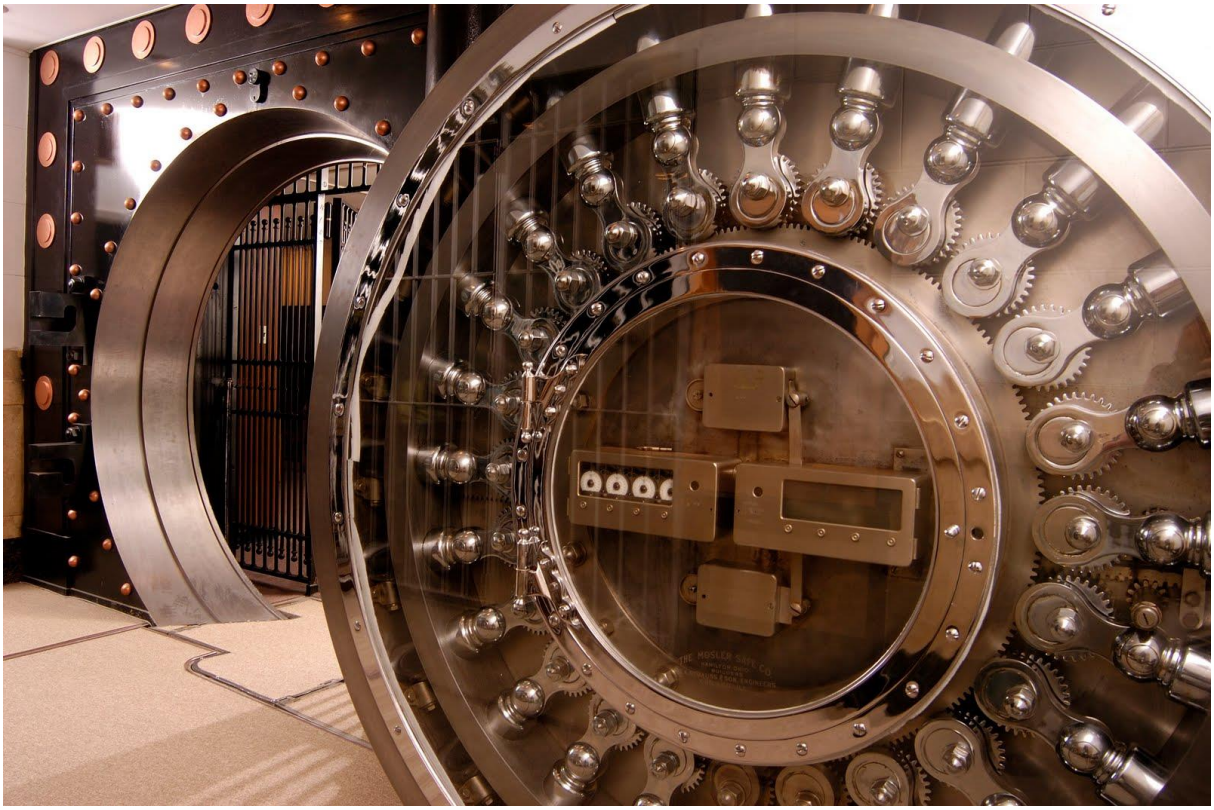
Jak utrudnić życie crackerowi. Zabezpieczenia antypirackie dla programistów.

Autor *Bartosz Wójcik* / Opublikowane *Wrzesień 2002* / Magazyn *Software 2.0*



Metody ochrony oprogramowania przed złamaniem, w tym zabezpieczenia antypirackie i triki antydebug pozwalające na wykrywanie popularnych narzędzi crackerskich.

Zastanawialiście się kiedyś, dlaczego w internecie można znaleźć cracki do prawie każdego programu, jaki tylko pojawi się na rynku? Może sami jesteście programistami i już doświadczyliście tego na własnej skórze? Jeśli tak jest, lub jeśli po prostu chcecie wiedzieć jak się przed tym ochronić, zachęcam do dalszej lektury.

Skąd się biorą cracki?

Publikowane w internecie cracki, to zazwyczaj małe programy, które modyfikują aplikację, sprawiając, że przykładowo *„znika”* ograniczenie czasowe, co do użytkowania programu, lub dzięki zastosowaniu cracka można zarejestrować aplikację na dowolne dane.

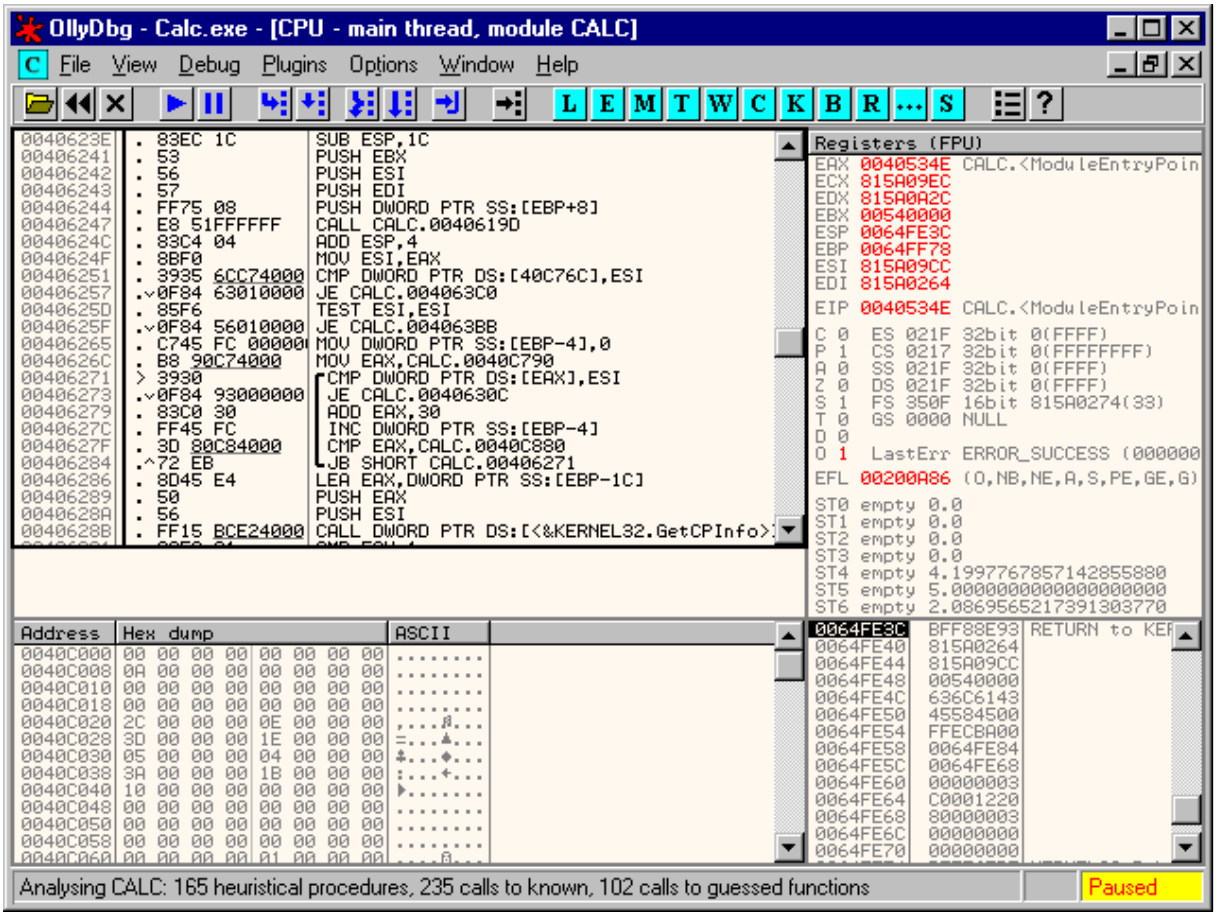
Skąd biorą się cracki czy np. generatory numerów seryjnych? Są one tworem crackerów, czyli ludzi, którzy zajmują się usuwaniem zabezpieczeń z programów komputerowych, co odróżnia ich od hakerów, którzy przełamują zabezpieczenia w systemach sieciowych.

Debugger

Aby cracker mógł przełamać zabezpieczenie danego programu, najczęściej musi zlokalizować punkt w kodzie programu, gdzie wywoływane są np. procedury weryfikujące poprawność wprowadzonych danych rejestracyjnych.

Do tego celu najczęściej wykorzystywanym narzędziem jest *debugger*, czyli program pozwalający śledzić wykonywanie kodu programu na poziomie instrukcji asemblera.

Cracker śledząc kod programu, jest w stanie przechwycić (stosując tzw. pułapki – z ang. breakpoint) wywoływanie funkcji systemowych, odpowiedzialnych przykładowo za odczytanie numeru seryjnego z okna edycyjnego programu. Następnie śledząc dalsze wykonywanie programu, cracker może zlokalizować miejsce, gdzie występuje sprawdzenie wprowadzonego numeru seryjnego z oryginalnym, dzięki czemu może uzyskać poprawny numer seryjny.



Okno debuggera OllyDbg

Jak się bronić przed złamaniem zabezpieczeń?

Mówi się, że każde zabezpieczenie można przełamać i to tylko kwestia czasu. Skoro tak, to dobrym pomysłem jest, aby tak utrudnić życie crackera, żeby musiał poświecić dużo więcej czasu, na jego złamanie, co z kolei może spowodować, że nie będzie w stanie złamać naszego programu i po prostu zrezygnuje z niego, na rzecz innego, słabiej zabezpieczonego programu.

Najczęściej stosowaną metodą pozwalającą utrudnić analizę programu, są tzw. *triki antydebug*. Polega to na tym, że zanim uruchomimy nasz program, sprawdzamy obecność narzędzi wykorzystywanych przez crackera i w przypadku ich wykrycia, możemy przykładowo zawiesić wykonywanie programu. Można zastosować także bardziej wyrafinowane metody, przykładowo, w razie wykrycia debuggera, pozwalamy programowi na normalne działanie, ale np. ustawiamy *timer*, który spowoduje, że program zawiesi się, lub wyłączy po pięciu minutach działania.

Tego typu metody są bardzo skuteczne, gdyż wymagają dogłębnej analizy programu, w celu wykrycia nieprawidłowości. Najgorszym rozwiązaniem jest

wyświetlanie informacji, że np. został wykryty debugger, to jeden z najczęściej popełnianych błędów, cracker, który zna treść tej wiadomości z łatwością może znaleźć kod w programie, gdzie jest ona wywoływana i usunąć kod odpowiedzialny za sprawdzanie obecności debuggera.

Należy nadmienić także, że metody *antydebug* można podzielić na dwie kategorie, te, które korzystają z udokumentowanych funkcji systemu Windows, oraz wykorzystujące luki w systemie operacyjnym. W tym artykule opiszę metody udokumentowane, gdyż gwarantują one poprawne działanie programu, oraz nie ograniczają ich zastosowań w różnych wersjach systemu Windows.

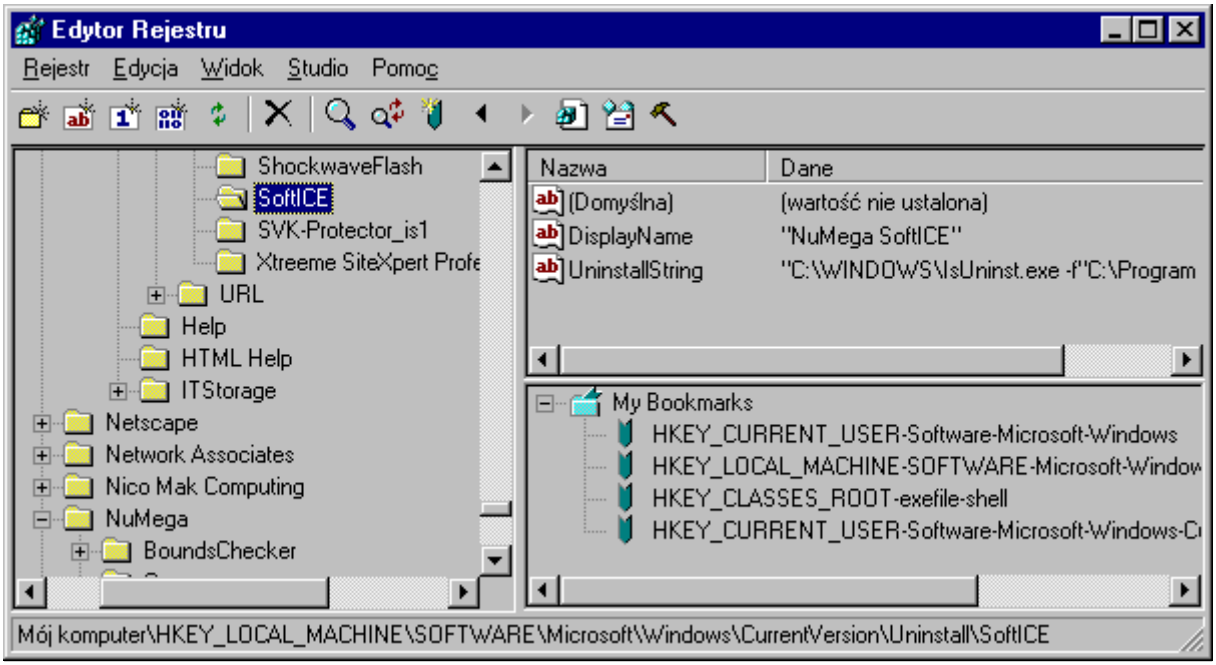
Wykrywanie debuggera

Obecnie najpopularniejszym debuggerem jest *SoftIce* firmy Numega, oprócz tego istnieje cała masa innych narzędzi jak *TRW*, *OllyDbg* itd. Program debuggera zazwyczaj działa na wyższym poziomie uprzywilejowania (debuggery systemowe) niż normalne programy, w takim wypadku program debuggera jest dostarczony w formie sterownika systemowego. System Windows umożliwia komunikację normalnych programów ze sterownikami, wykorzystując funkcję *DeviceIoControl()*, ale zanim będzie można komunikować się ze sterownikiem, należy do niego uzyskać dostęp. Robi się to poprzez wywołanie funkcji WinApi *CreateFile()* z nazwą sterownika, podaną w specjalnej postaci „\\.*STEROWNIK*”. Dzięki tej funkcji możemy wykryć sterownik debuggera, przykładowy kod:

```
////////////////////////////////////////////////////////////////////////////
// HANDLE CheckDriver(char *lpszDriver)
//
// funkcja próbuje otworzyć dostęp do podanego sterownika systemowego
//
// na wyjściu:
// uchwyt sterownika lub INVALID_HANDLE_VALUE w przypadku braku
// sterownika
//
////////////////////////////////////////////////////////////////////////////

HANDLE CheckDriver(char *lpszDriver)
{
  return (CreateFile(lpszDriver, GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL));
}


////////////////////////////////////////////////////////////////////////////
//
// BOOL IsSice()
//
// funkcja sprawdza obecność debuggera SoftIce w wersji dla systemów
// Windows 9x
//
// na wyjściu:
// jeśli debugger SoftIce jest aktywny, funkcja zwraca TRUE,
// w przeciwnym wypadku FALSE
//
////////////////////////////////////////////////////////////////////////////
```

```
BOOL IsSice()
{
  return (CheckDriver("\\\\.\\SICE") != INVALID_HANDLE_VALUE ? TRUE : FALSE);
}


//////////////////////////////////////////////////////////////////////////////
//
// BOOL IsNtice()
//
// funkcja sprawdza obecność debuggera SoftIce w wersji dla
// systemów Windows NT/2K/XP
//
// na wyjściu:
// jeśli debugger SoftIce jest aktywny, funkcja zwraca TRUE,
// w przeciwnym wypadku FALSE
//
//////////////////////////////////////////////////////////////////////////////

BOOL IsNtice()
{
  return (CheckDriver("\\\\.\\NTICE") != INVALID_HANDLE_VALUE ? TRUE : FALSE);
}

//////////////////////////////////////////////////////////////////////////////
//
// BOOL IsSiwdebug()
//
// funkcja sprawdza obecność dodatkowego sterownika zainstalowanego
// wraz z debuggerem SoftIce dla systemów Windows 9x, obecność tego
// sterownika w systemie, świadczy o obecności debuggera
//
// na wyjściu:
// jeśli debugger SoftIce jest aktywny, funkcja zwraca TRUE,
// w przeciwnym wypadku FALSE
//
//////////////////////////////////////////////////////////////////////////////

BOOL IsSiwdebug()
{
  // otwórz pomocniczy sterownik
  CheckDriver("\\\\.\\SIWDEBUG");

  // sprawdź ostatni błąd, jeśli błąd był inny niż ten świadczący, że brak
  // jest sterownika, oznacza to, że debugger jest aktywny
  return ( GetLastError() != ERROR_FILE_NOT_FOUND ? TRUE : FALSE);
}
```

Wykrywanie debuggera SoftIce poprzez sprawdzanie jego sterownika, obecnie nie jest zbyt skuteczną metodą, gdyż crackerzy modyfikują swoje narzędzia pracy, tak aby nie była możliwa ich detekcja. Nieco inną metodą pozwalającą stwierdzić, że w systemie zainstalowany jest debugger, jest sprawdzanie kluczy rejestru Windows, na wstępowanie tam wpisów, jakie pozostawił np. program instalacyjny debuggera, lub kluczy zawierających ustawienia konfiguracyjne debuggera.



Wpisy debuggera w rejestrze Windows

Metoda ta jednak ma pewną wadę, nie pozwala na 100% stwierdzić, że debugger jest aktywny, bo wpisy w rejestrze mogły pozostać po poprzedniej instalacji. Przykładowy kod:

```
////////////////////////////////////////////////////////////////////////////
// BOOL IsReg()
//
// funkcja sprawdza obecność gałęzi rejestru Windows
//
// na wyjściu:
// jeśli w rejestrze Windows znajduje się podana gałąź, funkcja
// zwróci TRUE w przeciwnym wypadku FALSE
//
////////////////////////////////////////////////////////////////////////////

BOOL IsReg(char *lpszReg)
{
  PHKEY phkResult;
  DWORD iResult;

  // otwórz gałąź rejestru Windows w HKEY_LOCAL_MACHINE
  iResult = RegOpenKeyEx(HKEY_LOCAL_MACHINE, lpszReg, 0, KEY_ALL_ACCESS,
&phkResult);

  // zamknij uchwyt
  RegCloseKey(phkResult);

  return (iResult == ERROR_SUCCESS ? TRUE : FALSE);
}


////////////////////////////////////////////////////////////////////////////
// BOOL IsSiceReg1()
//
// funkcja sprawdza obecność gałęzi rejestru wykorzystywanych przez
// debugger SoftIce do trzymania informacji potrzebnych przy
// deinstalacji debuggera
//
```

```
// na wyjściu:
// jeśli w rejestrze Windows znajduje się gałąź debuggera, funkcja zwróci
// TRUE w przeciwnym wypadku FALSE
//
////////////////////////////////////////////////////////////////////////////

BOOL IsSiceReg1()
{
  return IsReg("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall\\SoftICE");
}


////////////////////////////////////////////////////////////////////////////
//
// BOOL IsSiceReg2()
//
// funkcja sprawdza obecność gałęzi rejestru wykorzystywanych przez
// debugger SoftIce do trzymania informacji konfiguracyjnych
//
// na wyjściu:
// jeśli w rejestrze Windows znajduje się gałąź debuggera, funkcja zwróci
// TRUE w przeciwnym wypadku FALSE
//
////////////////////////////////////////////////////////////////////////////

BOOL IsSiceReg2()
{
  return IsReg("SOFTWARE\\NuMega\\SoftICE");
}
```

Wykrywanie debuggera można także zrealizować analizując wpisy w pliku *„autoexec.bat”*, gdzie znajduje się ścieżka do programu debuggera uruchamianego podczas startu systemu.

```
SET SOUND=C:\PROGRA~1\CREATIVE\CTSND
SET MIDI=SYNTH:1 MAP:E MODE:0
SET BLASTER=A220 I5 D1 H5 P330 E620 T6
SET Path=C:\WINDOWS;C:\WINDOWS\COMMAND;E:\DEV\MASM\;E:\DEV\MASM\BIN;E:\DEV\TASM;
mode con codepage prepare=((852) C:\WINDOWS\COMMAND\ega.cpi)
mode con codepage select=852
keyb pl,,C:\WINDOWS\COMMAND\keybrd4.sys
C:\PROGRA~1\Numega\Softice\winice.exe
```

Ostatnia linijka służy do ładowania debbugera. Sprawdzając zawartość pliku *„autoexec.bat”* możemy stwierdzić obecność debuggera:

```
////////////////////////////////////////////////////////////////////////////
//
// BOOL IsAutoexec(char *lpString)
//
// funkcja szuka w pliku autoexec.bat podanych wpisów
//
// na wyjściu:
// jeśli znaleziono podany wpis, funkcja zwróci TRUE
//
////////////////////////////////////////////////////////////////////////////

BOOL IsAutoexec(char *lpString)
{
  BOOL bResult = FALSE;
  int iSize, i;
  char *lpMap, *lpFi, lpFile[256];
  HANDLE hFile, hMap;
```

```
  // odczytaj ścieżkę katalogu Windows
  GetWindowsDirectory(lpFile,256);

  // zbuduj ścieżkę do pliku autoexec.bat
  strcpy(&lpFile[3], "autoexec.bat");


  // otwórz plik
  hFile = CreateFile(lpFile, GENERIC_READ, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);

  if (hFile != INVALID_HANDLE_VALUE)
  {
    // pobierz rozmiar pliku
    iSize = GetFileSize(hFile, NULL);

    // jeśli plik jest pusty wyjdź
    if (iSize)
    {
      // utwórz mapę pliku
      hMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);

      if (hMap)
      {
        // mapuj plik do pamięci
        lpMap = MapViewOfFile(hMap, FILE_MAP_READ, 0, 0, 0);

          if (lpMap != 0)
          {
            lpFi = lpMap;

            // skanuj zawartość pliku autoexec.bat
            while (iSize--)
            {
              if strnicmp(lpFi++, lpString, strlen(lpString)) == 0)
              {
                bResult = TRUE;
                break;
              }
            }
          }

          UnmapViewOfFile(lpMap);
        }

        CloseHandle(hMap);
      }

    CloseHandle(hFile);
  }

  return bResult;
}
```

Następnie w programie należy sprawdzić, czy podane wpisy są obecne w pliku:

```
if (IsAutoexec("numega") == TRUE || IsAutoexec("softice") == TRUE)
{
  ExitProcess(-1);
}
```

Wykrywanie pułapek debuggera

Na początku artykułu wspomniałem o pułapkach debuggera, crakerzy stosują je, aby przykładowo przechwycić wywołanie funkcji odczytującej numer seryjny z okna edycyjnego. Pułapki debuggera działają na tej zasadzie, iż pierwszy bajt procedury, na którą zastawiona jest pułapka, jest nadpisywany specjalną instrukcją asemblera *„int 3”*. Gdy debugger natrafi na instrukcję *„int 3”*, zatrzymuje działanie programu i przekazuje sterowanie w ręce crackera. Reprezentacja heksadecymalna tej instrukcji to wartość 0xCC, aby wykryć aktywną pułapkę wystarczy sprawdzić, czy pierwszy bajt procedury jest równy właśnie 0xCC.

Przykładowo, jeśli w procedurze rejestracyjnej wykorzystujemy funkcję WinApi *GetDlgItemText()* można przed jej wywołaniem sprawdzić, czy na jej adresie cracker zastawił pułapkę i w razie jej wykrycia, można zakończyć działanie aplikacji.

```
////////////////////////////////////////////////////////////////////////////
//
// BOOL IsBpx(char *lpszModule,char *lpszFunction)
//
// funkcja sprawdza, czy na adresie podanej funkcji zastawiona
// jest pułapka debuggera
//
// na wyjściu:
// jeśli pułapka jest zastawiona, funkcja zwraca TRUE
//
////////////////////////////////////////////////////////////////////////////

BOOL IsBpx(char *lpszModule,char *lpszFunction)
{
  HINSTANCE hLibrary;
  DWORD lpProc;

  // sprawdź czy biblioteka jest już załadowana
  hLibrary = GetModuleHandle(lpszModule);

  // załaduj bibliotekę
  if (hLibrary == NULL)
  {
    hLibrary = LoadLibrary(lpszModule);
  }

  if (hLibrary != NULL)
  {
    // pobierz adres procedury
    lpProc = (DWORD)GetProcAddress(hLibrary, lpszFunction);

    if (lpProc != 0)
    {
      // sprawdź, czy zastawiona jest pułapka
      if (*(BYTE *)lpProc == 0xCC) return TRUE;
    }
  }

  return FALSE;
}

// w procedurze rejestracyjnej sprawdzamy, czy została założona pułapka
if (IsBpx("USER32.dll","GetDlgItemTextA") == TRUE)
{
  ExitProcess(-1);
```

```
}
```

Jeśli np. nasze zabezpieczenie opiera się na pliku kluczu, zamiast sprawdzać funkcję *GetDlgItemTextA*, należy sprawdzić funkcję, którą wykorzystujemy do otwarcia pliku klucza i w razie wykrycia pułapki, można wprowadzić program w nieskończoną pętle np.:

```
// powtarzaj wykonywanie kodu, dopóki zastawiona pułapka
while(IsBpx("KERNEL32.dll", "CreateFileA") == TRUE);
```

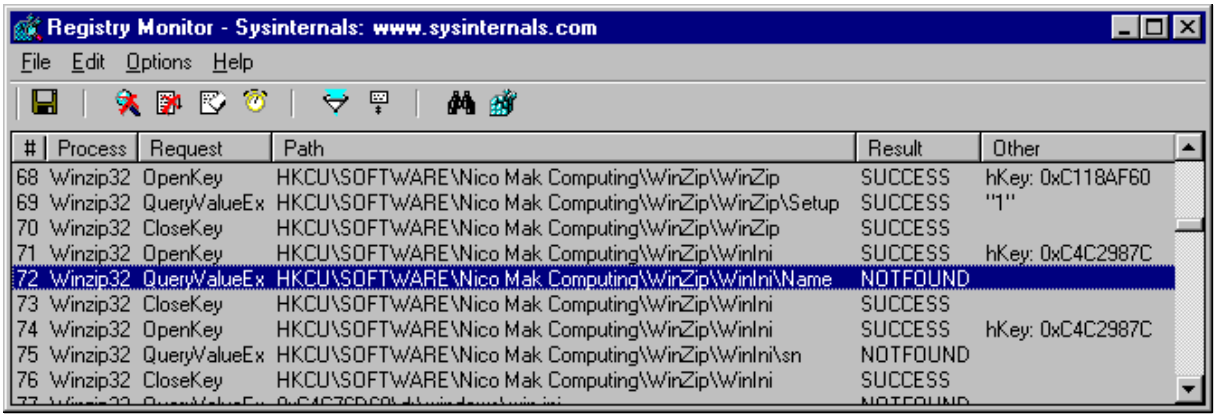
Korzystając z funkcji *IsBpx* można także wykryć aktywnego debuggera SoftIce w systemach Windows NT, 2K i XP. Debugger SoftIce pod tymi systemami, domyślnie zakłada pułapkę (na własne potrzeby) na funkcję WinApi *SetUnhandledExceptionFilter()*, wykrycie pułapki świadczy o obecności debuggera w systemie.

```
// sprawdź obecność debuggera SoftIce NT
if (IsBpx("KERNEL32.dll", "SetUnhandledExceptionFilter") == TRUE)
{
  ExitProcess(-1);
}
```

Monitory systemowe

Crackerzy oprócz debuggerów często używają wielu pomocniczych narzędzi, do tych bardziej popularnych można zaliczyć monitory systemowe. Są to programy, które pozwalają zobaczyć, jaki program np. próbował odczytać klucze z rejestru Windows, lub pozwala zanalizować, jakie pliki próbował otworzyć.

Analizując dane z monitorów można z łatwością zauważyć, że przykładowo aplikacja próbowała otworzyć klucz w rejestrze Windows, gdzie powinny być zapisane dane rejestracyjne. Obecnie najpopularniejszymi narzędziami tego typu są monitory *Filemon* i *Regmon* firmy Sysinternals.



Monitor rejestru w akcji

Działanie monitorów po części opiera się także na wykorzystaniu sterowników i do ich wykrycia, można skutecznie użyć metod, które skutkowały w wykrywaniu sterowników debuggera.

```
//////////////////////////////////////////////////////////////////////////////
//
// BOOL IsFilemon()
//
// funkcja sprawdza obecność sterownika wykorzystanego w programie
// Filemon, służącym do śledzenia odwołań do plików
//
// na wyjściu:
// jeśli monitor plików jest aktywny, funkcja zwraca TRUE,
// w przeciwnym wypadku FALSE
//
//////////////////////////////////////////////////////////////////////////////

BOOL IsFilemon()
{
  return ( CheckDriver("\\\\.\\FILEVXD") != INVALID_HANDLE_VALUE ? TRUE : FALSE);
}

//////////////////////////////////////////////////////////////////////////////
//
// BOOL IsRegmon()
//
// funkcja sprawdza obecność sterownika wykorzystanego w programie
// Regmon, służącym do śledzenia odwołań do rejestru Windows
//
// na wyjściu:
// jeśli monitor rejestru jest aktywny, funkcja zwraca TRUE,
// w przeciwnym wypadku FALSE
//
//////////////////////////////////////////////////////////////////////////////

BOOL IsRegmon()
{
  return ( CheckDriver("\\\\.\\REGVXD") != INVALID_HANDLE_VALUE ? TRUE : FALSE);
}
```

Programy Filemon i Regmon posiadają okienkowy interfejs użytkownika, dzięki nazwie głównego okna możliwe jest wykrycie, czy program monitora jest aktywny lub nie. Do tego celu wykorzystuje się funkcję WinApi *FindWindowEx()*.

```
//////////////////////////////////////////////////////////////////////////////
//
// BOOL IsFilemonWnd()
//
// funkcja sprawdza, czy obecnie otwarte jest okno Filemon'a
//
// na wyjściu:
// jeśli monitor plików jest aktywny, funkcja zwraca TRUE,
// w przeciwnym wypadku FALSE
//
//////////////////////////////////////////////////////////////////////////////

BOOL IsFilemonWnd()
{
return ( FindWindowEx(NULL, NULL, NULL, "File Monitor - Sysinternals:
www.sysinternals.com") != NULL ? TRUE:FALSE);
}
```

```
////////////////////////////////////////////////////////////////////////////
//
// BOOL IsRegmonWnd()
//
// funkcja sprawdza, czy obecnie otwarte jest okno Regmon'a
//
// na wyjściu:
// jeśli monitor rejestru jest aktywny, funkcja zwraca TRUE,
// w przeciwnym wypadku FALSE
//
////////////////////////////////////////////////////////////////////////////

BOOL IsRegmonWnd()
{
  return ( FindWindowEx(NULL, NULL, NULL, "Registry Monitor - Sysinternals:
www.sysinternals.com") != NULL ? TRUE : FALSE);
}
```

Dzięki *FindWindowEx* możemy z łatwością sprawdzić, czy uruchomione są monitory i podjąć odpowiednie kroki, np. wyłączyć naszą aplikacje, ale możliwe jest także wyłączenie samego monitora. Proszę zauważyć, że funkcja *FindWindowEx()* w przypadku znalezienia okna, zwraca jego uchwyt. Mając uchwyt okna, można wysłać do niego komunikat WM_CLOSE, który wysyłany jest, gdy użytkownik zamyka okno aplikacji. Wysłanie tego komunikatu spowoduje wyłączenie monitora.

```
////////////////////////////////////////////////////////////////////////////
//
// BOOL KillRegmonWnd()
//
// funkcja w razie wykrycia okna Regmona zamyka je
//
// na wyjściu:
// jeśli monitor rejestru był aktywny, funkcja zwraca TRUE,
// w przeciwnym wypadku FALSE
//
////////////////////////////////////////////////////////////////////////////

BOOL KillRegmonWnd()
{
  HANDLE hWindow;

  // znajdź okno Regmona
  hWindow = FindWindowEx(NULL, NULL, NULL, "Registry Monitor - Sysinternals:
www.sysinternals.com");

  // w razie znalezienia okna
  if (hWindow != NULL)
  {
    PostMessage(hWindow, WM_CLOSE, 0, 0);
  }

  return ( hWindow != 0? TRUE : FALSE);
}
```

Może się zdarzyć, że aplikacja nie będzie reagowała na komunikat WM_CLOSE, można również spróbować wysłać wiadomość WM_ENDSESSION, która symuluje zamykanie systemu Windows. Jeśli jednak i to nie pomoże, warto zastosować nieco

ostrzejsze metody. Mając uchwyt okna, otwieramy proces, do którego należy okno i zamykamy go wykorzystując funkcje *TerminateProcess()*.

```
////////////////////////////////////////////////////////////////////////////
//
// BOOL KillFilemonWnd()
//
// funkcja w razie wykrycia okna Filemona zamyka je
//
// na wyjściu:
// jeśli monitor Filemon był aktywny i został zamknięty, funkcja
// zwraca TRUE, w przeciwnym wypadku FALSE
//
////////////////////////////////////////////////////////////////////////////

BOOL KillFilemonWnd()
{
  HANDLE hWindow,hProcess;
  DWORD dwPid;

  // znajdź okno Filemona
  hWindow = FindWindowEx(NULL, NULL, NULL, "File Monitor - Sysinternals:
www.sysinternals.com");

  // w razie znalezienia okna
  if (hWindow != NULL)
  {
    // pobierz identyfikator procesu, do którego należy okno
    if (!GetWindowThreadProcessId(hWindow, &dwPid))
    {
      // otwórz proces z flagami pozwalającymi wyłączyć go
      hProcess = OpenProcess(PROCESS_TERMINATE, FALSE, dwPid);

      if (hProcess != NULL)
      {
        // zakończ proces
        if ( TerminateProcess(hProcess,-1) != 0) return TRUE;
      }
    }
  }

  return FALSE;
}
```

Zabezpieczanie pliku aplikacji

Nasz program, uzbrojony w metody antydebug może już sprawiać pewne kłopoty, ale nadal istnieje możliwość, że cracker złamie go i opublikuje w internecie cracka.

Najlepszym sposobem, aby tego uniknąć jest dodatkowe zabezpieczenie pliku. Zabezpieczenie opiera się na tej zasadzie, iż plik wykonywalny naszej aplikacji, przed jej publikacją, dodatkowo jest szyfrowany i kompresowany.

Po takim zabezpieczeniu pliku, jakakolwiek jego modyfikacja, np. spowodowana użyciem cracka, powoduje, że plik nie może być uruchomiony. Do zabezpieczania plików wykorzystuje się specjalne programy tzw. *exe-protektory*.

Po zabezpieczeniu pliku wykonywalnego, do jego struktury dodawany jest specjalny kod, który odpowiedzialny jest m.in. za deszyfrowanie i dekompresję danych, a następnie uruchomienie aplikacji.
Oprócz tego, kod dodawany do zabezpieczanych plików zawiera szereg metod antydebug, dzięki którym nie jest możliwe uruchomienie aplikacji w obecności debuggerów lub innych narzędzi wykorzystywanych przez crackera.

Zakończenie

Przedstawione metody, to tylko skrawek wszystkich możliwości, które pozwolą ochronić własny program przed złamaniem. Mam nadzieję, że mój artykuł pozwolił Wam dostrzec, że publikowane oprogramowanie, wcale nie jest tak bezpieczne jakby się to wydawało na pierwszy rzut oka, ale także chciałem pokazać, że w jakiś sposób można się bronić.

Adresy w internecie

- http://www.pelock.com – protektor i system kluczy licencyjnych PELock
- http://www.sysinternals.com – monitory Filemon i Regmon
- http://www.numega.com – strona producenta debuggera SoftIce
- http://www.knlsoft.com – debugger TRW
- http://home.t-online.de/home/Ollydbg – darmowy debugger OllyDbg
- http://upx.sourceforge.net – kompresor plików wykonywalnych UPX
- http://www.collakesoftware.com – kompresor plików wykonywalnych PECompact

O autorze

Bartosz Wójcik – autor protektora plików wykonywalnych PELock bazującego na systemie kluczy licencyjnych.