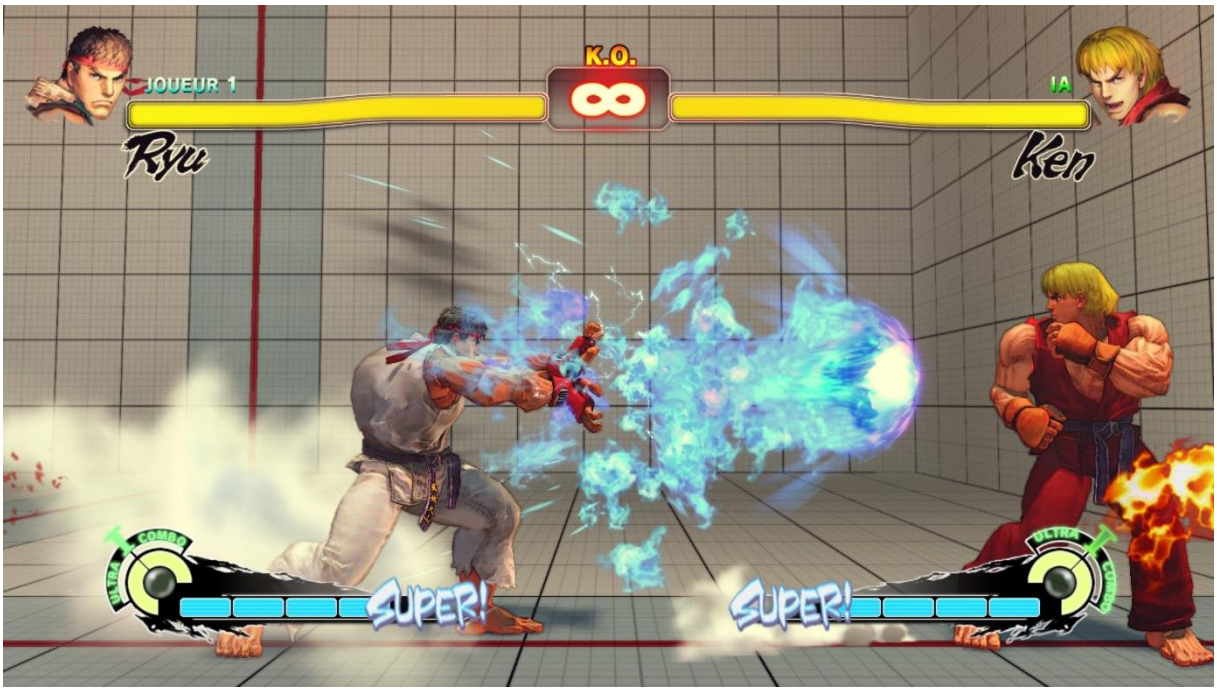
Anti reverse engineering. Wirusy kontra antywirusy

Autor Bartosz Wójcik | Opublikowane Maj 2013 | Magazyn Programista 5/2013 (12)



Anti reverse engineering, czyli techniki utrudniające oraz spowalniające inżynierię wsteczną złośliwego oprogramowania (malware).

Inżynierią wsteczną (ang. *reverse engineering*) określa się techniki wykorzystywane do analizy skompilowanego oprogramowania, bez dostępu do jego kodów źródłowych. W tym artykule chciałbym przedstawić metody, jakie wykorzystywane są przez twórców złośliwego oprogramowania do utrudnienia analizy wirusów komputerowych i złośliwego oprogramowania (tzw. *malware*), a także sposoby, jak sobie z tym radzą firmy antywirusowe i same antywirusy.

Aby utrudnić analizę firmom antywirusowym, trzeba na początku zrozumieć, w jaki sposób złośliwe oprogramowanie jest analizowane przez firmy antywirusowe. Jak zatem w dużym uproszczeniu wygląda analiza typowego malware?

- Testowanie złośliwego oprogramowania w wirtualnym środowisku

- Testowanie w piaskownicy (ang. *sandbox*) oraz emulatorach
- Monitorowanie zmian dokonywanych w systemie
- Statyczna analiza
- Dynamiczna analiza
- Generowanie sygnatur złośliwego oprogramowania

Oprogramowanie antywirusowe może natomiast weryfikować następujące elementy oraz wykorzystywać je wszystkie do ostatecznej oceny faktycznych zamiarów testowanego oprogramowania:

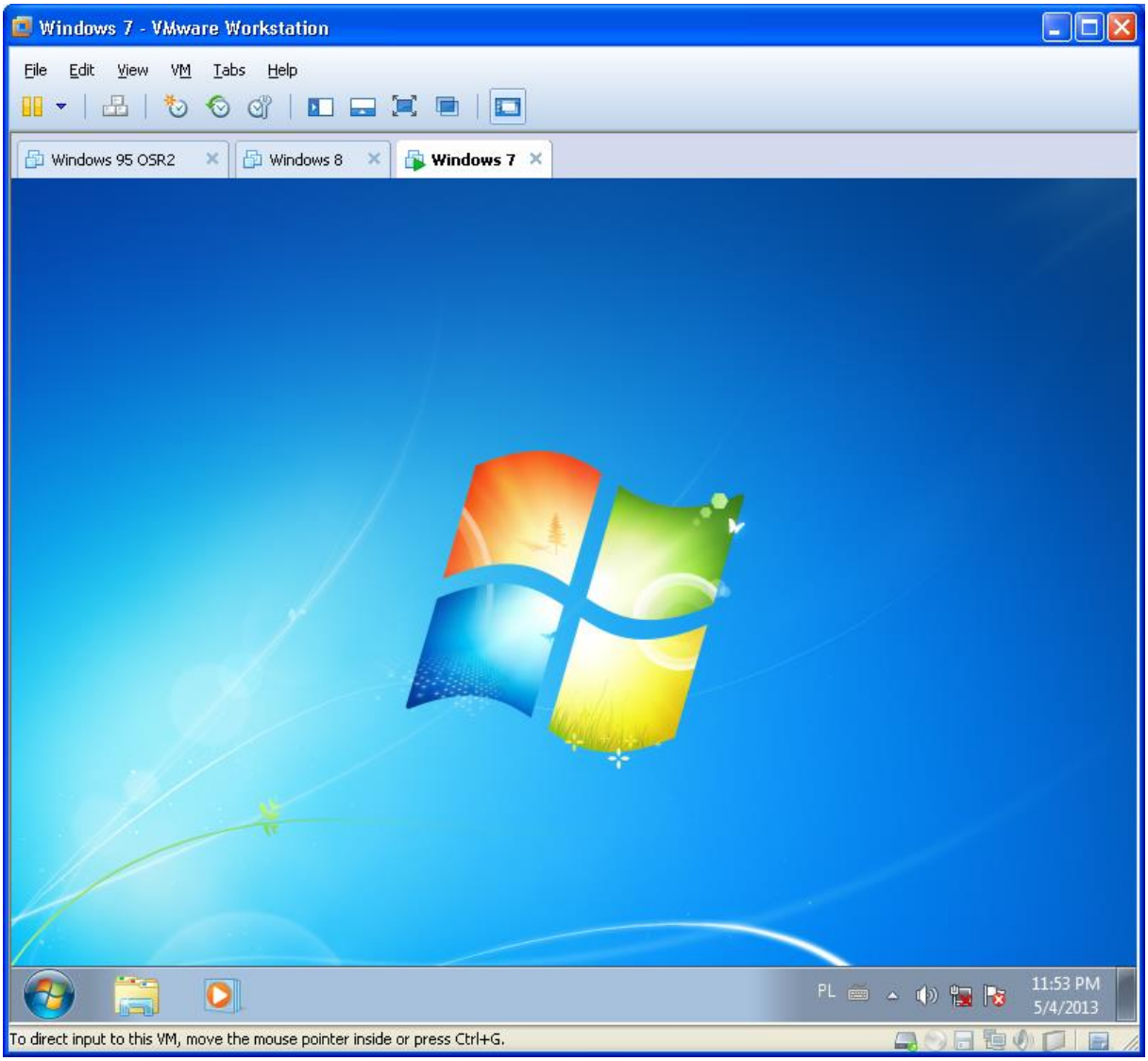
- Sumy kontrolne plików i ich fragmentów (np. *MD5*, *SHA1*, *SHA2*, *CRC32*)
- Nietypowe struktury i wartości w plikach
- Sygnatury fragmentów plików (metody heurystyczne)
- Stałe ciągi tekstowe
- Zachowanie aplikacji, tzw. analiza behawioralna (monitorowanie dostępu do systemu plików, Rejestru Windows etc.)
- Wywoływane funkcje (specyficzne funkcje, ich parametry, kolejność)

Na każdym z tych kroków, zarówno przy analizie przez firmy antywirusowe, jak i same antywirusy można napotkać na specjalnie zaprojektowane przeszkody w celu uniemożliwienia analizy lub jej spowolnienia.

Wykrywanie wirtualnych maszyn

Złośliwe oprogramowanie w 99% wypadków testowane jest na wirtualnych maszynach, takich jak np. *VMware*, *VirtualBox*, *Virtual PC, Parallels* etc. Ma to na celu zabezpieczenie analityków przed zainfekowaniem ich własnych maszyn, co czasami też się zdarza, jak np. w przypadku firmy antywirusowej *ESET*, gdzie przez błąd pracownika w 2011 roku, z komputerów analityka został wykradziony pakiet drogiego i znanego oprogramowania analizującego *IDA* wraz z dekompilatorem *HexRays*. Normalnie w viruslabie złośliwe oprogramowanie trzyma się także w katalogach bez prawa do wykonywania, tak żeby właśnie ustrzec się przed przypadkowym uruchomieniem zarażonego lub złośliwego oprogramowania. Stosowanie wirtualnych maszyn umożliwia także wykorzystanie dodatkowych narzędzi, np. pozwala na łatwe porównanie obrazu systemu przed infekcją oraz po infekcji w celu wykrycia najdrobniejszych zmian, jakich dokonuje złośliwe oprogramowanie w systemie plików, Rejestrze Windows i innych składnikach systemu i komputera.

Rysunek 1. Środowisko wirtualne VMware



Uruchamianie na wirtualnych maszynach pozwala również na dokładne śledzenie ruchu sieciowego (np. korzystając ze *snifferów* sieciowych jak *Wireshark*), jeśli malware kontaktuje się przykładowo z serwerami kontrolującymi działania sieci botnet, do której należy analizowany przykład.

Wirtualne maszyny nie są idealnym odwzorowaniem prawdziwego komputera i posiadają cechy, które wykorzystywane są do ich wykrywania, np.:

- Komponenty sprzętowe o specyficznych nazwach tylko dla wirtualnych maszyn
- Niekompletne lub ograniczone emulowanie struktur prawdziwych maszyn (tabele *IDT*, *GDT*)
- Ukryte interfejsy API do komunikacji z *wierzchnią warstwą* (np. *Virtual PC* wykorzystuje instrukcję assemblera `cmpxchg8b eax` z *magicznym* kluczem w rejestrach procesora do stwierdzenia swojej obecności)

- Narzędzia pomocnicze, np. *VMware Tools*, które można wykryć np. po nazwach obiektów systemowych (*mutex*, *event*, nazwy klas, nazwy okienek)

Listing 1. Wykrywanie środowiska VMware poprzez interfejs API.

```
BOOL IsVMware()

{

  BOOL bDetected = FALSE;


  __try

  {

    // wykryj obecność VMware

    __asm

    {

      mov     ecx,0Ah

      mov     eax,'VMXh'

      mov     dx,'VX'

      in      eax,dx

      cmp     ebx,'VMXh'

      sete    al

      movzx   eax,al

      mov     bDetected,eax

    }

  }

  __except(EXCEPTION_EXECUTE_HANDLER)

  {

    // w razie wystąpienia wyjątku

    // zwróć FALSE

    return FALSE;

  }


  return bDetected;

}
```

Listing 2. Wykrywanie środowisk wirtualnych przez wpisy w Rejestrze Windows.

```
BOOL IsVM()

{

  HKEY hKey;

  int i;

  char szBuffer[64];

  char *szProducts[] = { "*VMWARE*", "*VBOX*", "*VIRTUAL*" };


  DWORD dwSize = sizeof(szBuffer) - 1;


  if (RegOpenKeyEx(HKEY_LOCAL_MACHINE,
"SYSTEM\\ControlSet001\\Services\\Disk\\Enum", 0, KEY_READ, &hKey) ==
ERROR_SUCCESS)

  {

    if (RegQueryValueEx(hKey, "0", NULL, NULL, (unsigned char *)szBuffer, &dwSize )
== ERROR_SUCCESS)

    {

      for(i = 0; i < _countof(szProducts); i++)

      {

        if (strstr(szBuffer, szProduct[i]))

        {

          RegCloseKey(hKey);

          return TRUE;

        }

      }

    }


    RegCloseKey(hKey);

  }


  return FALSE;

}
```

Listing 3. Wykrywanie *Virtual PC*.

```
    sub     eax,eax          ; przygotuj magiczne
    sub     edx,edx          ; wartości w rejestrach
    sub     ebx,ebx
    sub     ecx,ecx
    db      0Fh, 0C7h, 0C8h ; instrukcja cmpxchg8b eax
; jeśli Virtual PC jest obecny, po wykonaniu cmpxchg8b eax
; rejestr EAX = 1 i nie nastąpi wyjątek
; w przeciwnym wypadku nastąpi wyjątek
```

Listing 4. Wykrywanie uruchomienia w środowisku *VirtualBox*.

```
BOOL IsVirtualBox()
{
  BOOL bDetected = FALSE;

  // czy w systemie zainstalowana jest
  // biblioteka pomocnicza VirtualBox?
  if (LoadLibrary("VBoxHook.dll") != NULL)
  {
    bDetected = TRUE;
  }
  // czy w systemie zainstalowany jest
  // sterownik narzędzi pomocniczych
  // środowiska VirtualBox?
  if (CreateFile("\\\\.\\VBoxMiniRdrDN", GENERIC_READ, \
                 FILE_SHARE_READ, NULL, OPEN_EXISTING, \
                 FILE_ATTRIBUTE_NORMAL, NULL) \
                 != INVALID_HANDLE_VALUE)
  {
    bDetected = TRUE;
  }

  return bDetected;
```

```
}
```

Co ciekawe, sama obecność funkcji wykrywających wirtualne maszyny może dla analityka być sygnałem o tym, że ma do czynienia ze złośliwym oprogramowaniem, które próbuje uniknąć analizy w bezpiecznym środowisku, jednak takie funkcje są powszechnie wykorzystywane w systemach ochrony oprogramowania i treści, np. odtwarzacz popularnej platformy multimedialnej *Ipla.tv* również nie pozwala na oglądanie programów na żywo, jeśli stwierdzone zostanie uruchomienie w wirtualnym środowisku.

Niektórzy autorzy oprogramowania również blokują możliwość uruchamiania aplikacji w wirtualnych środowiskach z prozaicznych powodów – piractwo. Oprogramowanie zainstalowane na wirtualnej maszynie, przypisane do komponentów sprzętowych wirtualnej maszyny (np. gdy klucz licencyjny przypisany jest do identyfikatora sprzętowego maszyny), w normalnych warunkach byłoby możliwe do uruchomienia tylko na jednym komputerze. W przypadku wirtualnej maszyny, możliwe jest proste skopiowanie obrazu wirtualnej maszyny i uruchomienie jej na dowolnym komputerze.

Piaskownice

Piaskownica, czyli z angielskiego *sandbox* , to, mówiąc w skrócie, *skrzynka*, bezpieczne środowisko, do którego wrzuca się złośliwe oprogramowanie i monitoruje jego zachowanie. Piaskownice mogą mieć formę osobnych systemów, najbardziej znane to:

- Cuckoo Sandbox
- Anubis Sandbox
- Norman Sandbox
- JoeBox
- Vipre ThreatAnalyzer
- Buster Sandbox Analyzer

Są to wirtualne środowiska, które pozwalają na uruchomienie dowolnego oprogramowania oraz dzięki wbudowanym narzędziom monitorującym udostępniają szczegółowe logi zmian, jakich dokonało oprogramowanie po uruchomieniu w systemie. Zwykle zbudowane są w oparciu o emulowane środowisko Windows i posiadają charakterystyczne cechy umożliwiające ich proste wykrycie. Na forach undergroundowych można znaleźć wiele przykładów wykrywania takich środowisk, np.:

Listing 5. Wykrywanie uruchomienia w środowisku *Norman Sandbox*.

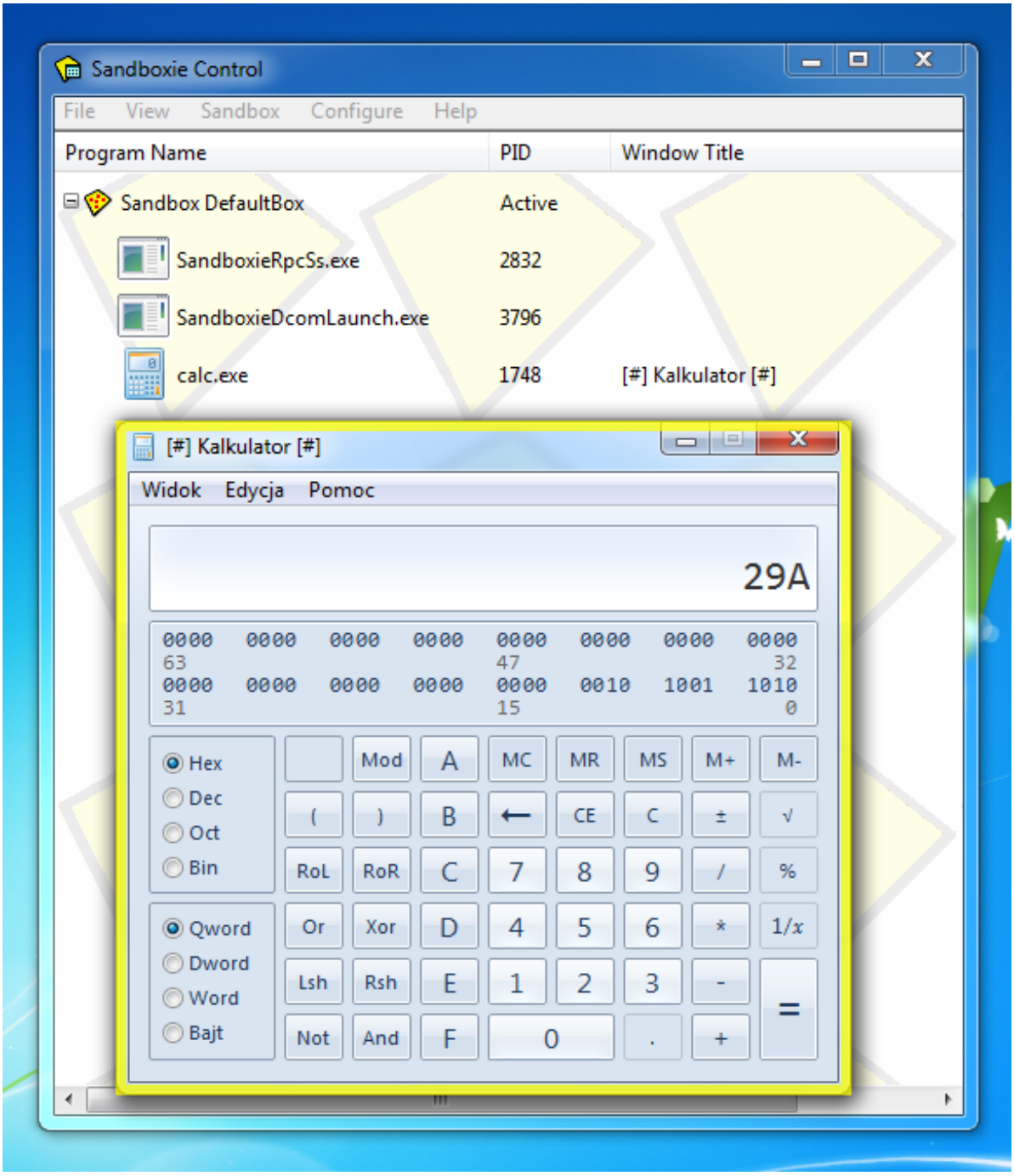
```
; pobierz adres procedury DecodePointer(), która
; w środowisku Norman Sandbox posiada charakterystyczny
; kod, niespotykany w prawdziwych systemach Windows
; DecodePointer:
;.7C80644E| C8 00 00 00  enter   0,0
;.7C806452: 8B 45 08     mov     eax,[ebp][8]
;.7C806455: 0F C8        bswap   eax
;.7C806457: C9           leave
;.7C806458: C2 04 00     retn    4

; pobierz adres procedury DecodePointer
    mov     eax,DecodePointer
    test    eax,eax
    je      _nie_wykryto

; weryfikuj sygnaturę bajtów
    cmp     dword ptr[eax],000000C8h
    jne     _nie_wykryto

    cmp     dword ptr[eax+4],0F08458Bh
    jne     _nie_wykryto

    cmp     dword ptr[eax+8],04C2C9C8h
    jne     _nie_wykryto

; wykryto obecność środowiska Norman Sandbox
; zakończ działanie aplikacji
    push    0
    call    ExitProcess

; kontynuuj działanie
```

```
_nie_wykryto:
```

Większość z wymienionych środowisk była przez lata darmowa, jednak obecnie część została skomercjalizowana i tylko niektóre pozostały darmowe, jak np. *Anubis Sandbox*.

Oprócz całych zintegrowanych systemów, jak omawiany powyżej, coraz popularniejsze stają się systemy wbudowane w oprogramowanie antywirusowe oraz takie jak np. *Sandboxie*, pozwalające na bezpieczne wykonywanie dowolnych aplikacji w ramach systemu operacyjnego, które chronią wrażliwe komponenty systemowe przed modyfikacją.

Rysunek 2. Kalkulator uruchomiony w środowisku Sandboxie



System *Sandboxie* nie zapisuje modyfikacji na dysku ani do plików systemowych, tylko do zewnętrznych plików, które można później przeanalizować. Działanie takich systemów opiera się na globalnych *hookach* systemowych (modyfikacjach niskopoziomowych funkcji systemu operacyjnego) oraz na wykorzystaniu dodatkowych sterowników kontrolujących zachowanie aplikacji. Jak można się domyśleć, te mechanizmy i komponenty są wykorzystywane przez twórców złośliwego oprogramowania do wykrywania tych narzędzi.

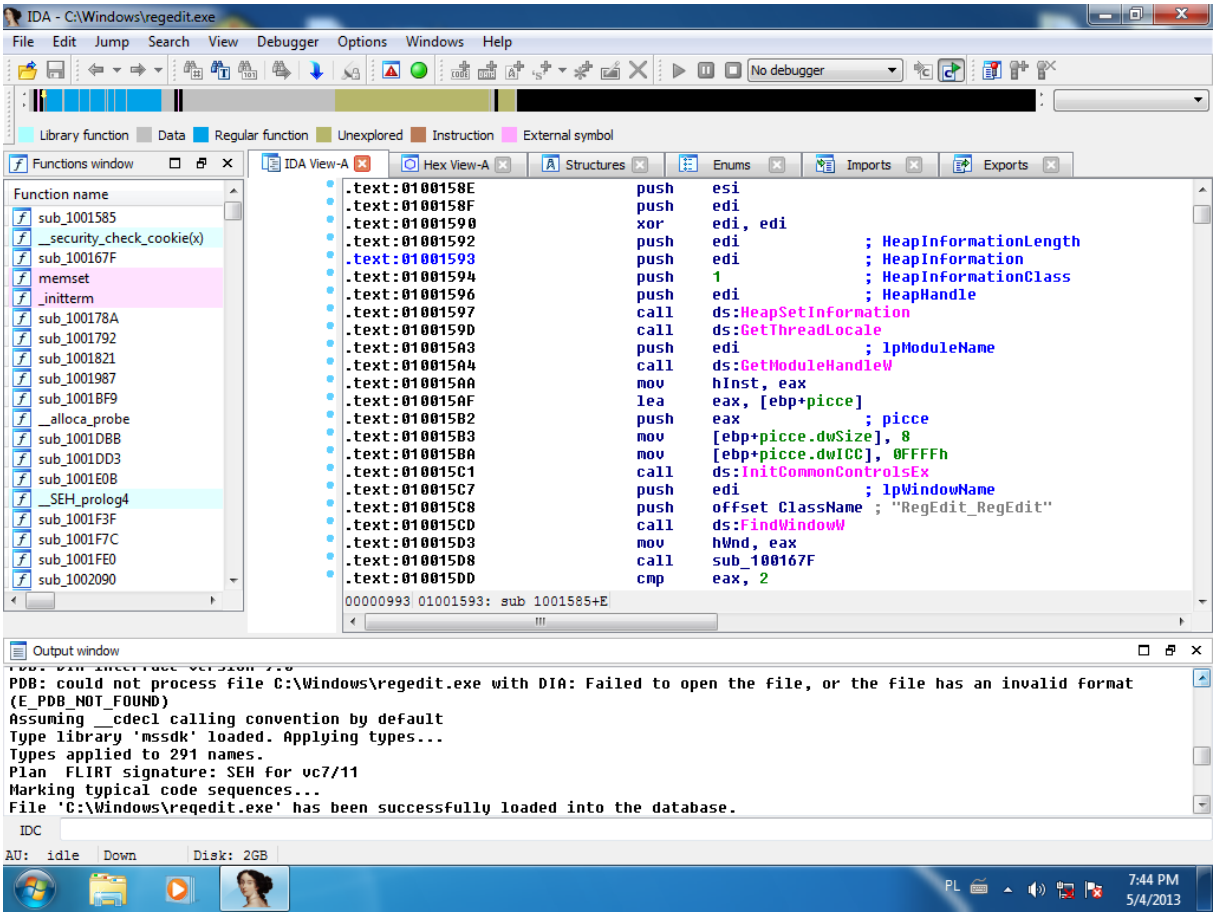
Listing 6. Wykrywanie uruchomienia w środowisku *Sandboxie*.

```
BOOL IsSandboxie()

{

  // sprawdź czy w naszym procesie jest

  // załadowana biblioteka pomocnicza

  // Sandboxie

  if (GetModuleHandle("SbieDll.dll") != NULL)

  {

    return TRUE;

  }


  return FALSE;

}
```

Utrudnienie analizy kodu

Statyczna analiza kodu polega na wykorzystaniu narzędzi takich jak deasembler *IDA* i dekompilator *HexRays* (jest to de facto standard w firmach antywirusowych) do analizy plików podejrzanych programów. Narzędzia te pozwalają na analizę skompilowanych aplikacji na poziomie assemblera oraz jeśli to możliwe, na poziomie *HLL*.

Rysunek 3. Deasembler IDA



Aby można było wykonać analizę statyczną, należy mieć dostęp do niezaszyfrowanych plików oprogramowania i tutaj do akcji wkraczają narzędzia, które utrudniają lub wręcz uniemożliwiają dokonanie takiej analizy bez dodatkowej pracy.

Cryptery

Cryptery, *cryptory*, *kryptory* etc. to najbardziej prymitywne narzędzia wykorzystywane do szyfrowania całych plików wykonywalnych. Tworzone są tylko w jednym celu – aby zaszyfrowany plik złośliwego oprogramowania uniknął detekcji przez programy antywirusowe. Sprzedawane są często na forach undergroundowych za kwoty rzędu kilkudziesięciu dolarów i każdemu klientowi dostarczana jest unikalna kopia, aby uniknąć wykrycia poprzez stare sygnatury w programach antywirusowych.

Zasada ich działania wygląda następująco:

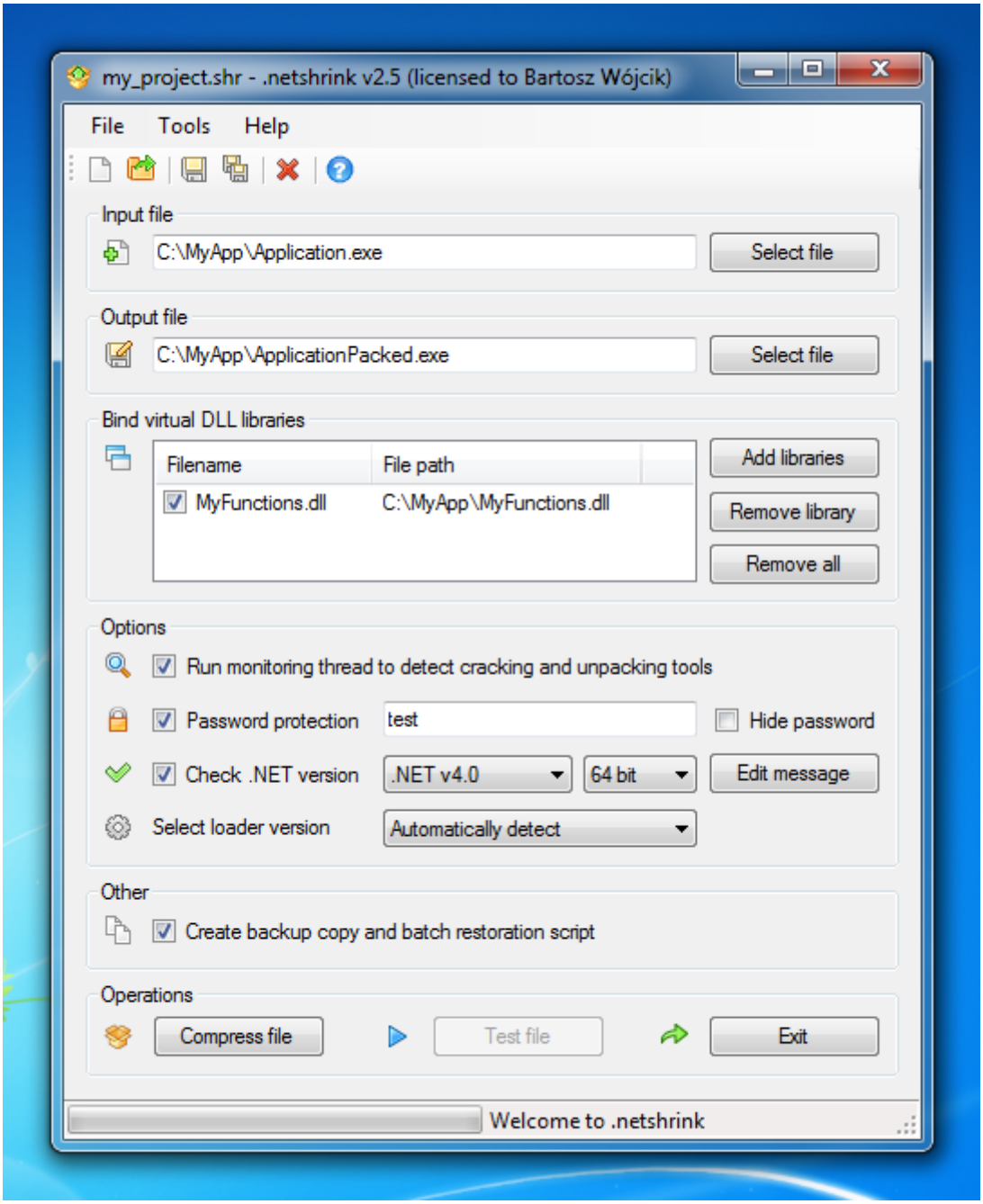
- Szyfrowanie oryginalnego pliku wykonywalnego (całego)
- Umieszczenie go w zasobach programu ładującego lub na końcu pliku (tzw. *overlay*)

Po uruchomieniu tak spreparowanego pliku dzieją się dwie rzeczy w zależności od zaawansowania autora takiego oprogramowania. Albo następuje odszyfrowanie pliku, jego wypakowanie do katalogu tymczasowego i uruchomienie z tego katalogu, albo *odmapowanie* sekcji kodu i danych oryginalnego programu ładującego i załadowanie tam sekcji odszyfrowanego złośliwego oprogramowania, a następnie jego uruchomienie (podmiana obrazu pliku wykonywalnego).

Exe-Packery

Exe-packery takie jak *UPX*, *FSG*, *MEW*, *ASpack*, *.netshrink* stosowane są od dawna do zmniejszenia rozmiarów programów wykonywalnych. Ich zasada działania opiera się na przebudowie podstawowych struktur plików wykonywalnych, które poddawane są kolejno kompresji, do takich danych, dołączany jest program ładujący (tzw. *loader*), zwykle małych rozmiarów, napisany w assemblerze. Po uruchomieniu tak skompresowanego pliku, program ładujący przejmuje kontrolę, dokonuje dekompresji danych i kodu, koryguje struktury pliku wykonywalnego (np. ładuje funkcje z *tabeli importów*). Kolejnym etapem jego działania jest skok do oryginalnego punktu wejściowego aplikacji (tzw. *entrypoint*).

Rysunek 4. Kompresor dla aplikacji .net - .netshrink



Analiza skompresowanych aplikacji nie stanowi problemu i kompresja jedyne, w czym pomaga, to zmiana potencjalnych sygnatur w programach antywirusowych, które w obecnych czasach potrafią automatycznie dokonać dekompresji najpopularniejszych *exe-packerów* (czy to stosując dedykowane moduły dekompresujące, czy też stosując emulację) i przeanalizować oryginalny plik.

Scramblery

Exe-packery znajdują zastosowanie tylko w przypadku zmniejszenia rozmiarów plików wykonywalnych, jednak ich popularność sprawiła, że znaleźli się ludzie, którzy chcieli stosować *exe-packery*, a jednocześnie nieco utrudnić innym łatwe

odtworzenie oryginalnie skompresowanych plików i tak powstały *scramblery*. Zasada ich działania opiera się na modyfikacji skompresowanych już plików. Przykładowo zmieniają nazwy sekcji, dodają dodatkowy kod startowy, który dopiero potem uruchamia kod właściwego programu kompresującego. Przykładem takiego narzędzia jest *UPX-SCRAMBLER* czy *UPolyX*, dla popularnego kompresora *UPX*.

Exe-protectory

Exe-protectory takie jak np. *PELock*, *ASProtect*, *ExeCryptor*, *Armadillo*, to naturalny efekt ewolucji *exe-packerów*. Do kodu odpowiedzialnego za dekompresowanie aplikacji dodane zostały funkcje wykrywające obecność narzędzi do debuggowania, struktury pliku wykonywalnego są niszczone lub ukrywane podczas zabezpieczania, aby utrudnić odtworzenie oryginalnego pliku, który został zabezpieczony. Integralną częścią *exe-protectorów* są również wbudowane systemy licencyjne bazujące na silnych algorytmach kryptograficznych, jak krzywe eliptyczne *ECC* czy szyfrowanie *RSA*.

Popularną techniką zabezpieczającą w *exe-protectorach* jest szyfrowanie wybranych fragmentów kodu, poprzez ich oznaczenie specjalnymi markerami. Kod pomiędzy markerami jest szyfrowany podczas zabezpieczania pliku. Po uruchomieniu tak zabezpieczonej aplikacji, wykonanie kodu pomiędzy markerami szyfrującymi nastąpi tylko w przypadku, jeśli aplikacja wykryje poprawny klucz licencyjny; wówczas kod zaraz przed jego wykonaniem zostanie odszyfrowany, a po wykonaniu ponownie zaszyfrowany w pamięci.

Listing 7. Przykład zastosowania markerów szyfrujących w *exe-protectorze PELock*.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include "pelock.h"

int main(int argc, char *argv[])
{
  // kod pomiędzy makrami DEMO_START i DEMO_END
  // będzie zaszyfrowany w zabezpieczonym pliku
  // i nie będzie dostępny (ani wykonany) bez
```

```
    // poprawnego klucza licencyjnego
    DEMO_START

    printf("Witaj w pełnej wersji mojej aplikacji!");

    DEMO_END

    printf("\n\nNaciśnij dowolny klawisz, aby kontynuować . . .");

    getch();

    return 0;
}
```

Z czasem poziom zaawansowania *exe-protectorów* sprawił, że analiza tak zabezpieczonych plików stała się prawdziwym problemem dla *crackerów* (czyli osób zajmujących się łamaniem oprogramowania, nazwa pochodzi od angielskiego słowa *crack* - łamać). O ile ten problem ma pozytywne skutki dla autorów oprogramowania, to z drugiej strony, analitycy firm antywirusowych mają poważne problemy z analizą bardzo zaawansowanych systemów ochrony wykorzystywanych do zabezpieczania złośliwego oprogramowania. Niejednokrotnie w takich systemach zabezpieczeń można znaleźć *polimorficzne* algorytmy szyfrowania (unikalne algorytmy szyfrowania generowane za każdym razem, gdy zabezpieczany jest plik), mutację kodu aplikacji (serie instrukcji zamieniane są na serię skomplikowanych, równoznacznych instrukcji), wirtualizację kodu (natywny kod zamieniany jest na kod pośredni) oraz cały zestaw najnowszych trików *antydebug*, utrudniających analizę i zrozumienie działania tak zabezpieczonych aplikacji.

Do innych popularnych zabezpieczeń stosowanych w *exe-protectorach* można zaliczyć m.in.:

- *redirecting* tabeli importów – ukrywanie prawdziwych adresów funkcji, z których korzysta aplikacja, ma to na celu utrudnienie odbudowy tabeli importów zabezpieczonego pliku

- relokację obrazu pliku wykonywalnego w losowy obszar pamięci – zapewniająca ochronę przed zrzucaniem obrazu odszyfrowanego pliku wykonywalnego z pamięci
- wykrywanie narzędzi do debuggowania i monitorowania systemu
- aktywne monitorowanie kodu aplikacji w celu wykrycia zmian w kodzie dokonywanych przez np. *trainery* (w grach)
- wykrywanie uruchomienia w środowiskach wirtualnych i piaskownicach
- wykrywanie emulatorów
- ukrywanie struktury np. zasobów poprzez emulację funkcji dostępu do tych elementów
- łączenie wielu plików aplikacji i bibliotek do jednego wyjściowego pliku wykonywalnego i emulowanie ich obecności przez *hooki* na funkcje dostępu

Virtualizery

Virtualizery takie jak *CodeVirtualizer*, *VMProtect*, to kolejny odłam *exe-protectorów*. Są to narzędzia, które umożliwiają zamianę natywnego kodu aplikacji (lub fragmentów kodu, oznaczonych specjalnymi *markerami*) na kod pośredni. Zabezpieczone pliki posiadają wbudowaną maszynę wirtualną dla tak przekonwertowanego kodu.

Analiza aplikacji, w których część funkcji jest zamieniona na kod pośredni jest bardzo trudna. Wynika to z faktu, że nie można do niej wykorzystać standardowych narzędzi takich jak *IDA* czy *OllyDbg*, ponieważ nie znają one formatu kodu pośredniego. Zaawansowane *virtualizery* potrafią także generować za każdym razem unikalny kod i format instrukcji pośrednich, co jeszcze bardziej komplikuje zrozumienie tak zabezpieczonych aplikacji.

W takich sytuacjach pisze się specjalizowane moduły do dekompilacji pośredniego kodu, do kodu maszynowego np. x86. Jest to bardzo żmudna i czasochłonna praca, tak zabezpieczone złośliwe oprogramowanie jest wyjątkowo trudne do przeanalizowania, zwłaszcza, że dodatkowo może wykorzystywać wszystkie dodatkowe techniki zabezpieczeń stosowane w *exe-protectorach*.

Obfuscatory

Terminem *obfuscatorów* (z ang. *obfuscate* – zagmatwać) określa się narzędzia, których celem jest taka modyfikacja skompilowanych aplikacji lub kodów źródłowych, aby jak najbardziej utrudnić ich analizę.

Obfuscatory stosowane są w większości do zabezpieczania aplikacji kompilowanych do pośredniego kodu takich jak *Java* i *.NET*. Zdarzają się również wyjątki, jak obfuscator *Pythia* dla aplikacji napisanych w *Delphi*, jednak najwięcej tych narzędzi powstało dla aplikacji dla *.NET*. Pierwotnie platforma *.NET* powstała na zgliszczach technologii języka *Visual Basic*. Programy tworzone w języku *Visual Basic* do wersji 6 mogły być kompilowane zarówno do natywnego kodu (*x86*) lub do kodu pośredniego, który wymagał dodatkowych bibliotek z silnikiem wirtualnej maszyny. Specyfikacja wirtualnej maszyny *Visual Basic* nie była nigdy upubliczniona i analiza takich aplikacji była utrudniona, jednak metodą prób i błędów powstawały kolejne amatorskie narzędzia pozwalające na dekompilację takich aplikacji.

Wraz z wejściem na rynek języków C#, VB.NET, firma Microsoft udostępniła dokumentację wirtualnej maszyny i okazało się, że dekompilacja kodu takich aplikacji jest bardzo prosta. Powstały takie znane narzędzia jak *.NET Reflector*, które jednym kliknięciem pozwalają na odtworzenie kodów źródłowych zamkniętych aplikacji. Problem ten spowodował, że na rynku pojawiły się dziesiątki narzędzi zabezpieczających kod pośredni IL aplikacji pisanych dla platformy *.NET*. Zabezpieczenia te wykorzystują m.in. takie elementy jak:

- Zmiana kolejności wykonywania kolejnych instrukcji (z linearnego schematu na nielinearny połączony seriami skoków bezwarunkowych)
- Dynamiczne szyfrowanie kodu .NET
- Szyfrowanie zasobów aplikacji .NET
- Szyfrowanie ciągów tekstowych
- Dodatkowa wirtualizacja kodu pośredniego
- Celowe uszkadzanie struktur aplikacji .NET

Na rynku istnieje mnogość aplikacji zabezpieczających, w większości posiadają one podobne mechanizmy ochrony. Mówi się, że każda akcja generuje jakąś reakcję, przesyt narzędzi zabezpieczających (często przeciętnych) sprawił, że powstały dedykowane narzędzia odbezpieczające, aż w końcu w 2011 roku ukazał się program *de4dot*, który jest uniwersalnym *unpackerem* dla zabezpieczeń *.NET* i potrafi odbezpieczyć aplikacje zabezpieczone ponad 20 najpopularniejszymi *obfuscatorami*, takimi jak np. *SmartAssembly*, *.NET Reactor*, *Dotfuscator*, *Eazfuscator* i wiele innych. Można powiedzieć, że jest to prawdziwy policzek w twarz dla branży zabezpieczeń oprogramowania, a *de4dot* jest aktywnie rozwijany i rozszerzany o obsługę kolejnych zabezpieczeń.

Z ciekawostek mogę powiedzieć, że ceny *obfuscatorów* są znacznie wyższe niż ceny narzędzi zabezpieczających natywne aplikacje, co jest o tyle dziwne, że dokumentacja platformy *.NET* jest publicznie dostępna, a poziom zaawansowania technologicznego (i wiedza potrzebna do ich stworzenia) natywnych *exe-protectorów* i *virtualizerów* znacznie przekracza poziom zabezpieczeń stosowanych w *obfuscatorach*, czego najlepszym przykładem jest opisany program *de4dot*, który jednym kliknięciem potrafi odtworzyć większość zabezpieczonych aplikacji *.NET*. Podobnych, uniwersalnych narzędzi nie znajdziemy dla natywnych *protektorów*.

Konflikt interesów

Sytuacja systemów zabezpieczających stała się na tyle poważna zarówno dla twórców tych systemów, jak i firm antywirusowych, że zabezpieczone, legalne i oryginalne programy często są wykrywane przez antywirusy jako *false-positive* (fałszywe wykrycia). Autorzy systemów zabezpieczeń tracą potencjalnych klientów (a raczej ich klienci tracą swoich klientów), a firmy antywirusowe mają problemy z analizą faktycznych złośliwych aplikacji, które też zostały zabezpieczone tymi samymi systemami zabezpieczeń. W ramach organizacji *IEEE Standards Association* prowadzone są prace mające na celu ustandaryzowanie wymiany informacji między twórcami zabezpieczeń a firmami antywirusowymi. System ten został nazwany *TAGGANT* (*tag* – z ang. oznaczyć) i będzie opierać się na tej zasadzie, że każdy zabezpieczony plik komercyjnym systemem zabezpieczeń będzie oznaczony sygnaturą osoby lub firmy, która zakupiła dany pakiet zabezpieczający. Tak zabezpieczone i oznaczone pliki nie będą oznaczane jako *false-positive*. Jeśli jednak pakiet zabezpieczający zostanie wykorzystany do zabezpieczenia złośliwego oprogramowania (np. gdy autor malware zakupi takie oprogramowanie, wykorzystując kradzione informacje o kartach kredytowych), sygnatura tego klienta umieszczona w pliku wykonywalnym trafi na ogólnodostępną czarną listę firm antywirusowych i każda aplikacja oznaczona sygnaturą tego klienta od tej pory będzie traktowana i flagowana jako potencjalnie złośliwe oprogramowanie.

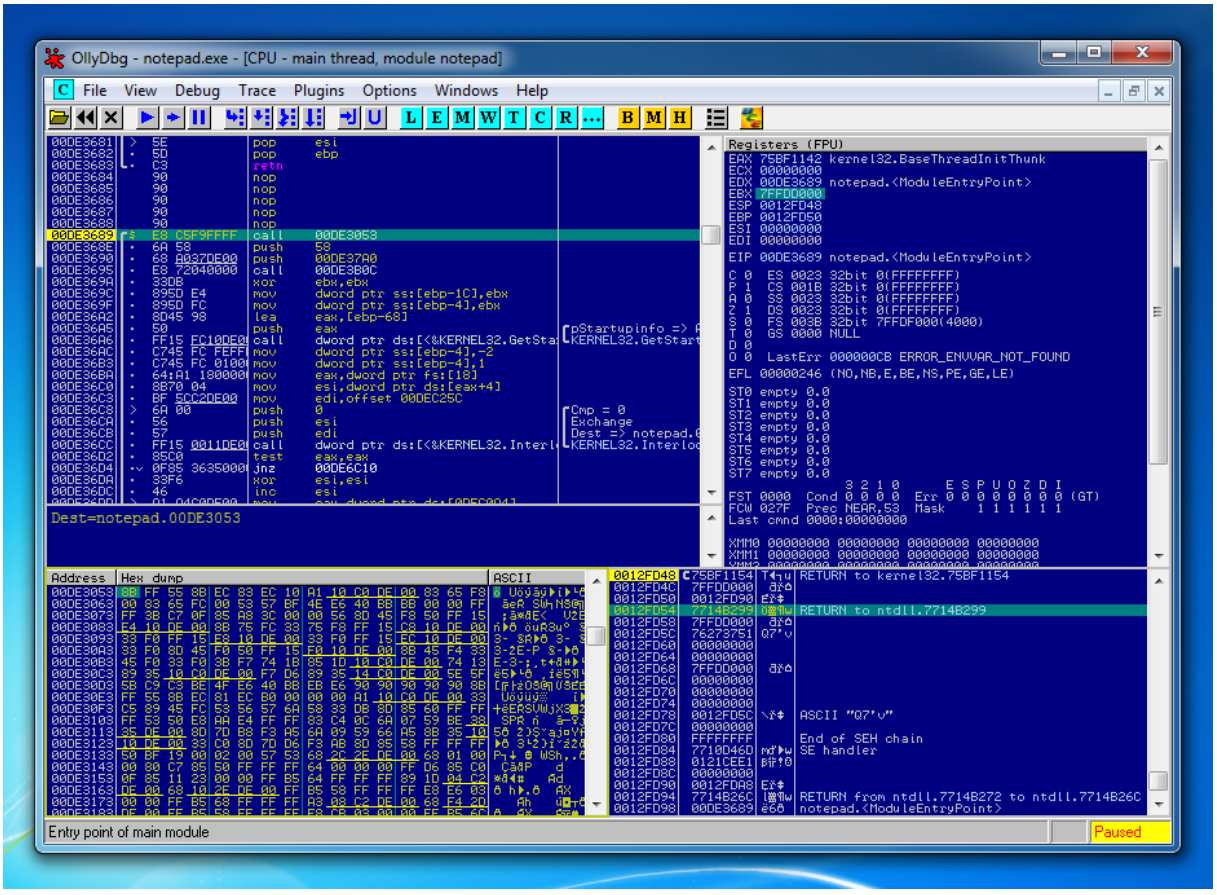
Wykrywanie debuggerów

Jeśli złośliwe oprogramowanie jest bardziej skomplikowane i statyczna analiza kodu w narzędziach takich jak *IDA* (deasembler) czy *HexRays* (dekompilator) nie pozwala analitykowi stwierdzić, co dokładnie robi złośliwe oprogramowanie,

kolejnym narzędziem w rękach analityka jest debugger, czyli narzędzie pozwalające śledzić wykonywanie programu krok po kroku.

Do najczęściej używanych narzędzi do debuggowania można zaliczyć debugger wbudowany w komercyjny deasembler *IDA* (dla 32 i 64 bitowego kodu), darmowy *WinDbg* oraz, w mojej ocenie najpopularniejszy i darmowy - debugger *OllyDbg*, pozwalający śledzić działanie aplikacji w trybie użytkownika (jedynie 32 bitowe aplikacje, co jest jego główną wadą).

Rysunek 5. Okno debuggera OllyDbg2



Popularność debuggera *OllyDbg* sprawiła, że powstało mnóstwo jego modyfikacji i rozszerzeń, które ukrywają jego obecność przed metodami *antydebug* oraz bardzo ułatwiają analizę dowolnych aplikacji, np. poprzez wprowadzenie skryptów automatyzujących wszystkie czynności, które można w nim wykonać (są one wykorzystywane przykładowo do automatycznego rozpakowywania znanych zabezpieczeń).

Z drugiej strony, istnieje cała masa metod, które mogą być wykorzystane do wykrycia tego debuggera i twórcy złośliwego oprogramowanie chętnie sięgają po nie, aby utrudnić dynamiczną analizę kodu.

Listing 8. Wykrywanie debuggera *OllyDbg* (autor Walied Assar).

```
int __cdecl Hhandler(EXCEPTION_RECORD* pRec,void*,unsigned char* pContext,void*)
{
  if(pRec->ExceptionCode==EXCEPTION_BREAKPOINT)
  {
    (*(unsigned long*)(pContext+0xB8))++;
    MessageBox(0,"Expected","waliedassar",0);
    ExitProcess(0);
  }
  return ExceptionContinueSearch;
}

void main()
{
  __asm
  {
    push offset Hhandler
    push dword ptr fs:[0]
    mov dword ptr fs:[0],esp
  }
  RaiseException(EXCEPTION_BREAKPOINT,0,1,0);
  __asm
  {
    pop dword ptr fs:[0]
    pop eax
  }
  MessageBox(0,"OllyDbg Detected","waliedassar",0);
}
```

Kiedyś bardzo znanym debuggerem był słynny *SoftICE*, debugger systemowy (czyli pozwalający śledzić działanie zarówno aplikacji użytkownika, jak i sterowników systemowych). Obecnie jego miejsce zajął *WinDbg*, czyli debugger firmy *Microsoft*, który od lat jest aktywnie rozwijany i uaktualniany. Debuggery systemowe (ang. *kernel mode debugger*) są wykorzystywane do analiz, jeśli oprogramowanie wykorzystuje sterowniki systemowe, dotyczy to np. *rootkitów*, czyli komponentów złośliwego oprogramowania, których głównym celem jest ukrycie właściwego złośliwego oprogramowania w systemie (najczęściej działającego w trybie użytkownika), np. poprzez ukrycie procesów malware lub ukrycie własnych plików przez modyfikację struktur dysku albo ruchu sieciowego.

Częste aktualizacje

Złośliwe oprogramowanie może być wykrywane na podstawie wielu czynników, zaczynając od sum kontrolnych całych plików, przez zaawansowane sygnatury fragmentów pliku, charakterystyczne ciągi znakowe, kolejność wywoływania funkcji (czy wykorzystanie charakterystycznych funkcji, których używa 1/1000 programów), badanie zachowania oprogramowania w systemie etc. Łączna ocena tych cech może decydować o tym, czy oprogramowanie antywirusowe oznaczy taki program jako złośliwy czy nie.

Modyfikacja niektórych elementów może sprawić, że czynniki decydujące o zaklasyfikowaniu oprogramowania jako malware nie będą już takie same jak w poprzednio wykrytym przykładzie. Co może być wykorzystane do zmian, bez zbędnego wysiłku włożonego w modyfikację źródeł?

- Zmiana kompilatora – najprostszy trik, kompilatorów dla języków *C/C++*, *Delphi* i *Visual Basic* jest całkiem sporo i przystosowanie źródeł dla innego kompilatora (jeśli program nie wykorzystuje jakoś intensywnie jego charakterystycznych funkcji) nie stanowi zwykle problemu
- Zmiana opcji kompilatora powodująca natychmiastowe zmiany w strukturach pliku, np. włączenie lub wyłączenie optymalizacji, sposobów przekazywania parametrów do funkcji

Częste aktualizacje w połączeniu z wykorzystaniem opcji kompilatora sprawiają, że wynikowe pliki, mimo że posiadają taką funkcjonalność są zupełnie inaczej skonstruowane na poziomie binarnym.

Modyfikacje funkcji

Wszystkie drogi prowadzą do Rzymu – mówi przysłowie. Również oprogramowanie malware, aby uzyskać określony cel, nie musi korzystać z jednego rozwiązania, może korzystać z wielu równoznacznych technik do osiągnięcia tych samych zadań. Przykładowo, aby uzyskać dostęp do pliku, mogą być wykorzystane funkcje systemu Windows oraz dodatkowych bibliotek:

- KERNEL32.dll – `CreateFileA` (wersja ANSI funkcji otwierającej plik)
- KERNEL32.dll – `CreateFileW` (wersja UNICODE)
- KERNEL32.dll – `CreateFileTransactedA` (transakcyjna wersja funkcji)
- KERNEL32.dll – `CreateFileTransactedW`
- MSVCR100.dll – `fopen` (standardowa funkcja z bibliotek CRT)
- MSVCR100.dll – `fopen_s`
- ...i dziesiątki innych bibliotek, które udostępniają podobne funkcje

Listing 9. Przykład odczytu pliku w różnych wariantach kodu.

```
#include <windows.h>

#include <stdio.h>


// tryb otwierania plików

#define FILE_MODE 3


int main()

{

  #if FILE_MODE == 1

  HANDLE hFile = CreateFileA("notepad.exe", GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING, 0, NULL);

  #elif FILE_MODE == 2

  HANDLE hFile = CreateFileW(L"notepad.exe", GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING, 0, NULL);

  #elif FILE_MODE == 3

  FILE *hFile = fopen("notepad.exe", "rb+");

  #else

  //...

  #endif
```

```
  // zamknij uchwyt pliku
  #if FILE_MODE < 3
  CloseHandle(hFile);
  #else
  fclose(hFile);
  #endif

  return 0;
}
```

Wykorzystanie funkcji z jednej grupy może sugerować jakieś nieprawidłowości w analizowanym oprogramowaniu, jednak tak zmodyfikowane programy, ze względu na inny zestaw funkcji i ich parametrów, za każdym razem będą posiadały inną strukturę kodu.

Skrypty zamiast kodu

Zamiast bezpośredniego wywoływania funkcji systemowych, rolę niektórych zadań mogą przejąć skrypty napisane np. w *VBScript* lub *JScript*. Oprogramowanie malware może również zawierać wbudowane silniki do obsługi takich języków skryptowych jak *Lua* (wirus *Flame* używał skryptów w tym języku), *Python* (np. *IronPython*) czy nawet *PHP* (silnik *PH7*). Analiza takich kodów źródłowych (po ich poprzednim wydobyciu) może być łatwiejsza niż analiza skompilowanego kodu, jednak ukazuje, jak różnymi metodami można osiągnąć ten sam cel.

Modularna budowa

Złośliwe oprogramowanie nie zawsze jest instalowane w całości. Często jest tak, że moduł infekujący jest bardzo małych rozmiarów, a po infekcji malware komunikuje się z serwerem kontrolującym i pobiera brakujące moduły. Taka budowa pozwala również łatwo rozszerzać funkcjonalność oprogramowania oraz aktualizować błędne komponenty oprogramowania. Dla analityka łatwiejsza jest analiza, jeśli całość znajduje się w jednym pliku. Gdy kod malware rozrzucony jest w wielu dodatkowych bibliotekach, może to spowolnić pracę, zwłaszcza jeśli dodatkowe moduły zostały stworzone w różnych językach programowania i zabezpieczone są np. kilkoma różnymi systemami zabezpieczeń.

Niepopularne języki programowania

Oprogramowanie malware tworzone jest w najbardziej popularnych językach programowania, czyli *C/C++*, *Delphi*, *VisualBasic*, *C#*. Wynika to w mojej ocenie z dwóch faktów: pierwszy to możliwość uruchomienia takiego oprogramowania na wielu wersjach systemu Windows (bez konieczności instalacji dodatkowych komponentów systemowych). Drugim faktem jest to, że dla tych języków dostępna jest cała masa powszechnie dostępnych tutoriali, grup dyskusyjnych, przykładów kodów źródłowych, dzięki czemu stworzenie oprogramowania malware nawet dla mało wprawnego programisty jest relatywnie prostym zadaniem.

Niektórzy twórcy malware poszli jednak krok dalej i zorientowali się, że tak tworzone oprogramowanie jest równie łatwo analizowane, ponieważ jest ono najpowszechniejsze. Analitycy firm antywirusowych mają z tak zbudowanymi programami codzienny styk, posiadają odpowiednie narzędzia i rozłożenie takich programów na części pierwsze nie stanowi dla nich najmniejszego problemu.

Problemem są natomiast programy tworzone w niepopularnych językach programowania, jak np. tworzone w językach funkcyjnych lub biznesowych platformach, np.:

- Haskell
- Lisp
- VisualFox Pro
- Fortran
- COBOL
- Egzotyczne kompilatory Visual Basic (np. DarkBasic, PureBasic etc.)

Takie programy posiadają zwykle bardzo skomplikowaną budowę, nieraz ich cały kod jest zapisany w języku pośrednim (*IL - Intermediate Code*) lub wymaga setek megabajtów dodatkowych bibliotek pomocniczych, których analiza jest bardzo trudna, a przede wszystkim czasochłonna. Istnieją dedykowane narzędzia do dekompilacji takich programów, np. dla aplikacji *VisualFox Pro* istnieje dekompilator *ReFox*, jednak takie dedykowane narzędzia to rzadkość.

Do utworzenia słynnego wirusa *Stuxnet* został wykorzystany egzotyczny obiektowy framework dla języka C i analitycy mieli spore problemy z dojściem do tego, w jakim w ogóle języku został stworzony kod ze względu na niespotykaną strukturę kodu.

Na końcu tego artykułu znajdziecie odnośnik do przykładowego programu w formie *CrackMe* – czyli programu specjalnie stworzonego do tego, aby go złamać, napisanego w języku *Haskell*, którego analiza, mimo prostego, kilkulinijkowego algorytmu sprawdzania poprawności klucza licencyjnego, nie należy do najłatwiejszych.

Programy automatyzujące

Na rynku dostępne są gotowe pakiety, pozwalające nawet mało zaawansowanym użytkownikom na tworzenie skomplikowanych aplikacji malware z całą funkcjonalnością dostępu do systemu plików, pobierania komponentów z sieci, dostępem do Rejestru Windows. Mowa tutaj o programach takich jak np.:

- AutoIt
- Winbatch
- Macro Express

Tworzone w nich aplikacje nie wzbudzają podejrzeń, ponieważ ich kod jest zwykle zapisany w formie pośredniej i do ich analizy wymagane są odpowiednie dekompilatory.

Modyfikacje źródeł

Modyfikacje kodów źródłowych należą do bardziej zaawansowanych metod pozwalających uzyskać unikalny kod wynikowy. Jakie zmiany mogą być wprowadzone do kodów źródłowych, aby spowodować zmianę wynikowych plików?

- Mutacja kodów źródłowych np. poprzez wykorzystanie *templates*
- Zmiana kolejności funkcji w plikach źródłowych
- Różne poziomy optymalizacji włączane dla indywidualnych funkcji
- Wprowadzenie fałszywych parametrów do funkcji i ich sztuczne wykorzystanie (optymalizacja kompilatora usunie nieużywane parametry)
- Wprowadzenie transparentnych konstrukcji pomiędzy linie kodu (np. instrukcje zaśmiecające tzw. *junks*, instrukcje skaczące między sobą, fałszywe weryfikacje parametrów funkcji i odwołania do zmiennych lokalnych)
- Nielinearne wykonywanie linii kodu (np. poprzez wykorzystanie `switch`)

- Jeśli to możliwe, to zmiana definicji struktur, tzn. losowe rozłożenie poszczególnych elementów struktur lub wprowadzenie fałszywych (nieużywanych) pól do struktur

Wszystkie te zmiany spowodują drastyczne zmiany w wynikowym pliku wyjściowym.

Listing 10. Instrukcje zaśmiecające w kodzie Delphi.

```
procedure TForm1.FormCreate(Sender: TObject);
begin

  // instrukcje zaśmiecające (nie mają one
  // żadnego wpływu na działanie aplikacji)
  asm
    db 0EBh,02h,0Fh,078h
    db 0EBh,02h,0CDh,20h
    db 0EBh,02h,0Bh,059h
    db 0EBh,02h,038h,045h
    db 0E8h,01h,00h,00h,00h,0BAh,8Dh,64h,24h,004h
    db 07Eh,03h,07Fh,01h,0EEh
    db 0E8h,01h,00h,00h,00h,03Ah,8Dh,64h,24h,004h
    db 0EBh,02h,03Eh,0B8h
  end;

  // ten kod będzie nieczytelny pod
  // deasemblerem, ze względu na
  // instrukcje zaśmiecające
  Form1.Caption := 'Hello world';

  asm
    db 070h,03h,071h,01h,0Ch
    db 0EBh,02h,0Fh,037h
    db 072h,03h,073h,01h,080h
    db 0EBh,02h,0CDh,20h
```

```
    db 0EBh,02h,0Fh,0BBh

    db 078h,03h,079h,01h,0B7h

    db 0EBh,02h,094h,05Ch

    db 0C1h,0F0h,00h

  end;


end;
```

Do najbardziej zaawansowanych mutacji kodu źródłowego, jakie widziałem, zaliczam zabezpieczenia stosowane przez firmę *Syncrosoft* i system *MCFACT* stosowany do ochrony znanego oprogramowania firmy *Steinberg* do obróbki dźwięku - *Cubase*. Kod źródłowy poddawany jest analizie, po czym jest transformowany do zabezpieczonej formy.

Listing 11. Mutacja kodu C++ (przed i po) systemem *MCFACT*.

```
//MCFACT_PROTECTED
unsigned int findInverse(unsigned int n)
{
  unsigned int test = 1;
  unsigned int result = 0;
  unsigned int mask = 1;

  while (test != 0)
  {
    if (mask & test)
    {
      result |= mask;
      //MCFACT_AUTHORIZED
      test -= n;
    }

    mask <<= 1;
    n <<= 1;
  }

  return result;
}

// kod po zabezpieczeniu
unsigned int findInverse(unsigned int n)
{
  ...
  class _calculations_cpp_test_0 test=((_init0_0::_init0)());
  class _calculations_cpp_result_0 result=((_init1_0::_init1)());
  class _calculations_cpp_mask_0 mask=((_init2_0::_init2)());
  _cycle1:{
    signed char tmp8;
    ((tmp8)=((IsNotEqual)((test), (_calculations_cpp_c_0_0))));
    if ((tmp8)) {
    {
      unsigned int tmp7;
      ((And)((mask), (test), (tmp7)));
      if ((tmp7)) {
        ((Or)((result), (mask), (result)));
        ((Sub)((test), (n), (test)));
```

```
      }
      ((ShiftLeftOne)((mask), (mask)));
      ((n)<<=(1U));
    }
    goto _cycle1;
    }
  }
  {
    unsigned int tmp9;
    ((Copy)((result), (tmp9)));
    return(tmp9);
  }
}
```

Analiza tak zabezpieczonego kodu stanowi prawdziwy problem. Zabezpieczenie to zostało ostatecznie złamane przez grupę *warez AiR*, jednak z informacji zawartych w *NFO* (pliku dołączonym do złamanego pakietu oprogramowania) wynikało, że analiza zajęła im 4000 godzin, czyli prawie pół roku pracy! Przez ten czas producent oprogramowania mógł spokojnie sprzedawać oprogramowanie i nie odczuł tak skutków działania piratów. Podobny system zabezpieczenia produkowała firma *Cloakware* (obecnie *Irdeto*), jednak projekt ten został porzucony. Sądzę, że zastosowanie tak zaawansowanego systemu zabezpieczającego do ochrony malware wraz z zastosowaniem wirtualizacji byłoby katastrofalne dla użytkowników oraz firm antywirusowych. Na szczęście systemy takie są bardzo trudne do zbudowania oraz nie są tak powszechnie dostępne.

Szyfrowanie danych i ciągów znakowych

Jest to najczęściej spotykana metoda do ukrycia jakichś wrażliwych danych. Szyfrowane są stałe ciągi znakowe, zawierające np. nazwy plików do zainfekowania, adresy serwerów kontrolujących czy nawet hasła do serwerów FTP / skrzynek pocztowych (tak, hasła do prywatnych repozytoriów są publikowane, w zaszyfrowanej formie, ale zawsze, w co może trudno uwierzyć...), gdzie niektóre programy malware przesyłają wykradzione dane z komputerów użytkowników. Szyfrowanie jest też powszechnie używane do ukrywania komunikacji malware ze swoimi serwerami kontrolującymi.

Rootkit *Rustock* wykorzystywał szyfrowanie algorytmem *RC4* (jeden z najczęściej spotykanych algorytmów w malware) do zabezpieczenia własnych modułów przed analizą. Podczas infekcji pobierany był sprzętowy identyfikator komputera i stanowił on klucz szyfrujący dla pliku sterownika wirusa. Chodziło o to, że taki plik nie uruchamiał się na innych komputerach (nie zgadzał się sprzętowy identyfikator) oraz jednocześnie nie była możliwa jego analiza na innym komputerze (np. gdy plik został wysłany do zbadania do firmy antywirusowej).

Należało pobrać sprzętowy identyfikator z zainfekowanego komputera i dopiero wtedy można było odszyfrować kod wirusa.

Znane algorytmy szyfrowania (np. *AES*) posiadają stałe tablice pomocnicze i narzędzia wykorzystywane do analizy oprogramowania potrafią je odnaleźć w kodzie (np. skaner sygnatur *PEiD*), co często jest sygnałem dla analityka, że coś tam zaszyfrowanego w środku siedzi i być może warto się tym zainteresować, dlatego autorzy malware czasami korzystają z dynamicznie generowanych algorytmów szyfrowania, które nie wzbudzają podejrzeń, np.:

Listing 12. Dynamicznie wygenerowany kod deszyfratora.

```
// encrypted with www.stringencrypt.com (v1.0.0) [C/C++]

// wszLabel = "C/C++ String Encryption"

wchar_t wszLabel[24] = { 0x1249, 0x1275, 0x1261, 0x1279,

                         0x1279, 0x1284, 0x1239, 0x1218,

                         0x121A, 0x1243, 0x1216, 0x1245,

                         0x138C, 0x1267, 0x121E, 0x1249,

                         0x121A, 0x1233, 0x121C, 0x1230,

                         0x121B, 0x121D, 0x121E, 0x1284 };


for (unsigned int EpKHC = 0, HNUrJ = 0; EpKHC < 24; EpKHC++)

{

  HNUrJ = wszLabel[EpKHC];

  HNUrJ -= EpKHC;

  HNUrJ = ~HNUrJ;

  HNUrJ ^= 0x8D7C;

  HNUrJ += 0xFC5C;

  HNUrJ ^= 0xE617;

  HNUrJ -= 0xBB74;

  HNUrJ = ~HNUrJ;

  HNUrJ -= EpKHC;

  HNUrJ ++;

  wszLabel[EpKHC] = HNUrJ;

}
```

Bomby czasowe

Czasami programy malware posiadają mechanizmy opóźniające aktywację swoich funkcji, przykładowo sprawdzają lokalny czas i swoje działanie rozpoczynają po ściśle określonej dacie. Takie oprogramowanie może nie wzbudzać podejrzeń (żadne zmiany nie są obserwowane w systemie), jeśli jednak z wcześniejszych analiz wyszło, że oprogramowanie nie do końca jest czyste, konieczna jest jego dalsza weryfikacja.

Listing 13. Uruchomienie złośliwego kodu po ustalonej dacie.

```
#include <windows.h>


void MrMalware(void)
{
  // złośliwy kod
  MessageBox(NULL, "Jestem wirusem!", "Bu!", MB_ICONWARNING);
  return;
}


int main()
{
  SYSTEMTIME stLocalTime = { 0 };

  // pobierz lokalny czas
  GetLocalTime(&stLocalTime);

  // uruchom złośliwy kod tylko
  // po określonej dacie
  if (stLocalTime.wYear >= 2013 && \
      stLocalTime.wMonth >= 5 && \
      stLocalTime.wDay >= 2)
  {
    MrMalware();
  }
```

```
  return 0;
}
```

Co w takiej sytuacji zrobić, jeśli podejrzewamy, że być może oprogramowanie posiada taki mechanizm (co wcale nie jest takie oczywiste)? Najprostszym rozwiązanie jest analiza ze zmienioną datą systemową ustawioną w przód lub w tył i obserwowanie zmian w systemie narzędziami monitorującymi. Mimo swojej prostoty, bomby czasowe są bardzo skuteczną bronią przed ukrywaniem złośliwej funkcjonalności.

Opóźnione wykonywanie

Oprócz bomb czasowych, niektóre programy malware stosują także opóźnienia czasowe, bazujące np. na czasomierzach (ang. *timer*). Technika polega na tym, że kod złośliwego oprogramowania jest wykonywany np. z godzinnym opóźnieniem w stosunku do uruchomienia tego programu. Co to ma na celu? Po pierwsze – zmylenie analityka, uruchamiając podejrzane oprogramowanie, nic się nie dzieje. Wniosek? Oprogramowanie jest czyste i konieczna byłaby dodatkowa analiza. Po drugie – opóźnienia czasowe potrafią skutecznie zmylić emulatory stosowane w samym oprogramowaniu antywirusowym. Emulatory śledzą kod podejrzanego oprogramowania, próbując dojść jak najdalej, emulując kolejne napotkane instrukcje w kodzie (w przypadku np. zaszyfrowanego programu), jednak posiadają ograniczenia czasowe, bo nie może być takiej sytuacji, że użytkownik uruchamia program i czeka pół godziny, aż emulator dojdzie do końca zaszyfrowanej aplikacji. Dobre emulatory posiadają wykrywanie takich opóźnień czasowych (np. funkcji `Sleep`, pętli opóźniających etc.), jednak nie wszystkie te funkcje brane są pod uwagę, a mnogość dostępnych metod odmierzających czas w systemie Windows sprawia, że takie opóźnienia są równie dobrą metodą przeciwko dynamicznej analizie kodu.

Listing 14. Uruchomienie złośliwego kodu po określonym czasie.

```
#include <windows.h>


DWORD dwTimerId = 0;

const DWORD dwTimerEventId = 666;


// funkcja callback uruchamiana po określonym czasie
```

```
VOID CALLBACK MrMalware(HWND hwnd, UINT uMsg, UINT idEvent, DWORD dwTime)
{
  // złośliwy kod
  MessageBox(NULL, "Jestem wirusem!", "Bu!", MB_ICONWARNING);
  return;
}

int main()
{
  // uruchom złośliwy kod po 5 sekundach
  // od uruchomienia programu
  dwTimerId = SetTimer(NULL, dwTimerEventId, 5 * 1000, MrMalware);

  MessageBox(NULL, "Jestem przyjaznym programem", "Hi!", MB_ICONINFORMATION);

  return 0;
}
```

Cyfrowe sygnatury

Cyfrowe sygnatury wykorzystywane są do podpisywania aplikacji, są one jakby gwarantem, że podpisana cyfrowo aplikacja pochodzi z zaufanego źródła. Niektóre programy antywirusowe po wykryciu cyfrowo podpisanej aplikacji nie przystępują do jej dalszej weryfikacji, niezależnie co to za aplikacja. Aby uzyskać cyfrowy certyfikat, należy przejść weryfikację personalną oraz weryfikację firmy, na którą ma być wystawiony certyfikat. Bez odpowiednich dokumentów (np. wyciągów z kont bankowych, rachunków za telefon, skanów dokumentów personalnych) nie uzyskamy go. Jednak zdarzają się przypadki, że cyfrowe certyfikaty są wykradane np. w wyniku ataków hakerskich i później wykorzystywane do podpisywania oprogramowania malware. Przypadki takie są sporadyczne, jednak rzuciły cień na funkcjonowanie oprogramowania antywirusowego, które bezgranicznie zaufało tak podpisanym programom.

Z ciekawostek mogę powiedzieć, że zdarzają się również komiczne sytuacje, np. w przypadku oprogramowania antywirusowego firmy *Comodo*, nawet cyfrowo podpisywane aplikacje czasami zgłaszane są jako podejrzane wraz z

komunikatem, że są podpisane cyfrowo, jednak nie certyfikatem pochodzącym z firmy *Comodo* (która takie certyfikaty także wystawia).

Skanery online

Twórcy malware, aby skutecznie rozprowadzać swoje programy, zdają sobie sprawę, że muszą sprawdzić czy ich oprogramowanie jest wykrywane przez programy antywirusowe. Instalowanie wszystkich popularnych pakietów zajęłoby sporo czasu (nie mówiąc już nawet o tym, że nie są kompatybilne ze sobą), dlatego do tego celu wykorzystywane są skanery online, które posiadają wbudowane skanowanie przez dziesiątki pakietów antywirusowych. Jeśli pomyślałeś teraz o usłudze *VirusTotal* (zakupionej niedawno przez firmę Google), to niewiele się pomyliłeś, jednak rozwiązania takie jak *VirusTotal* czy skaner online *Jotti* nie są wykorzystywane przez twórców malware z tego względu, że próbki wysyłane na stronę są później rozprowadzane wśród firm antywirusowych. Do takich celów powstały strony takie jak np. *NoVirusThanks*, które chwalą się tym, że nie rozprowadzają próbek wśród firm antywirusowych.

Wnioski

Z moich doświadczeń i obserwacji wynika, że trwa nieustanna walka pomiędzy tymi, którzy oprogramowanie analizują, zarówno w tych dobrych celach, jak firmy antywirusowe, jak i w tych złych celach, jak np. *crackerzy* łamiący zabezpieczenia w oprogramowaniu, a między tymi, którzy tworzą zabezpieczenia (i osobami, które je wykorzystują nie zawsze do dobrych celów). Patrząc na ewolucję narzędzi, zaczynając od *exe-packerów* i na obecnie stosowane zabezpieczenia w postaci *virtualizerów*, można się zastanawiać, jaki będzie kolejny krok w zabezpieczeniach? Moim zdaniem wszystko pójdzie w kierunku wirtualizacji kodu, tylko jeszcze bardziej zaawansowanej niż obecnie. Widać to po obecnych trendach na rynku zabezpieczeń natywnych aplikacji oraz pojawiających się *virtualizerach* dla aplikacji *.NET*. Jak poradzą sobie z tym firmy antywirusowe? Sądzę, że tak samo jak zwykle – mozolną pracą rękami swoich zdolnych pracowników.

Źródła

- Środowisko VMware – http://www.vmware.com
- Środowisko VirtualBox – http://www.virtualbox.org
- Środowisko Parallels – http://www.parallels.com
- Piaskownica Cuckoo Sandbox – http://www.cuckoosandbox.org

- Piaskownica Norman Sandbox – http://www.norman.com
- Piaskownica Anubis Sandbox – http://anubis.iseclab.org
- Piaskownica Buster Sandbox Analyzer – http://bsa.isoftware.nl
- Piaskownica Sandboxie – http://www.sandboxie.com
- Deasembler IDA i dekompilator HexRays – http://www.hex-rays.com
- Debugger OllyDbg – http://www.ollydbg.de
- Debugger WinDbg – http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx
- Dekompilator .NET Reflector – http://www.reflector.net
- Analiza wirusa Stuxnet i jego kompilatora – http://www.networkworld.com/community/node/80008
- Analiza wirusa Rustock – http://www.eset.com/us/resources/white-papers/Yet_Another_Rustock_Analysis.pdf
- Narzędzia do analizy aplikacji .NET – http://www.secnews.pl/2011/11/17/narzedzia-do-analizy-aplikacji-net/
- Sniffer sieciowy Wireshark – http://www.wireshark.org
- Skaner sygnatur popularnych zabezpieczeń PEiD – http://www.peid.info
- Obfuscator Pythia dla Delphi - http://www.the-interweb.com/serendipity/index.php?/archives/86-Pythia-1.1.html
- Uniweralny unpaker de4dot – https://bitbucket.org/0xd4d/de4dot
- CrackMe w Haskellu – http://www.secnews.pl/2012/10/14/proste-crackme/
- Dynamiczne szyfrowanie ciągów znakowych – http://www.stringencrypt.com
- Skaner VirusTotal – http://www.virustotal.com
- Skaner Jotti – http://virusscan.jotti.org/en
- Skaner NoVirusThanks – http://vscan.novirusthanks.org

O Autorze

Bartosz Wójcik (support@pelock.com)

Autor zajmuje się systemami ochrony oprogramowania przed złamaniem (http://www.pelock.com), zaawansowaną analizą wsteczną kodu (*reverse engineering*), tematy te często porusza na swoim blogu (http://www.secnews.pl). Wykonuje również profesjonalne audyty bezpieczeństwa oprogramowania pod względem podatności na analizę i ochronę przed złamaniem.