MAREK ZMYSŁOWSKI

Metody wykrywania debuggerów

Stopień trudności

Im więcej wiemy o przeciwniku tym skuteczniej potrafimy z nim walczyć oraz zabezpieczać się przed nim. Ale tę zasadę stosują obie strony. Nie tylko specjaliści od bezpieczeństwa starają się poznać szkodliwy kod, ale również twórcy złośliwego oprogramowania starają się przed nimi zabezpieczyć i ukryć.

W świecie Internetu coraz większe spustoszenie sieją różnego rodzaju szkodliwe programy – trojany, robaki czy malware. Do walki stają specjaliści od spraw bezpieczeństwa, którzy starają się unieszkodliwić lub uniemożliwić działanie tym programom. Wyposażeni w odpowiednie oprogramowanie dążą do zrozumienia, jak działa szkodliwy kod. W tym celu mogą posługiwać się wieloma narzędziami do analizy, które dają im gigantyczne możliwości.

Jeden z takich programów to IDA Pro, który jest swoistym kombajnem do analizy oprogramowania. Jednak szkodliwe programy nie są w tej kwestii bezbronne. Istnieje wiele metod polegających na zabezpieczeniu się i ukryciu mechanizmów działania przed tego typu analizą.

Poniższy artykuł prezentuje w jaki sposób działający proces może wykryć czy poddawany jest analizie przez debugger. Samo ukrywanie i zaciemnianie kodu stanowi zupełnie odrębny i obszerny temat. Artykuł ten nie ma na celu pomóc twórcom szkodliwego oprogramowania, ale ma zaprezentować mechanizmy, którymi twórcy ci się posługują, aby można je było lepiej wykrywać. Przedstawione w nim metody zostały uszeregowane w czterech grupach – w zależności od sposobu działania oraz rodzaju funkcji z jakich korzystają.

Wszystkie przedstawione przykłady zostały skompilowane przy użyciu Microsoft Visual Studio 2008 Express Edition w systemie Windows XP SP2. Wykorzystane zostały również debuggery: OllyDbg w wersji 1.10 oraz IDA Pro w wersji 5.2.0.

Metody wykorzystujące informacje o procesie

Te metody opierają się głównie na informacji o samym procesie. Są to odpowiednie zmienne lub funkcje , które w sposób bezpośredni informują nas o debugowaniu programu.

- Funkcja `IsDebuggerPresent`

To najprostszy sposób sprawdzenia, czy proces debugowany – należy zapytać o to system. Funkcja zwraca 1, jeżeli nasz proces jest podłączony do debuggera, 0 jeżeli nie. Listing 1. prezentuje fragment kodu, który wykorzystują tę metodę.

- Odczyt wartości `BeginDebugged` ze struktury `PEB` procesu

Metoda ta opiera się o identyczny mechanizm, jak metoda przedstawiona powyżej. Tutaj jednak zamiast wywołania funkcji sami sprawdzamy wartość odpowiedniego pola struktury `PEB` (ang. *process environment*

Z ARTYKUŁU DOWIESZ SIĘ

przy pomocy jakich metod i mechanizmów dany proces może sprawdzić czy poddawany jest analizie przy użyciu debuggera,

jak zaimplementować dane mechanizmy.

CO POWINIENEŚ WIEDZIEĆ

zasady programowania w C++ oraz asemblerze,

jak używać Visual Studio C++, OllyDbg oraz IDA Pro,

jak korzystać z Windows API,

podstawowe zasady obsługi wyjątków w systemie Windows.

block) procesu. Struktura ta w różny sposób opisuje dany proces. Dla każdego procesu znajduje się ona zawsze pod tym samym adresem `fs:[30h]`. Jednym z pól tej struktury jest `BeingDebugged`. Wartość 1 oznacza, że proces podłączony został do debuggera. Listing 2. prezentuje fragment kodu, za pomocą którego można sprawdzić wartość tego pola. Wykorzystana została wstawka asemblerowa uproszczenia kodu.

- Funkcja `CheckRemoteDebuggerPresent`

Funkcja ta sprawdza, czy proces podłączony został do zdalnego debuggera. Słowo *zdalny* rozumiane jest przez Microsoft jako odrębny proces, nie koniecznie działający na innej maszynie. Obecnie na oficjalnej stronie MSDN funkcja ta jest zalecana zamiast dwóch metod opisanych powyżej. Wynika to z nieokreślonej przyszłości struktury `PEB`, która w kolejnych wersja Windowsa może się nie pojawić. Listing 3. prezentuje, w jaki sposób wykorzystywać opisaną funkcję.

- Funkcja `NtQueryInformationProcess`

Funkcja ta umożliwia pobranie różnych informacji na temat procesu. W tym jednak przypadku można ją wykorzystać podobnie jak robi to funkcja `CheckRemoteDebuggerPresent`, która w ten sposób sprawdza obecność debuggera. Aby to zrobić należy ustawić parametr funkcji `ProcessInformationClass` na wartość `ProcessDebugPort` (0x07). Funkcja `NtQueryInformationProcess` nie jest dostępna poprzez API, wtedy należy jej adres pobrać bezpośrednio z pliku *ntdll.dll*. Jeżeli funkcja wykona się poprawnie oraz wartość parametru `ProcessInformation` zostanie ustawiona na -1 to proces jest debugowany. Listing 4. prezentuje kod funkcji, która sprawdza wspomniany parametr i zwraca `true`, jeśli proces jest debugowany lub `false` jeśli nie.

- Odczyt wartości `NtGlobalFlag` ze struktury `PEB` procesu

Struktura nie została całkowicie opisana na oficjalnej stronie MSDN. Aby uzyskać nieco więcej informacji warto odwiedzić stronę, na której znajdują się nieudokumentowane struktury oraz funkcje systemu Windows. Adres tej

Listing 1. *Wykorzystanie funkcji IsDebuggerPresent*

```
if(IsDebuggerPresent())
{
   cout << "  -  Debugger odnaleziony\n";
}
else
{
   cout << "  -  Nie odnaleziono debuggera\n";
}
```

Listing 2. *Odczyt wartości BeginDebugged ze struktury PEB procesu*

```
char IsDbgPresent = 0;

__asm
{
   mov eax, fs:[30h]     // Adres struktury PEB procesu
   mov al, [eax + 02h]  // Adres zmiennej BeginDebugged
   mov IsDbgPresent, al
}
if(IsDbgPresent)
{
   cout << "  -  Debugger odnaleziony\n";
}

else
{
   cout << "  -  Nie odnaleziono debuggera\n";
}
```

Listing 3. *Wykorzystanie funkcji CheckRemoteDebuggerPresent*

```
BOOL IsRemoteDbgPresent = FALSE;
CheckRemoteDebuggerPresent(GetCurrentProcess(), &IsRemoteDbgPresent);
if(IsRemoteDbgPresent)
{
   cout << "  -  Debugger odnaleziony\n";
}
else
{
   cout << "  -  Nie odnaleziono debuggera\n";
}
```

Listing 4. *Wykorzystanie funkcji NtQueryInformationProcess*

```
bool NtQueryInformationProcessTest()
{
   typedef NTSTATUS (WINAPI *pNtQueryInformationProcess)
         (HANDLE ,UINT ,PVOID ,ULONG , PULONG);
   HANDLE hDebugObject = NULL;
   NTSTATUS Status;
   // Pobranie adresu funkcji
   pNtQueryInformationProcess NtQueryInformationProcess = (pNtQueryInformationPro
                     cess)

         GetProcAddress(GetModuleHandle(TEXT("ntdll.dll")), "NtQueryInformationPro
                     cess" );
   Status = NtQueryInformationProcess(GetCurrentProcess(),7, &hDebugObject, 4,
                     NULL);
   if(Status == 0x00000000 && hDebugObject == (HANDLE)-1)
      return true;
   else
      return false;

}
```

strony to *http://undocumented.ntinternals.net/*. Na stronie tej znaleźć można dokładny opis struktury `PEB`.

`NtGlobalFlag` to pole, które definiuje w jaki sposób ma zachowywać się uruchomiony proces. Podczas normalnego działania (proces nie jest debugowany) jego wartość ustawiona jest na 0. W przeciwnym przypadku ustawione są następujące flagi:

```
FLG_HEAP_ENABLE_TAIL_CHECK (0x10),
FLG_HEAP_ENABLE_FREE_CHECK (0x20),
FLG_HEAP_VALIDATE_PARAMETERS (0x40).
```

Listing 5. pokazuje, w jaki sposób sprawdzić czy flagi te zostały ustawione.

Listing 5. *Odczyt wartości NtGlobalFlag ze struktury PEB procesu*

```
unsigned long NtGlobalFlags = 0;
__asm
{
   mov eax, fs:[30h]
   mov eax, [eax + 68h]
   mov NtGlobalFlags, eax
}
if(NtGlobalFlags & 0x70)
{
   cout << "  -  Debugger odnaleziony\n";
}
else
{

   cout << "  -  Nie odnaleziono debuggera\n";
}
```

Listing 6. *Odczyt wartości HeapFlags ze struktury PEB.ProcessHeap procesu*

```
unsigned long HeapFlags = 0;
__asm
{
   mov eax, fs:[30h]                //Adres struktury PEB
   mov eax, [eax+18h]                 //Adres struktury ProcessHeap

   mov eax, [eax+0Ch]           //Adres pola HeapFlags
   mov HeapFlags, eax
}
if(HeapFlags & 0x20)
{
 cout << "  -  Debugger odnaleziony\n";
}
else
{
 cout << "  -  Nie odnaleziono debuggera\n";

}
```

Listing 7. *Odczyt wartości HeapFlags ze struktury PEB.ProcessHeap procesu*

```
unsigned long ForceFlags = 0;
__asm
{
   mov eax, fs:[30h]          //Adres struktury PEB

   mov eax, [eax+18h]             //Adres struktury Heap
   mov eax, [eax+10h]           //Adres pola ForceFlags
   mov ForceFlags, eax
}
if(ForceFlags)
{
   cout << "  -  Debugger odnaleziony\n";
}
else
{
   cout << "  -  Nie odnaleziono debuggera\n";
}
```

Wartość 0x70 występująca w warunku jest sumą bitową powyższych flag `(FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS)`.

- Odczyt wartości `HeapFlags` ze struktury `PEB.ProcessHeap` procesu

`ProcessHeap` to kolejna struktura, której nie znalazła się na oficjalnej stronie. Służy ona do opisu sterty danego procesu oraz jej zachowania się. Dlatego też proces poddany debugowaniu musi ustawić nieco inne opcje. Wystarczy więc sprawdzić pole `HeapFlags`. Podczas normalnego działania procesu wartość ustawiona jest na 0x20 (`HEAP_GROWABLE`). W momencie gdy proces uruchomiony jest przez debugger, dodawane są dwie flagi:

```
HEAP_TAIL_CHECKING_ENABLED (0x20)
HEAP_FREE_CHECKING_ENABLED (0x40).
```

HeapFlags ma wtedy zazwyczaj wartość 0x50000062 ale jest ona uzależniona od wartości `NtGlobalFlag`. Listing 6. prezentuje, w jaki sposób można wykorzystać to pole.

- Odczyt wartości `ForceFlags` ze struktury `PEB.ProcessHeap` procesu

Wartość tego pola również steruje zachowaniem sterty. W tym przypadku 0 oznacza, że proces nie jest debugowany, natomiast wartość różna od 0 (zazwyczaj 0x40000060) , że proces jest debugowany. Listing 7. prezentuje wykorzystanie tej metody.

Metody wykorzystujące breakpointy

Breakpoint: to sygnał wysyłany do debuggera, który mówi, aby zawiesił on wykonywanie programu w danym punkcie. Program ten przechodzi wtedy w tryb debugowania (ang. *debug mode*). Przejście w ten tryb nie kończy programu, ale umożliwia jego dalsze wykonanie w dowolnym momencie.

Breakpointy są podstawą działania debuggerów, dlatego też mogą stanowić bardzo silne narzędzie podczas ich wykrywania.

- INT 3

To przerwanie jest używane przez debuggery do ustawiania *software breakpoint* (przerwanie programowe). Debugger, w miejscu w którym chcemy zatrzymać działanie programu, wstawia kod przerwania (`0xcc`) zamiast instrukcji.

Napotkanie takiej instrukcji powoduje wystąpienie wyjątku, który obsługiwany jest przez debugger.

Po zakończeniu obsługi (na przykład użytkownik każe debuggerowi kontynuować) nastąpi powrót do dalszego wykonywania programu. Aby wykryć debugger należy podmienić funkcję obsługującą wyjątki, a następnie wykonać instrukcję `INT 3`. Jeżeli nie zostanie wykonana nasza funkcja obsługująca wyjątek to znaczy, że zrobił to za nas debugger. Listing 8. zawiera funkcję obsługującą wyjątek. Funkcja ta ustawia starą ramkę stosu oraz miejsce w którym należy kontynuować dalsze wykonanie. Na Listingu 9. został umieszczony kod, który ustawia tę funkcję jako obsługującą wyjątek. Do funkcji tej, poprzez stos przekazywany jest adres miejsca oznaczonego etykietą `end`. Jeżeli debugger obsłuży wyjątek wtedy zostanie wykonana linijka `mov Int3Value, 1` i wartość `Int3Value` zostanie ustawiona na 1. Jeżeli natomiast nasza funkcja obsłuży wyjątek, to wtedy wykonana zostanie instrukcja zaczynająca się w miejscu znaczonym jak `end` – linijka z ustawieniem wartości `Int3Value` zostanie pominięta.

Prostota tej metody sprawia, że działa ona tylko na słabe debuggery. Nowoczesne programy wykrywają ustawienie funkcji obsługującej wyjątek i po wznowieniu działania do niej przekazują dalsze wykonywanie. Zarówno debugger z Visual Studio jak i OllyDbg dają się oszukać. Natomiast IDA Pro pyta się czy przekazać obsługę

Listing 8. *Funkcja obsługująca wyjątek*

```
EXCEPTION_DISPOSITION __cdecl
      exceptionhandler (struct _EXCEPTION_RECORD *ExceptionRecord, void *
                        EstablisherFrame,
      struct _CONTEXT *ContextRecord, void * DispatcherContext )
{
   ContextRecord->Eip = *(((DWORD *)EstablisherFrame)+2);
   ContextRecord->Ebp = *(((DWORD *)EstablisherFrame)+3);
   return ExceptionContinueExecution;
}
```

Listing 9. *Fragment kodu ustawiający nową funkcję obsługi wyjątku*

```
unsigned long Int3Value = 0;
__asm
{
   push ebp                // Adres ramki stosu
   push offset end     // Adres miejsca kontynuacji po obsłudze przerwania

   push exceptionhandler
   push fs:[0]
   mov fs:[0], esp
   int 3
   mov Int3Value, 1
   end:
   mov eax, [esp]
   mov fs:[0], eax
   add esp, 16
}
if(Int3Value)
{
   cout << "  -  Debugger odnaleziony\n";
}
else
{
   cout << "  -  Nie odnaleziono debuggera\n";
}
```

Listing 10. *Fragment kodu ustawiający nową funkcję obsługi wyjątku oraz uruchamiający Ice breakpoint*

```
unsigned long IceBreakValue = 0;
__asm
{
   push ebp                // Adres ramki stosu
   push offset end      // Adres miejsca kontynuacji po obsłudze przerwania

   push exceptionhandler

   push fs:[0]
   mov fs:[0], esp

   __emit 0F1h
   mov IceBreakValue, 1
   end:
   mov eax, [esp]
   mov fs:[0], eax
   add esp, 16
}
if(IceBreakValue)
{
   cout << "  -  Debugger odnaleziony\n";
}
else
{
   cout << "  -  Nie odnaleziono debuggera\n";
}
```

Listing 11. *Fragment kodu wykorzystujący memory breakpoint*

```
DWORD OldProtect = 0;
void *pAllocation = NULL;

pAllocation = VirtualAlloc(NULL, 1, MEM_COMMIT | MEM_RESERVE,
      PAGE_EXECUTE_READWRITE);
if (pAllocation != NULL)
{
   *(unsigned char*)pAllocation = 0xC3;    // Ustawienie kodu funkcji RET
   if (VirtualProtect(pAllocation, 1,PAGE_EXECUTE_READWRITE | PAGE_GUARD,
      &OldProtect) == 0)

   {
      cout << "Nie udało się ustawić odpowiedniej flagi" << endl;
   }
   else
   {
      __try
      {
         __asm
         {
            mov eax, pAllocation              // Zapis adresu pamięci do rejestru
                     eax
            push MemBreakDbg    // Umieszczenie adresu MemBreakDbg na stosie
            jmp eax                        // Wykonanie kodu spod adresu zawartego
                     w eax
                                           // Jeżeli zostanie wykonany, to
                     funkcja RET powróci
                                           // do wykonywania kodu pod adresem
                     umieszczonym na stosie
                                           // czyli od miejsca oznaczonego
                     jako MemBreakDbg
         }
      }
      __except(EXCEPTION_EXECUTE_HANDLER)
      {
         cout <<  "  -  Debugger nie odnaleziony\n";
         __asm {jmp MemBreakEnd}
      }
      __asm{MemBreakDbg:}
      cout << "  -  Debugger odnaleziony\n";
      __asm{MemBreakEnd:}
      VirtualFree(pAllocation, NULL, MEM_RELEASE);
   }
}
else
{
   cout <<"Nie udało się zaalokować pamięci" << endl;
}
```

Listing 12. *Fragment kodu ustawiający funkcję obsługi wyjątków i generujący wyjątek*

```
__asm

{
   push ebp
   push offset end
   push hardbreakhandler
   push fs:[0]
   mov fs:[0],esp
   xor eax, eax
   div eax

   end:
   mov eax, [esp]
   mov fs:[0], eax
   add esp, 16
}
```

wyjątku do aplikacji. Jeśli się zgodzimy metoda ta nie wykryje istnienia debuggera.

- *Ice breakpoint*

Ice breakpoint polega na wykorzystaniu nieudokumentowanej instrukcji procesorów Intela o kodzie `0xF1h`. Stosuje się ją do wykrywania programów śledzących. Wykonanie tej instrukcji powoduje wystąpienie wyjątku `SINGLE_STEP`. Jeżeli proces jest debugowany, debugger potraktuje to jako normalne polecenie wykonania pojedynczej instrukcji (ang. *single step*) i przejdzie do następnej w kolejce. W przypadku braku debuggera zostanie uruchomiona normalna procedura obsługi wyjątków. W zaprezentowanym przykładzie na Listingu 10. ustawiana jest nasza funkcja obsługi wyjątków (Listing 8), która po wykonaniu spowoduje powrót do miejsca oznaczonego jako `end`. W ten sposób nie zostanie wykonana linijka `mov IceBreakValue, 1`. W przypadku, gdy proces działa pod debuggerem, wykonanie programu zostanie zatrzymane na powyższej linijce.

- *Memory breakpoint*

Pamięciowe *breakpointy* używane są przez debuggery do sprawdzania, czy proces odwołuje się do jakiegoś miejsca w pamięci. W tym celu wykorzystywana jest flaga `PAGE_GUARD` ustawiana przy danym fragmencie. Gdy następuje odwołanie do takiej pamięci, generowany jest wyjątek `STATUS_GUARD_PAGE_VIOLATION`. Działanie kodu sprawdzającego istnieje debuggera jest następujące. Utworzony zostaje fragment pamięci z ustawioną flagą `PAGE_GUARD` do której zapisywany zostaje kod funkcji powrotu `RET` (`0xC3`). Następnie zostaje wykonany funkcji powrotu.

Jeżeli to się uda, funkcja `RET` wykona skok do pamięci odłożonej na stosie (w tym wypadku do miejsca oznaczonego jako `MemBreakDbg`). Oznacza to, iż debugger obsłużył wyjątek i kontynuował działanie

programu. Brak debuggera powoduje wystąpienie wyjątku i wykonania fragmentu odpowiedzialnego za obsługę wyjątku.

- Sprzętowe *breakpointy*

To specjalny mechanizm zaimplementowany przez intela. Do jego kontroli wykorzystuje się stworzony do tego celu zestaw rejestrów oznaczonych jako `Dr0` – `Dr7`. Jednak dostęp do nich jest zabroniony poprzez użycie instrukcji `mov`. Aby to ominąć stosuje pewien trick. Należy spowodować wystąpienie wyjątku. Kontekst procesu wraz z wartościami tych rejestrów zostanie udostępniony funkcji obsługującej wyjątek. Listing 12. prezentuje w jaki sposób ustawić taką funkcję oraz spowodować wystąpienie wyjątku. Uzyskuję się to poprzez dzielenia przez zero. W funkcji obsługującej wyjątek można sprawdzić lub ustawić wartości tych rejestrów. Rejestry `Dr0` – `Dr3` przechowują adresy, w których zostały ustawione breakpointy. `Dr4` oraz `Dr5` są zarezerwowane przez Intela do debugowania innych rejestrów, natomiast pozostałe dwa, `Dr6` i `Dr7`, służą do kontroli zachowania się breakpointów. Jeżeli wartość któregoś z pierwszych czterech rejestrów jest różna od 0 to oznacza, iż zostały ustawione breakpointy. Na Listingu 13 przedstawiona jest funkcja, która sprawdza zawartość rejestrów.

Metody wykorzystujące środowisko procesów oraz zarządzanie tymi procesami

Metody te oparte są o mechanizmy systemu służące do zarządzania środowiskiem procesu. Również ono może zdradzać obecność debuggera

- *Parent Process*

To metoda wykorzystuje identyfikator procesu nadrzędnego. Jeżeli program uruchamiany jest bez debuggera, to jego nadrzędnym procesem jest *explorer.exe*. Jeżeli został uruchomiony przez debugger, to on jest wtedy procesem nadrzędnym. Listing 14. przedstawia funkcję sprawdzającą proces nadrzędny. Najpierw pobierany jest `PID` (ang. *Process IDentifier*) procesu `explorer`. Następnie pobieramy `PID` naszego procesu. W celu pobrania `PID` procesu nadrzędnego potrzeba jest nieco więcej wysiłku.

Najpierw robimy `SnapShot` wszystkich procesów systemu, a następnie wyszukujemy struktury opisującej nasz proces. Po jej znalezieniu odczytujemy `PID` procesu nadrzędnego.

- Open Process

Ta metoda opiera się na wykorzystaniu błędnie ustawionych przywilejów dla debugowanego procesu. Jeżeli proces zostanie podłączony do debuggera, a jego przywileje nie zostaną odpowiednio zmienione uzyska on przywilej o nazwie `SeDebugPrivilige`. Pozwoli to na otwarcie dowolnego procesu w systemie. Przykładem takiego procesu jest *csrss.exe*, do którego normalnie nie ma dostępu. Aby sprawdzić czy proces jest podłączony do debuggera należy

Listing 13. *Funkcja obsługująca wyjątek, która sprawdza zawartość rejestrów Dr0 – Dr3*

```
EXCEPTION_DISPOSITION __cdecl
      hardbreakhandler(struct _EXCEPTION_RECORD *ExceptionRecord, void *
                       EstablisherFrame,
      struct _CONTEXT *ContextRecord, void * DispatcherContext )
{
   if(ContextRecord->Dr0 || ContextRecord->Dr1 || ContextRecord->Dr2 ||
                       ContextRecord->Dr3)
   {
      cout << "  -  Debugger odnaleziony\n";
   }
   else
   {
      cout << "  -  Nie odnaleziono debuggera\n";
   }
   ContextRecord->Eip = *(((DWORD *)EstablisherFrame)+2);
   ContextRecord->Ebp = *(((DWORD *)EstablisherFrame)+3);
   return ExceptionContinueExecution;
}
```

Listing 14. *Funkcja porównująca PID procesu nadrzędnego oraz procesu explorer.exe*

```
bool ParentProcessTest()
{
   DWORD ExplorerPID = 0;
   GetWindowThreadProcessId(GetShellWindow(), &ExplorerPID);
   DWORD CurrentPID = GetCurrentProcessId();
   DWORD ParentPID = 0;
   HANDLE SnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
   PROCESSENTRY32 pe = { 0 };
   pe.dwSize = sizeof(PROCESSENTRY32);
   if(Process32First(SnapShot, &pe))
   {
      do
      {
         if(CurrentPID == pe.th32ProcessID)
            ParentPID = pe.th32ParentProcessID;
      }while( Process32Next(SnapShot, &pe));
   }
   CloseHandle(SnapShot);
   if(ExplorerPID == ParentPID)
      return false;
   else
      return true;
}
```

otworzyć proces *csrss.exe* i sprawdzić wynik takiej operacji. Listing 15. przedstawia funkcję, która używa tej metody do sprawdzenia czy debugger jest podłączony.

· *Self-Debugging*

Metoda ta polega na stworzeniu procesu potomnego, który za pomocą metody `DebugActiveProcess` spróbuje się podłączyć do swojego procesu nadrzędnego czyli naszego głównego programu. Jeżeli mu się to nie uda, oznacza, że jakiś debugger jest już podpięty. Schemat działania takiego programu wygląda następująco. Gdy mamy jedną funkcję dla procesu nadrzędnego i potomnego, najpierw należy je rozróżnić. W tym celu posłużymy się nazwanym *muteksem*. Oba procesy wykonują funkcję `CreateMutex`. Dla procesu nadrzędnego *muteks* zostanie poprawnie utworzony, natomiast dla procesu potomnego zostanie zwrócony błąd `ERROR_ALREADY_EXISTS`. Listing 16. przedstawia fragment kodu odpowiedzialnego za rozróżnienie procesów.

Zadaniem procesu potomnego jest próba podłączenia się jako debugger. W tym celu wyszukuje on swój proces nadrzędny i podłącza się do niego za pomocą wcześniej wspomnianej funkcji `DebugActiveProcess`. Jeżeli uda się należy najpierw rozłączyć się z procesem nadrzędnym (jeżeli nie nastąpi rozłączenie to po wyjściu z procesu potomnego proces nadrzędny również zostanie zakończony) za pomocą funkcji `DebugActiveProcessStop`. W zależności od rezultatu proces kończy się z odpowiednim kodem. Listing 17. przedstawia kod działań opisanych powyżej. Występująca tu funkcja `GetParentPID` jest abstrakcyjna, zwracającą `PID` procesu nadrzędnego. Fragment kodu, który można wykorzystać do jej implementacji został przedstawiony w poprzedniej metodzie.

Proces nadrzędny natomiast oczekuje na wartość zwróconą przez proces potomny. To od niej zależy czy debugger jest podłączony czy nie. Listing 18. zawiera kod dla procesu nadrzędnego.

· *UnhandledExceptionFilter*

`UnhandleExceptionFilter` to funkcja wywoływana przez system w momencie gdy wystąpił wyjątek i nie istnieje żadna funkcja go

Listing 15. *Funkcja sprawdzająca obecność debuggera poprzez próbę dostępu do procesu csrss.exe*

```
bool OpenProcessTest()
{
   HANDLE csrss = 0;

   PROCESSENTRY32 pe = { 0 };
   pe.dwSize =  sizeof(PROCESSENTRY32);
   HANDLE SnapShot = NULL;
   DWORD csrssPID = 0;
   wchar_t csrssName [] = TEXT("csrss.exe");
   SnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
   if(Process32First(SnapShot, &pe))
   {
      do
      {

         if(wcscmp(pe.szExeFile, csrssName) == 0)
         {
            csrssPID = pe.th32ProcessID;
            break;
         }
      }while(Process32Next(SnapShot, &pe));
   }
   CloseHandle(SnapShot);
   csrss = OpenProcess(PROCESS_ALL_ACCESS, FALSE, csrssPID);
   if (csrss != NULL)
   {
      CloseHandle(csrss);
      return true;
   }
   else
      return false;
}
```

Listing 16. *Fragment kodu służący do rozróżniania procesów*

```
WCHAR *MutexName = TEXT("SelfDebugMutex");
HANDLE  MutexHandle = CreateMutex(NULL, TRUE, MutexName);
if(GetLastError() == ERROR_ALREADY_EXISTS)
{
   ... /// Kod procesu potomnego
}
else
{
   ... /// Kod procesu nadrzędnego
}
```

Listing 17. *Fragment kodu procesu potomnego*

```
DWORD ParentPID = GetProcessParentID(GetCurrentProcessId());
if(DebugActiveProcess(ParentPID))
{
   DebugActiveProcessStop(ParentPID);
   exit(0);
}
else
{
   exit(1);
}
```

obsługująca. Zadaniem tej funkcji jest decyzja co należy zrobić z procesem. Jeżeli proces nie jest debugowany zostanie wywołana ostateczna funkcja obsługująca wszystkie wyjątki (jeżeli taka została ustawiona). Słabość tej metody uwidacznia się w przypadku wykrycia debugger. W takim wypadku proces zostaje zakończony co uniemożliwia jego analizę przez debugger. Listing 19. przedstawia fragment kodu, który ustawia funkcję obsługującą wyjątki oraz generuje wyjątek (poprzez dzielenie przez 0). Różnica między tą metodą ustawiania obsługi wyjątku, a tymi zaprezentowanymi poprzednio polega na tym, że poprzednio nasza funkcja była pierwsza w łańcuchu poszukiwań, a tutaj jest ostatnia. Listing 20. zawiera zaś kod funkcji służącej do obsługi tego wyjątku.

· *NtQueryObject*

Funkcja ta służy do pobierania informacji na temat różnych obiektów w systemie Windows. Na oficjalnej stronie opisane są jedynie niektóre opcje, dlatego polecam do zapoznania się z nieudokumentowanymi właściwościami tej funkcji. Użycie parametru `ObjectAllTypesInformation` (wartość `0x03`) powoduje zwrócenie szczegółowych informacji na temat wszystkich obiektów. Podczas procesu debugowania tworzone są tak zwane `DebugObject`. Należy użyć funkcji `NtQueryObject` i sprawdzić ile obiektów `DebugObject` jest w systemie. Jeżeli więcej niż 0 oznacza, że uruchomiony jest debugger. W metodzie tej nawet jeżeli pod debuggerem będzie uruchomiony inny proces, to i tak zostanie to wykryte. Informacje zwracane przez funkcje znajdują się w buforze w następującej kolejności: najpierw znajduje się `OBJECT_ALL_INFORMATION` zawierająca liczbę wszystkich zwróconych struktur. Zaraz za nią znajduje się tablica zawierająca tablice znaków Unicode, na którą wskazuje `OBJECT_TYPE_INFORMATION->TypeName`. Po wyrównaniu pamięci do 4 bajtów umieszczany jest kolejny obiekt typu `OBJECT_ALL_INFORMATION`. Ponieważ definicje tych obiektów oraz funkcji nie znajdują się w bibliotekach nagłówkowych, należy zdefiniować je samemu. Listing 21. przedstawia te definicje oraz kod źródłowy funkcji, która wykorzystując `NtQueryObject` sprawdza obecność debuggera

· *DebugObject Handle*

Metoda ta zbliżona jest do poprzedniej. Również wykorzystujemy wiedzę o tym, iż tworzone są `DebugObject` podczas procesu debugowania. W tej metodzie jednak nie pobieramy wszystkich obiektów, a jedynie uchwyt do tego obiektu. Wykorzystana do tego zostanie funkcja `NtQueryInformationProcess`. Podobnie jak w poprzednim przypadku, również nie istnieje jej deklaracja w plikach nagłówkowych, dlatego też jej adres należy pobrać z pliku *ntdll.dll*. Jeżeli uchwyt będzie miał wartość NULL to znaczy, że proces nie jest debugowany. Listing 22. zawiera kod źródłowy funkcji.

· *OutputDebugString*

Bardzo prosta metoda polegająca na wysłaniu ciągu znakowego do debuggera Jeżeli proces jest debugowany wtedy funkcja zakończy się sukcesem, jeśli nie zostanie zwrócony kod błędu. Listing 23. pokazuje w jaki sposób to wykorzystać.

· Wyszukiwanie okien debuggerów

Metoda raczej mało uniwersalna ale również potrafiąca znaleźć uruchomiony debugger Jej działanie jest proste. Za pomocą funkcji `FindWindow` wyszukujemy interesujących nas

Listing 18. *Fragment kodu procesu nadrzędnego*

```
PROCESS_INFORMATION pi;
STARTUPINFO si;
DWORD ExitCode = 0;
ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
ZeroMemory(&si, sizeof(STARTUPINFO));
GetStartupInfo(&si);
// Utworzenie procesu potomnego
CreateProcess(NULL, GetCommandLine(), NULL, NULL, FALSE, NULL, NULL, NULL, &si,
                        &pi);
WaitForSingleObject(pi.hProcess, INFINITE);
GetExitCodeProcess(pi.hProcess, &ExitCode);
if(ExitCode)
{
   cout << "  -  Debugger odnaleziony\n";
}
else
{
   cout << "  -  Nie odnaleziono debuggera\n";
}
```

Listing 19. *Fragment kodu ustawiającego funkcję obsługującą wyjątek*

```
SetUnhandledExceptionFilter(UnhandledExcepFilterHandler);
__asm
{
   xor eax, eax
   div eax
```

Listing 20. *Funkcja obsługująca wyjątki*

```
LONG WINAPI UnhandledExcepFilterHandler(PEXCEPTION_POINTERS pExcepPointers)
{
   SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)
      pExcepPointers->ContextRecord->Eax);
   pExcepPointers->ContextRecord->Eip += 2;
   return EXCEPTION_CONTINUE_EXECUTION;
}
```

Listing 21. *Definicje struktur oraz funkcji korzystającej z NtQueryObject*

```
typedef struct _OBJECT_TYPE_INFORMATION {
   UNICODE_STRING TypeName;
   ULONG TotalNumberOfHandles;

   ULONG TotalNumberOfObjects;
   ULONG  Reserved[20];

} OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;
typedef struct _OBJECT_ALL_INFORMATION {
   ULONG NumberOfObjects;

   OBJECT_TYPE_INFORMATION ObjectTypeInformation[1];
}OBJECT_ALL_INFORMATION, *POBJECT_ALL_INFORMATION;

#define ObjectAllInformation 3
int NtQueryObjectTest()
{
   typedef NTSTATUS(NTAPI *pNtQueryObject)(HANDLE, UINT, PVOID, ULONG, PULONG);
   POBJECT_ALL_INFORMATION pObjectAllInfo = NULL;
   void *pMemory = NULL;
   NTSTATUS Status;
   unsigned long Size = 0;
   pNtQueryObject NtQueryObject = (pNtQueryObject)GetProcAddress(
         GetModuleHandle(TEXT( "ntdll.dll" )),"NtQueryObject");

   // Pobranie ilości pamięci potrzebnej do otrzymania wszystkich obiektów
   Status = NtQueryObject(NULL, ObjectAllInformation, &Size, 4, &Size);
   // Alokacja pamięci na obiekty
   pMemory = VirtualAlloc(NULL, Size, MEM_RESERVE | MEM_COMMIT,PAGE_READWRITE);
   if(pMemory == NULL)
      return false;
   // Pobranie listy obiektów
   Status = NtQueryObject((HANDLE)-1, ObjectAllInformation, pMemory, Size, NULL);
   if (Status != 0x00000000)
   {
      VirtualFree(pMemory, 0, MEM_RELEASE);
      return false;
   }
   pObjectAllInfo = (POBJECT_ALL_INFORMATION)pMemory;
   ULONG NumObjects = pObjectAllInfo->NumberOfObjects;
   POBJECT_TYPE_INFORMATION pObjectTypeInfo = (POBJECT_TYPE_INFORMATION)
         pObjectAllInfo->ObjectTypeInformation;
   unsigned char *tmp;
   for(UINT i = 0; i < NumObjects; i++)
   {
      pObjectTypeInfo = (POBJECT_TYPE_INFORMATION)pObjectAllInfo->ObjectTypeInformation;
      if (wcscmp(L"DebugObject", pObjectTypeInfo->TypeName.Buffer) == 0)
      {
         if (pObjectTypeInfo->TotalNumberOfObjects > 0)
         {
            VirtualFree(pMemory, 0, MEM_RELEASE);
            return true;
         }
         else
         {
            VirtualFree(pMemory, 0, MEM_RELEASE);
            return false;
         }
      }
      tmp = (unsigned char*)pObjectTypeInfo->TypeName.Buffer;
      tmp += pObjectTypeInfo->TypeName.Length;
      pObjectAllInfo = (POBJECT_ALL_INFORMATION)(((ULONG)tmp) & -4);
   }
   VirtualFree(pMemory, 0, MEM_RELEASE);
   return true;
}
```

debuggerów. Funkcja zwraca uchwyt do takiego okna lub NULL, jeśli okno nie istnieje. Funkcja z Listingu 23. wyszukuje okien Ida PRO, OllyDbg oraz WinDbg.

Metody wykorzystujące czas

Ostatnia grupa metod wykorzystuje czas. Wadą tej metody jest to, iż nie sprawdza ona czy istnieje debugger, a jedynie czy nastąpiło jakieś zatrzymanie wykonywania programu pomiędzy miejscami uruchomienia funkcji pobierającej czas. Wykorzystywane są tutaj następujące funkcję:

- RDTSC

Funkcja procesorów Intel zwracająca ilość cykli zegara od momentu resetu procesora. Wartość ta jest 64 bitowa ,dlatego stanowi dobry miernik czasu.

- Funkcje API

Są to funkcje systemowe systemu Windows. Pierwsza z nich to `GetTickCount`. Zwraca ona ilość milisekund jakie minęły od czasu, kiedy wystartował system. Maksymalnie może to być 49,7 dnia. Funkcja ta może być zastąpiona `timeGetTime`, która zwraca taką samą informację. Można również wykorzystać funkcję `QueryPerformanceCounter`.

Wymienione powyżej funkcję nie są jedynymi dostępnymi, które nadają się do tego celu. Na stronie MSDN można znaleźć wiele innych, które również będą bardzo dobrze działały.

Podsumowanie

Nowoczesne procesory oraz system Windows daje nam wiele możliwość sprawdzenia czy nasz proces podlega debugowaniu. Warto pamiętać, że przedstawione metody mają najprostszą postać, aby lepiej można było się z nimi zapoznać. W praktyce powyższe implementacje mogą być o wiele bardziej skomplikowane, aby utrudnić ich wykrycie. Również łączone są z metodami zabezpieczania kodu ale to już zupełnie inna sprawa.

Marek Zmysłowski
Autor jest absolwentem Politechniki Warszawskiej. Obecnie pracuje jako audytor. Programista C oraz C++. Interesuje się ogólnie pojętym bezpieczeństwem w Internecie. W szczególnym kręgu zainteresowań autora znajduje się inżynieria wsteczna (ang. *reverse engineering*).
Kontakt z autorem: *marekzmyslowski@poczta.onet.pl*

Listing 22. *Funkcja pobierająca uchwyt do DebugObject*

```
int DebugObjectHandleTest()
{
   typedef NTSTATUS (WINAPI *pNtQueryInformationProcess)
         (HANDLE ,UINT ,PVOID ,ULONG , PULONG);
   HANDLE hDebugObject = NULL;
   NTSTATUS Status;
   pNtQueryInformationProcess NtQueryInformationProcess = (pNtQueryInformationProcess)

         GetProcAddress(  GetModuleHandle( TEXT("ntdll.dll") ), "NtQueryInformationProcess" );
   Status = NtQueryInformationProcess(GetCurrentProcess(),0x1e, &hDebugObject, 4, NULL);
   if (Status != 0x00000000)
      return -1;
   if(hDebugObject)
      return 1;
   else
      return 0;
}
```

Listing 23. *Funkcja wykorzystująca OutputDebugString*

```
bool OutputDebugStringTest()
{
   OutputDebugString(TEXT("DebugString"));
   if (GetLastError() == 0)
      return true;
   else
      return false;
}
```

Listing 24. *Funkcja wyszukująca okien debuggerów wykorzystując ich nazwy*

```
bool FindDebuggerWindowTest()
{
   HANDLE hOlly = FindWindow(TEXT("OLLYDBG"), NULL);
   HANDLE hWinDbg = FindWindow(TEXT("WinDbgFrameClass"), NULL);
   HANDLE hIdaPro = FindWindow(TEXT("TIdaWindow"), NULL);
   if(hOlly || hWinDbg || hIdaPro)
      return true;
   else
      return false;
}
```