Bezpieczne programowanie. Aplikacje hakeroodporne

Autor: Jacek Ross
ISBN: 978-83-246-2405-8
Format: 158×235, stron: 312

Wyobraź sobie sytuację, w której poświęcasz mnóstwo czasu na stworzenie nowego, ciekawego rozwiązania w świecie informatyki. Kosztuje Cię to wiele dni i nocy ogromnego wysiłku. Dokładnie w momencie opuszczenia Twojego bezpiecznego komputera, udostępniony światu, Twój pomysł zostaje wystawiony na ciężką próbę – w sieci działają krakerzy, którzy za wszelką cenę będę próbowali złamać Twoje zabezpieczenia lub wykorzystać luki w Twojej aplikacji. Jak tego uniknąć? Jak tworzyć oprogramowanie odporne na ich ataki?

Proste i przejrzyste odpowiedzi na podobnie skomplikowane pytania znajdziesz właśnie w tej książce! Podczas lektury poznasz zagrożenia, na jakie narażony jest programista, oraz proste sposoby utrudniania krakerom zadania. Dodatkowo zdobędziesz wiedzę na temat metod szyfrowania danych i wyznaczania sygnatur. Jednak, co najważniejsze, zobaczysz, jak wykorzystać tę wiedzę w praktyce! W publikacji „Bezpieczne programowanie. Aplikacje hakeroodporne" znajdziesz również sporo ciekawych informacji na temat zabezpieczania aplikacji sieciowych oraz zaawansowane metody, gwarantujące podniesienie bezpieczeństwa Twojego produktu do wysokiego poziomu. To jeszcze nie wszystko! W kolejnych rozdziałach autor prezentuje sposoby ochrony przed debugerami, patenty na bezpieczne tworzenie kodu na platformie .NET oraz psychologiczne aspekty tworzenia hakeroodpornych aplikacji!

- Przegląd zagrożeń, rodzaje oszustw i naruszeń bezpieczeństwa
- Zabezpieczenie programu przy użyciu numeru seryjnego
- Dostępne na rynku systemy zabezpieczania aplikacji
- Algorytmy szyfrujące
- Tworzenie skrótów wiadomości
- Wykorzystanie szyfrowania przy zabezpieczaniu oprogramowania
- Zabezpieczenia aplikacji wykorzystujących PHP i .NET
- Ochrona przed atakami typu: wstrzykiwanie SQL, XSS, DOS i DDOS
- Używanie zaawansowanych metod ochrony oprogramowania
- Sposoby zaciemniania programu
- Ochrona kodu przed debugerami
- Zastosowanie kluczy sprzętowych i technik biometrycznych
- Psychologiczne aspekty ochrony oprogramowania

Dowiedz się, jak tworzyć aplikacje odporne na ataki!

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 6.

Zabezpieczenie programów przed debugerami

6.1. Wstęp

Wykrywanie aktywnych debugerów oraz programów zrzucających fragment pamięci typu `ProcDump` to czynność, która może być opłacalna, jednak nie jest tak prosta, jak się wydaje. Istnieje wiele wyrafinowanych sztuczek wykrywających te programy, jednak zdecydowana większość z nich wykrywa dokładnie określone programy, jak na przykład SoftIce czy OllyDbg, i w dodatku często tylko określone ich wersje. Najgorszą informacją jest jednak to, że istnieje mnóstwo łatek i rozszerzeń popularnych debugerów, które zabezpieczają się przed wykryciem. Programy te po modyfikacjach (wykonanych często przez samych krakerów dla siebie) ukrywają się i stosują techniki mylące. Dlatego wykrywanie debugerów jest zadaniem często skazanym na porażkę. Zwykle jest się pół kroku do tyłu, a nawet jeśli uda nam się poprzez zaimplementowanie wielu technik jednocześnie wykryć skutecznie wszystkie istniejące debugery, to i tak jest to tylko chwilowe zwycięstwo, bo za chwilę powstaną takie, których nie wykryjemy. Jest to więc typowe leczenie objawów, a nie przyczyn choroby.

Nie oznacza to, że nie należy w ogóle przejmować się problemem i nie stosować żadnych zabezpieczeń. Jednak jeśli Czytelnik nie jest geniuszem assemblera, powinien zostawić to zadanie innym i użyć do tego celu programu pobranego z internetu. Jest ich wiele, zarówno darmowych, jak i płatnych. Do tych drugich należą m.in. opisane w rozdziale drugim TheMida czy AsProtect. Dzięki temu zaoszczędzimy sobie czasu na próby, których wyniki mogą być wątpliwe, a co więcej, których rezultat ciężko będzie nam zweryfikować, ponieważ nie będziemy dysponować odpowiednią ilością wersji debugerów. Procedury kontrolujące działanie takich programów nie muszą być niestandardowe i dlatego warto pozostawić je fachowcom.

Z tych właśnie powodów w rozdziale tym nie znajdziecie wielu wyrafinowanych sztuczek blokujących debugery. Opiszę jedynie kilka najprostszych, aby każdy mógł sobie wyrobić opinię na ten temat i w razie potrzeby poszukać bardziej zaawansowanych informacji w innych źródłach. Zamiast tego skupię się na ogólnych metodach utrudniania debugowania, które mogą być użyteczne niezależnie od stosowania funkcji wykrywających debugery lub komercyjnych systemów zabezpieczeń, które również takowe funkcje posiadają.

6.2. Wykrywanie debugerów

Najprostszym sposobem wykrywania debugerów jest wstawienie do kodu po prostu:

```
if(IsDebuggerPresent())
        // wykonaj jakiś kod, np. zamknij program;
```

Można to łatwo przetestować, wpisując w miejsce komentarza na przykład polecenie wyświetlenie jakiegoś komunikatu i uruchamiając aplikację krok po kroku w Visual Studio. Debuger VS zostanie wykryty i program wyświetli komunikat. Uruchamiając program z pliku wykonywalnego, nie zobaczymy komunikatu.

Tego typu zabezpieczenie będzie jednak całkowicie nieskuteczne w przypadku zaawansowanych debugerów używanych przez krakerów, takich jak SoftIce czy OllyDbg. Włamywacze używają zmodyfikowanych wersji tych programów, które są zabezpieczone nie tylko przed opisaną powyżej weryfikacją, ale także przed wieloma bardziej zaawansowanymi metodami. Twój program musiałby mieć naprawdę szeroką gamę testów, aby można było mieć pewność wykrycia większości mutacji tych debugerów. Dlatego właśnie lepiej jest użyć gotowego narzędzia wykonanego przez specjalistów, a samemu skupić się na innych zabezpieczeniach. Jako ciekawostkę podam jeszcze przykłady dwóch rozwiązań tego typu:

```
mov ah,43h
int 68h
cmp eax,0F386h
jz mamy_debugger
```

W tym teście próbujemy zbadać, czy uruchomiony został program SoftIce poprzez wykrywanie sterownika debugera systemowego. Kolejny przykład opiera się na wykrywaniu `int 41h` używanego przez debugery:

```
xor ax,ax
mov es,ax
mov bx, cs
lea dx, int41handler
xchg dx, es:[41h*4]
xchg bx, es:[41h*4+2]
in al, 40h
xor cx,cx
int 41h
xchg dx, es:[41h*4]
xchg bx, es:[41h*4+2]
```

```
cmp cl,al
jnz si_jest

int41handler PROC
mov cl,al
iret
int41handler ENDP
```

Obydwa powyższe przykłady wykrywania debugera SoftIce pochodzą ze strony: *http://4programmers.net/Assembler/FAQ/Wykrywanie_debuggera_(SoftICE)*[1].

6.3. Utrudnianie debugowania

6.3.1. Wstawki kodu utrudniające debugowanie

Utrudnienie debugowania można wykonać, wklejając w wiele miejsc kodu źródłowego programu relatywnie nieskomplikowane wstawki. Jeśli używamy języka C/C++, sprawa jest prosta, możemy bowiem wykorzystać makra preprocesora, co w połączeniu z twórczym użyciem instrukcji skoku bezwarunkowego (tak oczerniane przez wszystkich `goto`), wstawek assemblerowych czy konstrukcji takich jak `setjmp` czy `longjmp`, może solidnie skomplikować śledzenie programu w debugerze. Ważne jest, aby tego typu wstawki pozostawały neutralne dla kodu.

Oto przykład:

```
jmp_buf env;
#define AD(kod, numer) if( setjmp(env) == 0 )      \
{                                                  \
      goto skok_ad_##numer;                        \
}                                                        \
else                                                     \
{                                                        \
      kod;                                               \
}                                                        \
goto koniec_ad_##numer;                            \
skok_ad_##numer:                                         \
      longjmp(env, 1);                             \
koniec_ad_##numer:                                       \
;
```

Powyższe makro wstawia kod wyglądający mniej więcej tak:

```
if( setjmp(env) == 0 )
{
      goto skok_ad_1;
}
else
```

[1] Treść udostępniona na zasadach licencji Creative Commons Attribution: *http://creativecommons.org/licenses/by/2.0/pl/legalcode*.

```
{
      // WŁAŚCIWY FRAGMENT KODU
}
goto koniec_ad_1;
skok_ad_1:
      longjmp(env, 1);
koniec_ad_1:
```

W trakcie wykonania instrukcja `setjmp(env) == 0` wywołana po raz pierwszy ustawi punkt powrotu dla skoku `longjmp` na miejsce swojego wywołania oraz zwróci wartość `TRUE`. W związku z tym kolejną wykonywaną instrukcją będzie `goto skok_ad_1`. Następnie wykona się `longjmp(env,1)`, które spowoduje przesunięcie wykonania z powrotem do pierwszej linii kodu. Tym razem jednak warunek nie będzie już spełniony i wykonany zostanie ciąg instrukcji oznaczony jako `WŁAŚCIWY FRAGMENT KODU`. Na koniec instrukcja `goto koniec_ad_1` zapewni prawidłowe opuszczenie tego bloku kodu. Cała konstrukcja ma wpleść w kod kilka dziwnych i zaciemniających sens skoków wydłużających i utrudniających śledzenie wykonania w debugerze. Oczywiście powinna być użyta w programie wiele razy, aby być naprawdę skuteczną.

Użycie makra jest następujące. Kod, który zaciemniamy, otacza się wywołaniem makra: `AD(kod, x)`. Pewną wadą tego makra jest to, że trzeba go wywoływać z dodatkowym parametrem, który za każdym razem musi być inny. To, co przed zaciemnieniem wygląda tak:

```
for( int i = 0; i < 10; i++)
for( int j = 0; j < i; j++)
      if( x[i] < x[j])
{
               int tmp = x[i];
               x[i] = x[j];
               x[j] = tmp;
      }
```

po — będzie wyglądać tak:

```
      AD(
      for( int i = 0; i < 10; i++)
               for( int j = 0; j < i; j++)
                        AD(if( x[i] < x[j])
                        {
                                 int tmp = x[i];
                                 AD(x[i] = x[j] ,2);
                                 AD(x[j] = tmp  ,3);
                        }
                        , 1)
      , 0)
```

Inne makra mogą korzystać wprost ze wstawek assemblerowych:

```
#define WSTAWKA_START_ASM __asm PUSH EAX __asm  PUSH EBX __asm  PUSH ECX __asm PUSHF

#define WSTAWKA_STOP_ASM __asm POPF __asm POP ECX __asm POP EBX __asm POP EAX

#define WSTAWKA_PUSTA_1 WSTAWKA_START_ASM __asm MOV EAX, ECX __asm MOV EAX, 0 __asm
↪ADD EAX, 0xD007 _asm MOV EBX, EAX WSTAWKA_STOP_ASM
```

```
#define WSTAWKA_PUSTA_2 WSTAWKA_START_ASM MAKE_DUMMY_1 __asm TEST EAX, EAX __asm
↪PUSHF __asm POP EAX __asm MOV EAC, EAX MAKE_STOP_ASM

#define WSTAWKA_PUSTA_3 WSTAWKA_START_ASM __asm MOV EAX, EBX __asm ADD EAX, 0x01ac
__asm PUSH EAX __asm MOV EAX, EBX __asm POP EBX WSTAWKA_STOP_ASM
```

Powyższe kody nie mają żadnego sensu, natomiast nie zmieniają stanu rejestrów ani flag (dzięki użyciu makr `WSTAWKA_START_ASM` i `WSTAWKA_STOP_ASM`). Użycie jest proste: wywołanie makra wstawia się po prostu do kodu.

```
for( int i = 0; i < 10; i++)
for( int j = 0; j < i; j++)
      if( x[i] < x[j])
{
                        WSTAWKA_PUSTA_3
                        int tmp = x[i];
                        WSTAWKA_PUSTA_3 WSTAWKA_PUSTA_2
                        x[i] = x[j];
                        WSTAWKA_PUSTA_3 WSTAWKA_PUSTA_1 WSTAWKA_PUSTA_3
                        x[j] = tmp;
                WSTAWKA_PUSTA_2
      }
```

Prawda, że proste? Polecam obejrzenie kodu assemblerowego wygenerowanego przez kompilator dla każdej z trzech powyższych wersji kodu. Poświęciwszy 3 – 4 godziny, można wyprodukować kilkanaście czy nawet kilkadziesiąt podobnych fragmentów kodu i użyć ich w setkach miejsc, wykonując po prostu pracę typu kopiuj/wklej. Nie powstrzyma to samo w sobie żadnego krakera, ale utrudni mu nieco pracę. Oczywiście takie wstawki wydłużają kod oraz nieznacznie spowalniają jego wykonanie. Jeśli jednak będziemy używać ich wyłącznie w najbardziej krytycznych fragmentach programu, dotyczących zabezpieczeń, to jest to opłacalne. Zwłaszcza że przy dzisiejszych prędkościach procesorów tych kilkadziesiąt instrukcji maszynowych nie będzie dużym obciążeniem.

Powyższe makra są banalne i mają wyłącznie posłużyć jako przykład tego, co można robić. W prawdziwym systemie warto posiedzieć nieco dłużej i dodać następujące elementy:

- Wstawki assemblerowe z dużą ilością skoków wymieszanych bez ładu, na przykład:

```
WSTAWKA_START_ASM
  __asm JMP skok_1x;
skok_1x: __asm TEST EAX,EBX;
  __asm JMP skok_3x;
skok_5x:
  __asm MOV EAX, EBX;
  __asm JMP skok_4x;
skok_2x: __asm MOV EAX, ECX;
  __asm JMP skok_5x;
skok_4x: __asm POPF;
  __asm JMP skok_6x;
skok_3x: __asm MOV EAX, 0;
  __asm PUSH EAX;
  __asm MOV EAX, EBX;
```

```
  __asm TEST EAX, EAX;
  __asm JNZ skok_4x;
  __asm JMP skok_2x;
skok_6x: __asm MOV EAX,EBX;
WSTAWKA_STOP_ASM
```

- Użycie nie tylko skoków bezwzględnych, ale także dużej ilości zagnieżdżonych funkcji oraz skoków warunkowych (jak w powyższym przykładzie skok `JNZ`).
- Wstawianie instrukcji, które nie będą nigdy wykonane, w pobliże właściwego kodu. Można tego dokonać poprzez skoki. Ciekawą instrukcją, którą można w takie miejsce wstawić, jest `__asm int 3`, czyli instrukcja przerwania programowego.
- Można nawet pokusić się o sterowanie przepływem wykonania za pomocą wyjątków.

Przez cały czas powinniśmy mieć jednak na uwadze to, że jest to tylko dodatek, i uważać, aby nie zaciemnić sobie kodu nadmiernie.

6.3.2. Generator wstawek kodu utrudniających debugowanie

Makra, o których pisałem powyżej, to interesująca technika, ale mają one kilka wad:

- Nie każdy język posiada makra preprocesora. Można jeszcze ratować się użyciem funkcji rozwijanych w miejscu wywołania (`inline`), ale nie zawsze i nie wszystko uda się nimi zastąpić.
- Makra pisane ręcznie są zazwyczaj schematyczne. Ciężko je sparametryzować czy zmieniać im dynamicznie treść.
- Nawet najwygodniejsze w użyciu widnieją w kodzie źródłowym, zmniejszając jego czytelność oraz utrudniając testowanie (i debugowanie, ale w końcu po to zostały napisane).

Lepszym rozwiązaniem będzie więc zostawienie wersji deweloperskiej kodu w stanie czystym i wstawianie kodu antydebugowego przez automat dopiero w procesie produkcji oprogramowania. Dzięki użyciu zautomatyzowanych systemów produkcji oprogramowania takich jak na przykład CruiseControl .NET, systemów kontroli wersji takich jak CVS czy SVN oraz narzędzi skryptowych takich jak NANT, Python czy Ruby proces ten nie powinien być skomplikowany. Wygląda on następująco:

- Prace deweloperskie wykonywane są przez programistów. Programista, kończąc prace, wysyła kod źródłowy do repozytorium systemu kontroli wersji.
- Automat pobiera zmodyfikowane źródła do swojej lokalnej kopii.
- Prekompilacja — wygenerowanie przez automat wstawek antydebugowych i wstawienie ich do kodu.
- Kompilacja zmodyfikowanego kodu.
- Wykonanie testów automatycznych.

Automat taki może generować wstawki losowo spośród zbioru elementów w miejsca oznaczone w kodzie w specyficzny sposób. Dzięki temu jedyne, co musi zrobić programista, to wstawić takie oznaczenia wszędzie, gdzie chce. Może to być na przykład specyficzny komentarz:

```
// ZABEZPIECZ
for( int i = 0; i < 10; i++)
for( int j = 0; j < i; j++)
      if( x[i] < x[j])
{
// ZABEZPIECZ
               int tmp = x[i];
// ZABEZPIECZ
               x[i] = x[j];
// ZABEZPIECZ
               x[j] = tmp;
// ZABEZPIECZ
      }
// ZABEZPIECZ
```

Na płycie CD znajduje się kod źródłowy programu GenerujZabezpieczenia napisanego w języku C#. Jest to program uruchamiany z linii poleceń, którego jedynym parametrem jest ścieżka do pliku źródłowego (w języku C++), w którym wstawki `// ZABEZPIECZ` mają zostać zmienione na wstawki antydebugowe. Czytelnik może rozbudować i zmodyfikować go według własnych potrzeb. Można zmienić go w następujący sposób:

- Dodanie większej ilości elementów służących do generowania wstawek.
- Dodanie generacji wstawek dla różnych języków programowania.
- Dodanie generacji kilku rodzajów wstawek, w tym parametryzowanych.
- Stworzenie bardziej skomplikowanych wstawek.
- Stworzenie mniej schematycznych wstawek.

Należy przy tym pamiętać o tym, że program po prostu wstawia wstawkę tam, gdzie zobaczy stosowny tekst. Może więc się zdarzyć, że kod źródłowy po modyfikacji nie będzie się kompilować lub w ogóle nie będzie działać. Zależy to od sposobu użycia oznakowania wstawek. Jeśli ich treść pojawi się w nietypowym miejscu, na przykład w ciągu znaków do wyświetlenia na konsoli itp., to działanie programu może zmienić się na nieprawidłowe.

Kod programu jest dość prosty. Podstawową funkcją jest `ModyfikujPlik`, która odczytuje plik, wyszukuje frazy `// ZABEZPIECZ` i podmienia je na losowo wygenerowane wstawki assemblerowe. Generowane są dwa typy wstawek:

- Statyczna — od 1 do 9 rozkazów assemblera wylosowanych z tablicy `statyczneWstawki`.
- Fałszywe skoki — trzy przeplatające się fragmenty kodu (wylosowane jak wstawka statyczna).

Oto główna funkcja:

```
static void ModyfikujPlik(string aŚcieżka)
    {
        // odczyt pliku
         TextReader reader = new StreamReader(aŚcieżka,
        ↪System.Text.Encoding.Default);
        string zawartość = reader.ReadToEnd();
        reader.Close();

        // modyfikacja zawartości
        int ile = 0;
        while (zawartość.IndexOf(oznaczenieWstawki) != -1)
        {
            int indeksWstawki = zawartość.IndexOf(oznaczenieWstawki);
            string wstawka = UtwórzWstawkę();
            zawartość = zawartość.Substring(0, indeksWstawki) + wstawka +
            ↪zawartość.Substring(indeksWstawki + oznaczenieWstawki.Length);
            ile++;
        }

        // zapis pliku
        TextWriter writer = new StreamWriter(aŚcieżka, false,
        ↪System.Text.Encoding.Default);
        writer.Write(zawartość);
        writer.Close();

        Console.WriteLine(aŚcieżka + " został zmodyfikowany "
        ↪+ ile.ToString() + " razy");
    }
```

Pojedynczą wstawkę tworzy funkcja `UtwórzWstawkę()`:

```
static string UtwórzWstawkę()
{
    string ciągPodmiany = początekWstawki;
    int ile = rd.Next(1, 3);
    for (int i = 0; i < ile; i++)
    {
        if(rd.Next(0,2) == 0)
            ciągPodmiany += UtwórzSztuczneSkoki();
        else
            ciągPodmiany += UtwórzStatycznąWstawkę();
    }
    return ciągPodmiany + koniecWstawki;
}
```

Tę funkcję zmodyfikuj, jeśli chcesz mieć więcej typów wstawek. Po prostu nie losuj liczb z zakresu <0; 2), lecz większego, i zastąp warunek instrukcją `switch`. Funkcje `UtwórzSztuczneSkoki` i `UtwórzStatycznąWstawkę` wyglądają następująco:

```
static string UtwórzStatycznąWstawkę()
{
    int ile = rd.Next(1, 10);
    string retWart = "";
    for (int i = 0; i < ile; i++)
                {
```

```
            retWart += statyczneWstawki[rd.Next(0, statyczneWstawki.Length)] +
            ➥"\r\n";
                    }
        return retWart;
    }

    static string UtwórzSztuczneSkoki()
    {
        string retWart = "";
        string etykieta1 = UtwórzEtykietę();
        string etykieta2 = UtwórzEtykietę();
        string etykieta3 = UtwórzEtykietę();
        retWart += "__asm JMP " + etykieta1 + ";\r\n";
        retWart += etykieta2 + ":" + UtwórzStatycznąWstawkę();
        retWart += "__asm JMP " + etykieta3 + ";\r\n";
        retWart += etykieta1 + ":" + UtwórzStatycznąWstawkę();
        retWart += "__asm JMP " + etykieta2 + ";\r\n";
        retWart += etykieta3 + ":" + UtwórzStatycznąWstawkę();
        return retWart;
    }
```

Druga z nich tworzy skoki układające się w ciąg:

1. Idź do 3
2. Idź do 4
3. Idź do 2
4. Koniec

Pomiędzy nimi znajdują się statyczne wstawki wygenerowane przez `UtwórzStatyczną-Wstawkę ()`, której z kolei kod jest tak prosty, że raczej nie trzeba go objaśniać. Warto jeszcze dodać, że każda wstawka zaczyna się odłożeniem stanu rejestrów i flag na stos, a kończy pobraniem ich stamtąd. Dzięki temu stan programu nie zmienia się.

Zadania do samodzielnego wykonania

1. Rozwiń program GenerujZabezpieczenia zgodnie z sugestiami w treści rozdziału.

Pytania kontrolne

P 6.1. Jakie są wady makr antydebugowych wstawianych ręcznie do kodu?

a) Są całkowicie nieskuteczne i nie przeszkadzają krakerom.

b) W niektórych językach programowania ich implementacja może być trudna.

c) Nie jest łatwo napisać takie makra nieschematycznie, zwykle są podobne do siebie.

d) Znajdują się w kodzie źródłowym, co zaciemnia go i utrudnia testowanie.