Polimorficzne algorytmy szyfrowania

Autor Bartosz Wójcik | Opublikowane Czerwiec 2013 | Magazyn Programista 6/2013 (13)

W tym artykule znajdziecie informacje o dynamicznym generowaniu unikalnych algorytmów szyfrowania, które zostaną zbudowane krok po kroku w kodzie assemblera x86.

Powszechnie dostępne algorytmy szyfrowania

Algorytmów szyfrowania dostępnych jest cała masa, są algorytmy szyfrujące blokowo (kolejne bloki o określonych rozmiarach), do których można zaliczyć np.:

- AES (Rijndael)
- Blowfish
- Twofish
- DES
- Serpent
- TEA – Tiny Encryption Algorithm
- RC5, RC6

Istnieją również algorytmy służące do szyfrowania strumieniowego (bajt po bajcie), takie jak popularny algorytm *RC4*. Algorytmy te w większości są symetryczne, co oznacza, że do szyfrowania i odszyfrowania danych wykorzystywany jest ten sam klucz. Oprócz algorytmów symetrycznych, istnieje odrębna grupa funkcji szyfrujących, bazująca na infrastrukturze klucza publicznego. Do takich algorytmów można zaliczyć:

- RSA
- ElGamal
- ECC

Szyfrowanie w takich algorytmach odbywa się zwykle za pomocą klucza publicznego, a odszyfrować dane można jedynie kluczem prywatnym. Klucze publiczne, jak nazwa wskazuje, mogą być publikowane w Internecie (tak jak klucze w systemie szyfrowania *PGP*). Taki sposób szyfrowania zapewnia, że nikt postronny bez łamania klucza publicznego nie będzie w stanie odczytać tak zaszyfrowanej wiadomości. W przypadku algorytmu RSA, złamanie klucza publicznego polega na rozłożeniu go na dwie liczby pierwsze (jest on iloczynem dwóch, dużych liczb pierwszych)..Jest to proces niezwykle czasochłonny. Firma *RSA* organizowała nawet konkursy na łamanie kluczy *RSA*, w których największa nagroda wynosiła 200000 USD za złamanie klucza o długości 2048 bitów.

Wszystkie wymienione powyżej algorytmy posiadają doskonałą dokumentację, ich implementacje można znaleźć w wielu językach programowania.

Silniki polimorficzne

Polimorfizm ma kilka znaczeń w informatyce, jednak omawiany tutaj dotyczy sposobu generowania unikalnego kodu. Wszystko ma swój początek wśród wirusów komputerowych. Twórcy wirusów komputerowych już w czasach *MS-DOS* , aby uchronić się przed detekcją antywirusową, stosowali szyfrowanie kodu wirusów. Proste algorytmy były jednak bardzo szybko oznaczane w sygnaturach programów antywirusowych i aby utrudnić detekcję, opracowano algorytmy, których kod był za każdym razem dynamicznie generowany (za każdym razem był inny), co zapobiegało wykrywaniu przez stałe sygnatury. Pierwsze polimorficzne silniki datowane są już na 1990 rok.

Z czasem algorytmy polimorficzne bardzo ewoluowały i stawały się coraz bardziej skomplikowane, aby jak najbardziej utrudnić analizę oraz emulację przez silniki

programów antywirusowych. Do takich znanych algorytmów można zaliczyć m.in.:

- KME – *Kewl Mutation Engine* (autor z0mbie i Vecna) - silnik bazujący na szyfrowaniu wartości przekazywanych jedynie przez stos (zaszyfrowane dane stanowią integralną część kodu, nie ma tam zaszyfrowanego bufora nigdzie zapisanego)
- MMXE – *MultiMedia eXtensions Engine* (autor Billy Belcebu) - silnik polimorficzny generujący kod MMX
- TUAREG – *Tameless Unpredictable Anarchic Relentless Encryption Generator* (autor Mental Driller) – wprowadzający nielinearne deszyfrowanie danych i wiele innowacyjnych metod do silnika polimorficznego
- PPE-II – *Prizzy Polymorphic Engine* (autor Prizzy) – silnik polimorficzny wykorzystujący kod *MMX* i *FPU* oraz celowe metody *brute-force* w swoim kodzie, aby spowolnić działanie emulatorów antywirusowych

W większości przypadków, tego rodzaju szyfrowanie było wykorzystywane w infektorach plików wykonywalnych. Poziom skomplikowania i fakt, że utworzenie takiego silnika wymaga sporej wiedzy z zakresu assemblera oraz wiedzy, jak zbudowane są instrukcje assemblera, sprawił, że obecnie silniki polimorficzne nie są prawie wcale stosowane (lub bardzo rzadko, jak np. w przypadku wirusa *Virut*, który *nota bene* ma najprawdopodobniej polskie korzenie).

Szersze zastosowanie silniki polimorficzne znalazły za to w systemach ochrony oprogramowania przed złamaniem, czyli w *exe-protectorach*, takich jak np. *PELock*, *ASProtect*, *EnigmaProtector*, *Themida*, *Obsidium*. Stosowane są po to, aby utrudnić analizę *crackerom* (czyli osobom zajmującym się przełamywaniem zabezpieczeń oprogramowania) oraz aby utrudnić lub uniemożliwić napisanie automatycznych narzędzi odbezpieczających tzw. *unpackerów*. Jeśli posiadasz zainstalowany jakiś program *shareware*, który jest zabezpieczony *exe-protectorem*, możesz być na 99% pewien, że korzysta on z algorytmu szyfrowania wygenerowanego przez jakiś silnik polimorficzny.

Nasz własny silnik polimorficzny

Poniżej zostaną zaprezentowane wszystkie kroki potrzebne do stworzenia prostego silnika polimorficznego, który posłuży nam do zaszyfrowania dowolnych danych oraz wygenerowania unikalnego kodu funkcji deszyfrującej, zawierającej w swoim ciele zaszyfrowany blok danych.

Do utworzenia silnika zostanie wykorzystany język programowania C++ oraz biblioteka *AsmJit*, która służy do dynamicznego generowania kodu assemblera, zarówno w wersji 32, jak i 64 bitowej. Biblioteka *AsmJit* umożliwia tworzenie kodu assemblera z poziomu C++, tak jak byśmy pisali go ręcznie. Możliwe jest tworzenie kodu 32 bitowego oraz 64 bitowego, a nawet stosowanie pseudoinstrukcji, dzięki którym możliwe jest wykorzystanie tej biblioteki do utworzenia kodu zarówno 32, jak i 64 bitowego. W naszym przykładzie utworzymy 32 bitowy kod.

Listing 1. Pseudokod funkcji deszyfrującej w C++, którą wygenerujemy.

```
DWORD DecryptionProc(PDWORD lpdwOutput)
{
  // losowo generowany klucz szyfrujący
  DWORD dwDecryptionKey = 0xA39383D;

  // wskaźnik zaszyfrowanych danych,
  // które znajdują się na końcu funkcji
  // deszyfrującej
  PDWORD lpdwInput = reinterpret_cast<PDWORD>(&cEncryptedData);

    // główna pętla deszyfrująca,
    // składająca się z losowo
    // dobranych instrukcji szyfrujących
    for (DWORD i = 0; i < ILOSC_BLOKOW; i++)
    {
      DWORD dwInputBlock = lpdwInput[i];

      dwInputBlock ^= 0x453BC;
      dwInputBlock += dwDecryptionKey;
      ...

      lpdwOutput[i] = dwInputBlock;
    }

    // zwróć rozmiar odszyfrowanych danych
    // (w bajtach)
    return ROZMIAR_ODSZYFROWANYCH_DANYCH;

    // zaszyfrowane dane, zapisane na końcu
    // funkcji
    BYTE cEncryptedData[] = { 0xAB, 0xBA... };
}
```

Nasza funkcja deszyfrująca będzie posiadać tylko jeden parametr, który posłuży jako wskaźnik do bufora wyjściowego, gdzie zostaną zapisane odszyfrowane dane. Funkcja zwróci rozmiar odszyfrowanych danych. Brzmi prosto? Zatem zabieramy się do pracy!

Losowy dobór rejestrów

Funkcja deszyfrująca będzie korzystała ze standardowych, 32 bitowych rejestrów procesora i tak, parametry przekazywane do funkcji będą odczytywane przez wskaźnik ramki stosu znajdujący się w rejestrze `EBP`. Rejestry wykorzystywane do

deszyfrowania danych, rejestry trzymające wskaźniki do zaszyfrowanego bufora pamięci oraz wskaźnik do bufora wyjściowego, będą losowo dobierane za każdym razem przy generowaniu funkcji.

Listing 2. Losowy dobór rejestrów, wykorzystywanych w funkcji.

```
////////////////////////////////////////////////////////
//
// dobierz losowe rejestry
//
////////////////////////////////////////////////////////

void CMutagenSPE::RandomizeRegisters()
{
  // zestaw losowych rejestrów
  AsmJit::GPReg cRegsGeneral[] = { eax, ecx, ebx, edx, esi, edi };

  // wymieszaj kolejność rejestrów w tablicy
  mixup_array(cRegsGeneral, _countof(cRegsGeneral));

  // rejestr, w którym będzie znajdował
  // się wskaźnik do zaszyfrowanych danych
  regSrc = cRegsGeneral[0];

  // rejestr, w którym znajdzie się wskaźnik
  // do bufora wyjściowego (podany jako parametr
  // funkcji
  regDst = cRegsGeneral[1];

  // rejestr, w którym znajdzie się rozmiar
  // zaszyfrowanego bloku danych
  regSize = cRegsGeneral[2];

  // rejestr trzymający klucz deszyfrujący
  regKey = cRegsGeneral[3];

  // rejestr, który będzie trzymał bieżący
  // blok danych i na którym będą wykonywane
  // operacje deszyfrujące
  regData = cRegsGeneral[4];

  // zestaw losowych rejestrów, które
  // będą zachowane pomiędzy wywołaniami
  // funkcji
  AsmJit::GPReg cRegsSafe[] = { esi, edi, ebx };

  // wymieszaj kolejność rejestrów w tablicy
  mixup_array(cRegsSafe, _countof(cRegsSafe));

  regSafe1 = cRegsSafe[0];
  regSafe2 = cRegsSafe[1];
  regSafe3 = cRegsSafe[2];
}
```

Taki losowy sposób doboru rejestrów sprawi, że wygenerowany kod będzie za każdym razem inny.

Prolog funkcji deszyfrującej

Prolog funkcji deszyfrującej to nic innego jak początek funkcji, który zawiera takie elementy jak ustawienie ramki stosu. Ramka stosu to specjalna konstrukcja,

pozwalająca zarezerwować miejsce na stosie dla zmiennych lokalnych (jeśli z takich funkcja by korzystała) oraz pozwalająca odczytać parametry wejściowe funkcji. W naszym przypadku jedynym parametrem będzie adres docelowy bufora w pamięci, gdzie dane mają zostać odszyfrowane.

Rysunek 1. Struktura ramki stosu po wywołaniu funkcji

Kod	Stos	EBP
...	...	...
push parametrX	Parametr X przekazany do funkcji	...
push parametr2	Parametr 2 przekazany do funkcji	EBP+12
push parametr1	Parametr 1 przekazany do funkcji	EBP+8
call Funkcja	Adres powrotu z funkcji deszyfrującej	EBP+4
push ebp	Zachowany rejestr EBP	EBP+0
mov ebp, esp	Zmienna lokalna 1	EBP-4
add esp, -8	Zmienna lokalna 2	EBP-8

Listing 3. Generowanie prologu funkcji szyfrującej.

```
//////////////////////////////////////////////////////////
//
// generuj prolog funkcji deszyfrującej
//
//////////////////////////////////////////////////////////

void CMutagenSPE::GenerateProlog()
{
  // prolog funkcji, zachowujemy oryginalną
  // wartość rejestru EBP i wszelkie parametry
  // będziemy pobierać przez rejestr EBP
  if (rnd_bin() == 0)
  {
    a.push(ebp);
    a.mov(ebp,esp);
  }
  else
  {
    // odpowiednik instrukcji
    // push ebp
    // mov ebp,esp
    a.enter(imm(0), imm(0));
  }

  // jeśli nasza funkcja jest w konwencji stdcall
  // i modyfikuje rejestry ESI EDI EBX, należy
  // je zachować na początku funkcji
  a.push(regSafe1);
```

```
  a.push(regSafe2);
  a.push(regSafe3);

  // wczytaj do rejestru regDst wskaźnik
  // do bufora wyjściowego, gdzie znajdą się
  // odszyfrowane dane, wskaźnik przekazany
  // jest jako pierwszy parametr funkcji
  a.mov(regDst, dword_ptr(ebp, 0x08 + (4 * 0)));
}
```

Adres zaszyfrowanych danych

Nasz kod został dynamicznie wygenerowany i może być umieszczony w dowolnym obszarze pamięci i stamtąd uruchomiony (pod warunkiem, że ten obszar będzie posiadał odpowiednie flagi do wykonywania kodu). W takich wypadkach nie możemy posługiwać się bezwzględnymi adresami pamięci, bo po prostu nie wiemy, gdzie może znaleźć się funkcja. Zaszyfrowane dane znajdują się zaraz na końcu funkcji deszyfrującej i adres do nich należy obliczyć dynamicznie, posługując się relatywnym adresowaniem.

Wykorzystamy tutaj technikę tzw. *delta offset*. Polega ona na wykonaniu instrukcji assemblera `call` używanej do wywoływania funkcji oraz faktu, że ta instrukcja zapisuje na stosie adres powrotu, po czym wykonuje skok do wybranego adresu pamięci.

Listing 4. Pobieranie bieżącego adresu kodu z wykorzystaniem techniki delta offset.

```
Funkcja proc near


; zachowaj oryginalną wartość rejestru EBP

    push    ebp


; ustawianie delta offset

    call    delta_offset

delta_offset:


; w tym miejscu rejestr EBP wskazuje na adres

; w pamięci labela "delta_offset"

    pop     ebp


; pobranie adresu var_1 poprzez wykorzystanie
```

```
; relatywnego adresowania względem delta_offset
    lea     eax,[ebp + (var_1 - delta_offset)]

; zweryfikuj czy adresowanie delta offset
; jest poprawne, zwróć TRUE / FALSE
    cmp     dword ptr[eax],0DEADC0DEh
    sete    al
    movzx   eax,al

; przywróć wartość rejestru EBP
    pop     ebp

    ret

; wartość zapisana w ciele funkcji
    var_1   dd 0DEADC0DEh

Funkcja endp
```

Adres powrotu w tym przypadku wskaże na kolejną instrukcję i posługując się tą wiedzą i rozmiarem pozostałego fragmentu funkcji, można dodać te wartości do siebie, uzyskując automatycznie adres zaszyfrowanego fragmentu danych, znajdującego się na końcu funkcji.

Ciekawą alternatywą dla tak obliczanego adresu *delta offset* jest metoda wykorzystująca instrukcję *FPU* - `fnstenv`, która zapisuje stan środowiska *FPU*, w tym informacje o położeniu ostatnio wykonywanej instrukcji *FPU* w pamięci. Można to wykorzystać jako alternatywny sposób na obliczanie *delta offset*.

Listing 5. Wykorzystanie instrukcji FPU do obliczenia adresu delta offset.

```
DeltaOffsetFPUTest proc uses esi

; lokalny bufor na środowisko FPU
    local   fpEnvironment[32]:byte
```

```
; inicjalizuj FPU
    finit

; wykonaj dowolną instrukcję FPU (np. fld1,
; fldz, fldln2, fldlg2 etc.)
delta_offset:
    fldpi

; zapisz stan środowiska FPU
    lea     eax,fpEnvironment
    fnstenv byte ptr[eax]

; odczytaj adres ostatniej instrukcji FPU
; powinien wskazywać na adres instrukcji
; fldpi - czyli adres labela delta_offset
    mov     esi,dword ptr[eax+12]

; zdejmij ze stosu FPU 1 rejestr (po fldpi)
    fistp   dword ptr[esp-4]

; pobranie adresu var_1 poprzez wykorzystanie
; relatywnego adresowania względem delta_offset
    lea     eax,[esi + (var_1 - delta_offset2)]

; zweryfikuj czy adresowanie delta offset
; jest poprawne, zwróć TRUE / FALSE
    cmp     dword ptr[eax],0ABBAh
    sete    al
    movzx   eax,al

    ret
```

```
; wartość zapisana w ciele funkcji

    var_1   dd 0ABBAh


DeltaOffsetFPUTest endp
```

Taki rodzaj obliczania delta offset wykorzystywany jest czasami w *exploitach*, np. we frameworku *Metasploit*. Ten kod wykorzystywany jest również do wykrywania debuggerów, gdyż instrukcja zapisująca stan środowiska FPU, uruchomiona pod debuggerem nie zwróci adresu ostatnio wykonywanej instrukcji FPU w naszym programie, tylko adres innych instrukcji FPU, które są bezpośrednio lub pośrednio wykonywane przez funkcje, z których korzysta debugger. Można w ten sposób wykryć śledzenie kodu przez debuggery takie jak np. *OllyDbg*.

Kalkulacje *delta offset* mogą być podejrzane dla programów antywirusowych, gdyż w normalnych aplikacjach nie znajdziemy takich sekwencji kodu. Połączenie instrukcji `call` + `pop r32` może powodować oflagowanie aplikacji, która zawiera taki kod, jako podejrzanej. Aby temu zapobiec, należy pomiędzy tymi instrukcjami, wygenerować inny kod oraz skorzystać z alternatywnej metody pobrania wartości ze stosu.

Listing 6. Generowanie kodu dla relatywnego adresowania.

```
//////////////////////////////////////////////////////////
//
// generuj delta offset
//
//////////////////////////////////////////////////////////

void CMutagenSPE::GenerateDeltaOffset()
{
  // generuj kod, który pozwoli nam ustawić
  // wskaźnik do zaszyfrowanych danych na
  // końcu funkcji deszyfrującej

  // funkcja_deszyfrująca:
  // ...
  // call delta_offset
  // mov eax,1 | xor eax,eax ; \
  // leave                   ;  > nieużywane instrukcje
  // ret 4                   ; /
  // delta_offset:
  // pop regSrc
  // add regSrc, (zaszyfrowane_dane-delta_offset +
  // ...          + rozmiar nieużywanych instrukcji)
  // ret 4
  // db 0CCh, 0CCh...
  // zaszyfrowane_dane:
  // db 0ABh, 0BBh, 083h...

  // utworzenie labela dla delta offset
  lblDeltaOffset = a.newLabel();

  // generuj instrukcję call delta_offset
```

```
  a.call(lblDeltaOffset);

  sysint_t posUnusedCodeStart = a.getOffset();

  // aby uniknąc fałszywych detekcji przez
  // oprogramowanie antywirusowe, unikamy
  // typowej konstrukcji delta offset, czyli
  // call + pop, wstawiając między instrukcje
  // nieużywaną sekwencję kodu, w naszym wypadku
  // sekwencję udającą typowy kod, który
  // wraca z wywołanej funkcji
  if (rnd_bin() == 0)
  {
    a.mov(eax, imm(1));
  }
  else
  {
    a.xor_(eax,eax);
  }

  a.leave();
  a.ret(1 * sizeof(DWORD));

  // oblicz rozmiar nieużywanego kodu
  // czyli różnicę między bieżącą pozycją
  // w kodzie, a pozycją początkową
  dwUnusedCodeSize = static_cast<DWORD>(a.getOffset() - posUnusedCodeStart);

  // ustaw w tym miejscu label "delta_offset:"
  a.bind(lblDeltaOffset);

  posDeltaOffset = a.getOffset();

  // zamiast instrukcji pop użyjemy innego
  // odpowiednika, aby nie wzbudzić podejrzeń
  // programów antywirusowych

  //a.pop(regSrc);
  a.mov(regSrc, dword_ptr(esp));
  a.add(esp, imm(sizeof(DWORD)));

  // w rejestrze regSrc znajdzie się adres
  // labela "delta_offset:", należy go skorygować
  // o rozmiar pozostałej części funkcji
  // (którego jeszcze nie znamy i później to
  // nastąpi) i nieużywanych instrukcji
  // na razie tymczasowo zapisujemy tam
  // wartość 987654321, aby AsmJit wygenerował
  // długą formę instrukcji "add"
  a.add(regSrc, imm(987654321));

  // zapisz pozycję do wartości DWORD, którą
  // trzeba będzie później uaktualnić o rozmiar
  // pozostałej części funkcji dekodującej
  posSrcPtr = a.getOffset() - sizeof(DWORD);
}
```

Na razie nie jest znany rozmiar pozostałego bloku funkcji deszyfrującej, dlatego zostanie on uaktualniony w kodzie po wygenerowaniu wszystkich innych instrukcji funkcji deszyfrującej.

Szyfrowanie danych

Szyfrowanie danych będzie odbywało się w blokach 4 bajtowych. Rozmiar danych może być mniejszy (będzie zaokrąglony do 4). Do procesu szyfrowania zostaną wylosowane pseudoinstrukcje, które później posłużą do wygenerowania kodu deszyfrującego.

Listing 7. Szyfrowanie danych.

```
//////////////////////////////////////////////////////////
//
// generuj klucze szyfrujące, instrukcje szyfrujące
// i w końcu szyfruj dane wejściowe
//
//////////////////////////////////////////////////////////

void CMutagenSPE::EncryptInputBuffer(PBYTE lpInputBuffer, \
                                     DWORD dwInputBuffer, \
                                     DWORD dwMinInstr, \
                                     DWORD dwMaxInstr)
{
  // losuj klucz szyfrujący
  dwEncryptionKey = rnd_dword();

  // wyrównany rozmiar wejściowego bufora
  DWORD dwAlignedSize = align_dword(dwInputBuffer);

  // ilość bloków do zaszyfrowania
  // podziel rozmiar wejściowych
  // danych na bloki o rozmiarze 4
  // bajtów (DWORD)
  dwEncryptedBlocks = dwAlignedSize / sizeof(DWORD);

  PDWORD lpdwInputBuffer = reinterpret_cast<PDWORD>(lpInputBuffer);

  // alokuj pamięć na wyjściowe dane
  // (jej rozmiar będzie zaokrąglony
  // do wyrównania bloku)
  di_valloc(&diEncryptedData, dwAlignedSize);

  PDWORD lpdwOutputBuffer = reinterpret_cast<PDWORD>(diEncryptedData.lpPtr);

  // losuj ile ma być instrukcji szyfrujących
  dwCryptOpsCount = rnd_range(dwMinInstr, dwMaxInstr);

  // alokuj pamięć na tablicę, gdzie zapisane
  // zostaną informacje o kolejnych instrukcjach
  // szyfrujących
  di_valloc(&diCryptOps, dwCryptOpsCount * sizeof(SPE_CRYPT_OP));

  // ustaw bezpośredni wskaźnik do tej
  // tablicy w pomocniczej zmiennej
  lpcoCryptOps = reinterpret_cast<P_SPE_CRYPT_OP>(diCryptOps.lpPtr);

  // generuj instrukcje szyfrujące oraz ich typ
  for (DWORD i = 0; i < dwCryptOpsCount; i++)
  {
    // czy instrukcja ma wykorzystywać
    // szyfrowanie pomiędzy rejestrem
    // regData a regKey?
    lpcoCryptOps[i].bCryptWithReg = rnd_bool();

    // rejestr, na którym operujemy
```

```
    lpcoCryptOps[i].regDst = regData;

    // jeśli instrukcja nie wykorzystuje
    // rejestru regKey, losuj klucz szyfrujący,
    // który zostanie bezpośrednio użyty do
    // wygenerowania
    if (lpcoCryptOps[i].bCryptWithReg == FALSE)
    {
      lpcoCryptOps[i].dwCryptValue = rnd_dword();
    }
    else
    {
      lpcoCryptOps[i].regSrc = regKey;
    }

    // losuj rodzaj instrukcji szyfrującej
    lpcoCryptOps[i].cCryptOp = static_cast<BYTE>(rnd_range(SPE_CRYPT_OP_ADD,
SPE_CRYPT_OP_NEG));
  }

  // szyfruj wejściowe dane według
  // wylosowanych wcześniej instrukcji
  for (DWORD i = 0, dwInitialEncryptionKey = dwEncryptionKey; \
       i < dwEncryptedBlocks; i++)
  {
    // pobierz kolejny blok do zaszyfrowania
    DWORD dwInputBlock = lpdwInputBuffer[i];

    // pętla szyfrująca, wykonująca wszystkie
    // instrukcje szyfrujące na bloku danych
    for (DWORD j = 0, dwCurrentEncryptionKey; j < dwCryptOpsCount; j++)
    {
      if (lpcoCryptOps[j].bCryptWithReg == FALSE)
      {
        dwCurrentEncryptionKey = lpcoCryptOps[j].dwCryptValue;
      }
      else
      {
        dwCurrentEncryptionKey = dwInitialEncryptionKey;
      }

      // w zależności od instrukcji szyfrującej
      // wykonaj odpowiednią modyfikację
      // bloku danych
      switch(lpcoCryptOps[j].cCryptOp)
      {
      case SPE_CRYPT_OP_ADD:
        dwInputBlock += dwCurrentEncryptionKey;
        break;
      case SPE_CRYPT_OP_SUB:
        dwInputBlock -= dwCurrentEncryptionKey;
        break;
      case SPE_CRYPT_OP_XOR:
        dwInputBlock ^= dwCurrentEncryptionKey;
        break;
      case SPE_CRYPT_OP_NOT:
        dwInputBlock = ~dwInputBlock;
        break;
      case SPE_CRYPT_OP_NEG:
        dwInputBlock = 0L - dwInputBlock;
        break;
      }
    }

    // zapisz zaszyfrowany blok do bufora
    lpdwOutputBuffer[i] = dwInputBlock;
  }
}
```

Ustawianie kluczy szyfrujących

Nasz algorytm będzie wykorzystywał losowo generowane klucze szyfrujące. Klucze te są na początku funkcji przypisywane do wcześniej wylosowanych rejestrów oznaczonych jako `rKey1` i `rKey2`.

Listing 8. Generowanie kodu inicjalizującego wybrane rejestry kluczami szyfrującymi.

```
//////////////////////////////////////////////////////////
//
// ustaw klucze wykorzystane do odszyfrowania danych
//
//////////////////////////////////////////////////////////

void CMutagenSPE::SetupDecryptionKeys()
{
  // ustaw w rejestrze regKey klucz deszyfrujący,
  // który dodatkowo będzie zaszyfrowany
  DWORD dwKeyModifier = rnd_dword();

  // losowo generuj instrukcję ustawiającą
  // klucz deszyfrujący
  switch(rnd_max(2))
  {
  // mov regKey,dwKey - dwMod
  // add regKey,dwMod
  case 0:
    a.mov(regKey, imm(dwEncryptionKey - dwKeyModifier));
    a.add(regKey, imm(dwKeyModifier));
    break;

  // mov regKey,dwKey + dwMod
  // sub regKey,dwMod
  case 1:
    a.mov(regKey, imm(dwEncryptionKey + dwKeyModifier));
    a.sub(regKey, imm(dwKeyModifier));
    break;

  // mov regKey,dwKey ^ dwMod
  // xor regKey,dwMod
  case 2:
    a.mov(regKey, imm(dwEncryptionKey ^ dwKeyModifier));
    a.xor_(regKey, imm(dwKeyModifier));
    break;
  }
}
```

Odszyfrowanie danych

Korzystając z wcześniej wygenerowanych pseudoinstrukcji, które posłużyły do zaszyfrowania danych, wygenerowana będzie pętla zawierająca odwrócony algorytm szyfrujący. Przed kodem pętli zostanie zainicjalizowany rejestr oznaczony jako `regSize`, wartością określającą ilość bloków, jaka ma zostać odszyfrowana.

Listing 9. Generowanie kodu pętli deszyfrującej.

```
//////////////////////////////////////////////////////////
//
// generuj kod deszyfrująca (główna pętla deszyfrująca)
//
//////////////////////////////////////////////////////////

void CMutagenSPE::GenerateDecryption()
{
  // ustaw rozmiar zaszyfrowanych danych
  // (ilość bloków)
  a.mov(regSize, imm(dwEncryptedBlocks));

  // umieśc tutaj początek pętli deszyfrującej
  Label lblDecryptionLoop = a.newLabel();

  a.bind(lblDecryptionLoop);

  // instrukcja odczytująca blok danych
  // z rejestru regSrc
  a.mov(regData, dword_ptr(regSrc));

  // buduj kod deszyfrujący, generując instrukcje
  // deszyfrujące (w odwrotnej kolejności niż są
  // na liście)
  for (DWORD i = dwCryptOpsCount - 1; i != -1L; i--)
  {
    // szyfrowanie korzystało albo z klucza,
    // który znajduje się w rejestrze regKey,
    // lub ze stałej wartości, więc odpowiednio
    // należy wygenerować instrukcje deszyfrujące
    // (odwrotne w działaniu do szyfrujących)
    if (lpcoCryptOps[i].bCryptWithReg == FALSE)
    {
      DWORD dwDecryptionKey = lpcoCryptOps[i].dwCryptValue;

      switch(lpcoCryptOps[i].cCryptOp)
      {
      case SPE_CRYPT_OP_ADD:
        a.sub(lpcoCryptOps[i].regDst, imm(dwDecryptionKey));
        break;
      case SPE_CRYPT_OP_SUB:
        a.add(lpcoCryptOps[i].regDst, imm(dwDecryptionKey));
        break;
      case SPE_CRYPT_OP_XOR:
        a.xor_(lpcoCryptOps[i].regDst, imm(dwDecryptionKey));
        break;
      case SPE_CRYPT_OP_NOT:
        a.not_(lpcoCryptOps[i].regDst);
        break;
      case SPE_CRYPT_OP_NEG:
        a.neg(lpcoCryptOps[i].regDst);
        break;
      }
    }
    else
    {
      switch(lpcoCryptOps[i].cCryptOp)
      {
      case SPE_CRYPT_OP_ADD:
        a.sub(lpcoCryptOps[i].regDst, lpcoCryptOps[i].regSrc);
        break;
      case SPE_CRYPT_OP_SUB:
        a.add(lpcoCryptOps[i].regDst, lpcoCryptOps[i].regSrc);
        break;
      case SPE_CRYPT_OP_XOR:
        a.xor_(lpcoCryptOps[i].regDst, lpcoCryptOps[i].regSrc);
        break;
      case SPE_CRYPT_OP_NOT:
```

```
        a.not_(lpcoCryptOps[i].regDst);
        break;
      case SPE_CRYPT_OP_NEG:
        a.neg(lpcoCryptOps[i].regDst);
        break;
      }
    }
  }

  // instrukcja zapisująca odszyfrowany blok
  // do bufora wyjściowego
  a.mov(dword_ptr(regDst), regData);

  // zwiększ wskaźnik w rejestrze regSrc i regDst
  a.add(regSrc, imm(sizeof(DWORD)));
  a.add(regDst, imm(sizeof(DWORD)));

  // zmniejsz licznik pętli (ilość bloków
  // do odszyfrowania)
  a.dec(regSize);

  // sprawdź czy pętla się zakończyła
  // jeśli nie, nastąpi skok do początku
  a.jne(lblDecryptionLoop);
}
```

Pętla pobiera kolejne zaszyfrowane bloki danych z końca funkcji, wykonuje instrukcje deszyfrujące i zapisuje do bufora wyjściowego odszyfrowane dane. Po odszyfrowaniu bloku danych, aktualizowane są wskaźniki do zaszyfrowanych danych i wskaźnik do bufora wyjściowego oraz zmniejszany jest licznik pozostałych bloków do odszyfrowania, jeśli nie osiągnął on wartości 0, pętla zostaje powtórzona.

Ustawienie wartości i rejestrów wyjściowych

Nasza funkcja deszyfrująca zwracać będzie wartość typu `DWORD` (32 bitowa wartość typu `unsigned int`), zawierającą rozmiar odszyfrowanych danych. Wartość ta zwracana będzie w rejestrze procesora EAX.

Listing 10. Ustawianie wartości, jaką zwraca funkcja deszyfrująca (rozmiar odszyfrowanego bufora danych) oraz pozostałych wartości wyjściowych rejestrów procesora.

```
//////////////////////////////////////////////////////////
//
// ustaw rejestry wyjściowe, w tym wartość zwracanej
// funkcji
//
//////////////////////////////////////////////////////////

void CMutagenSPE::SetupOutputRegisters(SPE_OUTPUT_REGS *regOutput, DWORD dwCount)
{
  // jeśli nie ma być żadnych rejestrów
  // wyjściowych do ustawienia - wyjdź
  if ((regOutput == NULL) || (dwCount == 0))
  {
    return;
```

```
  }

  // mieszaj kolejność, w jakiej rejestry
  // wyjściowe będą ustawiane
  mixup_array(regOutput, dwCount);

  // generuj instrukcje ustawiające rejestry wyjściowe
  // mov r32, imm32
  for (DWORD i = 0; i < dwCount; i++)
  {
    a.mov(regOutput[i].regDst, imm(regOutput[i].dwValue));
  }
}
```

Nasz silnik polimorficzny pozwala na zdefiniowanie zestawu rejestrów wyjściowych (czyli możemy ustawić nie tylko to, co ma być zwrócone w rejestrze `EAX`), co umożliwia na przekazanie przez funkcję dodatkowych wartości liczbowych.

Epilog funkcji

Epilog, czyli końcowy fragment funkcji, w którym znajdzie się przywrócenie oryginalnej wartości rejestru `EBP` (wykorzystywanego jako ramka stosu) oraz ewentualnie, przywrócenie wrażliwych rejestrów, takich jak `ESI EDI EBX`, których stan musi być zachowywany pomiędzy wywołaniami funkcji w konwencji *stdcall*.

Listing 11. Generowanie epilogu funkcji deszyfrującej.

```
////////////////////////////////////////////////////////////
//
// generuj epilog funkcji deszyfrującej
//
////////////////////////////////////////////////////////////

void CMutagenSPE::GenerateEpilog(DWORD dwParamCount)
{
  // przywróć oryginalne wartości
  // rejestrów ESI EDI EBX
  a.pop(regSafe3);
  a.pop(regSafe2);
  a.pop(regSafe1);

  // przywróć wartość rejestru EBP
  if (rnd_bin() == 0)
  {
    a.leave();
  }
  else
  {
    // odpowiednik instrukcji "leave"
    a.mov(esp,ebp);
    a.pop(ebp);
  }

  // powrót do kodu wywołującego
  // funkcje, dodatkowo korygujemy
  // stos o rozmiar przekazanych
  // parametrów (konwencja stdcall)
```

```
  a.ret(imm(dwParamCount * sizeof(DWORD)));
}
```

Wyrównania

Instrukcje dostępu do pamięci są szybsze, jeśli dane, które odczytujemy lub zapisujemy, znajdują się na adresach wyrównanych do wartości podzielnych przez rozmiar odczytywanych bloków pamięci. Czyli np. dla naszych zaszyfrowanych danych, czytanych w blokach 32 bitowych (32 bitowe są rejestry), dane muszą być dostępne na adresie podzielnym przez 4. Ma to związek z wykorzystaniem pamięci podręcznej procesora i jeśli dane są wyrównane, po pierwszym odczycie mogą trafić do szybkiej pamięci podręcznej procesora *L1*. Niewyrównane dane będą czytane z wolniejszej pamięci cache *L2* lub bezpośrednio z pamięci komputera. Dla małych bloków danych brak wyrównania nie stanowi takiego problemu wydajnościowego, jednak dla dużych buforów jest to już realny problem, który trzeba wziąć pod uwagę.

Listing 12. Wyrównanie rozmiaru funkcji deszyfrującej do podanej wartości liczbowej.

```
//////////////////////////////////////////////////////////
//
// wyrównaj rozmiar kodu do określonej wartości
//
//////////////////////////////////////////////////////////

void CMutagenSPE::AlignDecryptorBody(DWORD dwAlignment)
{
  // pobierz rozmiar bieżącego kodu
  DWORD dwCurrentSize = a.getCodeSize();

  // wyrównaj rozmiar do wielokrotności
  // ustalonej liczby (np. 4)
  DWORD dwAlignmentSize = align_bytes(dwCurrentSize, dwAlignment) - dwCurrentSize;

  // sprawdź czy w ogóle ma być wyrównanie
  if (dwAlignmentSize == 0)
  {
    return;
  }

  // wstaw instrukcje wyrównujące (int3 lub nop)
  if (rnd_bin() == 0)
  {
    while (dwAlignmentSize--) a.int3();
  }
  else
  {
    while (dwAlignmentSize--) a.nop();
  }
}
```

Wyrównania między funkcjami zwykle wypełnione są instrukcjami `nop` (ang. *no-operation*) lub `int3` (przerwanie sygnalizujące pułapkę dla debuggera).

Wyrównywania stosowane są także w przypadku początkowych adresów pętli (instrukcjami równoznacznymi dla `nop`, jednak o dłuższej budowie binarnej np. `lea eax,[eax*8+eax+00000000]`), w naszym przypadku jednak pominiemy ten rodzaj optymalizacji.

Korygowanie adresów zależnych od delta offset

W tym momencie cały kod funkcji deszyfrującej został wygenerowany. Czas skorygować relatywne adresy, takie jak adres zaszyfrowanego bloku danych.

Listing 13. Korekta wcześniej wygenerowanego kodu, zawierającego adres do zaszyfrowanych danych.

```
//////////////////////////////////////////////////////////
//
// koryguj wszystkie instrukcje korzystające z
// relatywnego adresowania delta offset
//
//////////////////////////////////////////////////////////

void CMutagenSPE::UpdateDeltaOffsetAddressing()
{
  DWORD dwAdjustSize = static_cast<DWORD>(a.getOffset() - posDeltaOffset);

  // koryguj instrukcję ustawiającą wskaźnik
  // do zaszyfrowanego bloku danych na
  // końcu funkcji deszyfrującej, wskaźnik
  // ten znajduje się w rejestrze regSrc
  // i należy go skorygować o rozmiar
  // pozostałej części funkcji względem
  // labela delta_offset
  a.setDWordAt(posSrcPtr, dwAdjustSize + dwUnusedCodeSize);
}
```

Zapisanie zaszyfrowanego bloku danych

Gdy cały kod funkcji jest już wygenerowany, na jej końcu zapisany zostanie blok zaszyfrowanych danych.

Listing 14. Zapis zaszyfrowanych danych na końcu funkcji deszyfrującej.

```
//////////////////////////////////////////////////////////
//
// doklej zaszyfrowane dane na koniec kodu funkcji deszyfrującej
//
//////////////////////////////////////////////////////////

void CMutagenSPE::AppendEncryptedData()
{
  PDWORD lpdwEncryptedData = reinterpret_cast<PDWORD>(diEncryptedData.lpPtr);

  // wstaw na końcu funkcji zaszyfrowany
  // bufor (w blokach po 4 bajty)
  for (DWORD i = 0; i < dwEncryptedBlocks; i++)
  {
    a._emitDWord(lpdwEncryptedData[i]);
  }
```

```
}
```

Cały silnik polimorficzny

Znając kolejne elementy działania silnika polimorficznego, poniżej znajdziecie plik nagłówkowy jego klasy oraz implementację głównej funkcji generującej polimorficzny kod.

Listing 15. Definicja elementów klasy silnika polimorficznego.

```
#include "mutagen.h"

class CMutagenSPE : private CMutagen
{
public:
  CMutagenSPE(void);
  ~CMutagenSPE(void);

  // główna funkcja szyfrująca i generująca
  // polimorficzny kod
  CMutagen::erCodes PolySPE(PBYTE lpInputBuffer, \
                            DWORD dwInputBuffer, \
                            PBYTE *lpOutputBuffer, \
                            PDWORD lpdwOutputSize);

private:

  // struktura opisująca wartości rejestrów wyjściowych
  typedef struct _SPE_OUTPUT_REGS {

    // rejestr docelowy
    AsmJit::GPReg regDst;

    // wartość, jaka ma być zapisana w rejestrze
    DWORD dwValue;

  } SPE_OUTPUT_REGS, *P_SPE_OUTPUT_REGS;

  // opis instrukcji szyfrującej
  typedef struct _SPE_CRYPT_OP {

    // czy szyfrowanie ma się odbywać
    // między rejestrami czy między
    // rejestrem docelowym i wartością
    // dwCryptValue
    BOOL bCryptWithReg;

    AsmJit::GPReg regDst;
    AsmJit::GPReg regSrc;

    // instrukcja szyfrująca
    BYTE cCryptOp;

    // wartość szyfrująca
    DWORD dwCryptValue;

  } SPE_CRYPT_OP, *P_SPE_CRYPT_OP;

  enum
  {
    SPE_CRYPT_OP_ADD = 0,
    SPE_CRYPT_OP_SUB,
    SPE_CRYPT_OP_XOR,
    SPE_CRYPT_OP_NOT,
```

```
  SPE_CRYPT_OP_NEG,
};

// bufor z instrukcjami szyfrującymi
DATA_ITEM diCryptOps;

// bezpośredni wskaźnik do tablicy
// z instrukcjami szyfrującymi
P_SPE_CRYPT_OP lpcoCryptOps;

// liczba instrukcji szyfrujących
DWORD dwCryptOpsCount;

// wskaznik do zaszyfrowanego bloku danych
DATA_ITEM diEncryptedData;

// ilosc blokow zaszyfrowanego kodu
DWORD dwEncryptedBlocks;

// klucz szyfrujący
DWORD dwEncryptionKey;

// definicja assemblera biblioteki AsmJit
Assembler a;

// rejestr, w którym będzie zapisany wskaźnik
// do danych, które mają być odszyfrowane
AsmJit::GPReg regSrc;

// rejestr, w którym będzie wskaźnik do
// danych wyjściowych
AsmJit::GPReg regDst;

// rejestr trzymający rozmiar
// zaszyfrowanych danych
AsmJit::GPReg regSize;

// rejestr z kluczem deszyfrującym
AsmJit::GPReg regKey;

// rejestr, na którym będą wykonywane
// operacje deszyfrowania
AsmJit::GPReg regData;

// bezpieczne rejestry (ESI EDI EBX w losowej kolejności)
AsmJit::GPReg regSafe1, regSafe2, regSafe3;

// label określający delta offset
Label lblDeltaOffset;

// pozycja delta offset
sysint_t posDeltaOffset;

// relatywny adres zaszyfrowanego bloku danych
sysint_t posSrcPtr;

// rozmiar nieużywanego kodu pomiędzy
// delta offset, a instrukcją pobierającą
// tą wartość ze stosu
DWORD dwUnusedCodeSize;

// metody pomocnicze
void RandomizeRegisters();
void GenerateProlog();
void GenerateDeltaOffset();
void EncryptInputBuffer(PBYTE lpInputBuffer, \
                        DWORD dwInputBuffer, \
                        DWORD dwMinInstr, \
```

```
                          DWORD dwMaxInstr);
  void SetupDecryptionKeys();
  void GenerateDecryption();
  void SetupOutputRegisters(SPE_OUTPUT_REGS *regOutput, \
                            DWORD dwCount);
  void GenerateEpilog(DWORD dwParamCount);
  void AlignDecryptorBody(DWORD dwAlignment);
  void AppendEncryptedData();
  void UpdateDeltaOffsetAddressing();
};
```

Listing 16. Główna funkcja, szyfrująca i generująca polimorficzny kod.

```
//////////////////////////////////////////////////////////
//
// główna funkcja generująca kod polimorficzny dekryptora
//
//////////////////////////////////////////////////////////

CMutagen::erCodes CMutagenSPE::PolySPE(PBYTE lpInputBuffer, \
                                       DWORD dwInputBuffer, \
                                       PBYTE *lpOutputBuffer, \
                                       PDWORD lpdwOutputSize)
{
  //////////////////////////////////////////////////////////
  //
  // sprawdź parametry wejściowe
  //
  //////////////////////////////////////////////////////////

  if ( (lpInputBuffer == NULL) || (dwInputBuffer == 0) || \
       (lpOutputBuffer == NULL) || (lpdwOutputSize == NULL) )
  {
    return CMutagen::MUTAGEN_ERR_PARAMS;
  }

  // dobierz losowe rejestry
  RandomizeRegisters();

  //////////////////////////////////////////////////////////
  //
  // generuj kod funkcji polimorficznej
  //
  //////////////////////////////////////////////////////////

  // generuj prolog funkcji
  GenerateProlog();

  // ustaw relatywne adresowanie techniką delta offset
  GenerateDeltaOffset();

  // szyfruj dane wejściowe, generuj klucze szyfrujące
  // dodatkowe parametry określają minimalną i maksymalną
  // liczbę instrukcji szyfrujących, jakie zostaną
  // wygenerowane (nie ma tutaj ograniczeń, można ustawić
  // liczby rzędu kilku tysięcy, ale należy liczyć się z
  // tym, że wyjściowy kod będzie bardzo duży)
  EncryptInputBuffer(lpInputBuffer, dwInputBuffer, 3, 5);

  // generuj kod ustawiający klucze do odszyfrowania danych
  SetupDecryptionKeys();

  // generuj kod deszyfrujący
  GenerateDecryption();

  // ustaw wartości rejestrów wyjściowych
```

```
SPE_OUTPUT_REGS regOutput[] = { { eax, dwInputBuffer } };

SetupOutputRegisters(regOutput, _countof(regOutput));

// generuj epilog funkcji
GenerateEpilog(1L);

// wyrównaj rozmiar funkcji do wielokrotności 4 lub 16
AlignDecryptorBody(rnd_bin() == 0 ? 4L : 16L);

// koryguj wszystkie instrukcje korzystające
// z adresowania delta offset
UpdateDeltaOffsetAddressing();

// dołącz na końcu funkcji zaszyfrowany blok danych
AppendEncryptedData();

////////////////////////////////////////////////////////////
//
// zwolnij zasoby
//
////////////////////////////////////////////////////////////

// zwolnij pamięć zaszyfrowanego bloku danych
di_vfree(&diEncryptedData);

// zwolnij tablicę z pseudoinstrukcjami szyfrującymi
di_vfree(&diCryptOps);

////////////////////////////////////////////////////////////
//
// kopiuj kod funkcji polimorficznej do wyjściowego bufora
//
////////////////////////////////////////////////////////////

DWORD dwOutputSize = a.getCodeSize();

// generuj kod funkcji polimorficznej
// (łączy skoki i labele)
PVOID lpPolymorphicCode = a.make();

// struktura opisująca zaalokowany blok pamięci
DATA_ITEM diOutput;

// alokuj pamięć (wykonywalną) na bufor wyjściowy
di_valloc(&diOutput, dwOutputSize);

// sprawdź czy udało się zaalokować pamięć
if (diOutput.lpPtr != NULL)
{
  // kopiuj wygenerowany kod funkcji
  // deszyfrującej
  memcpy(diOutput.lpPtr, lpPolymorphicCode, dwOutputSize);

  // uaktualnij wskaźniki do kodu
  // w parametrach wejściowych
  *lpOutputBuffer = diOutput.lpPtr;
  *lpdwOutputSize = dwOutputSize;

  MemoryManager::getGlobal()->free(lpPolymorphicCode);
}
else
{
  MemoryManager::getGlobal()->free(lpPolymorphicCode);

  return CMutagen::MUTAGEN_ERR_MEMORY;
}
```

```
  ////////////////////////////////////////////////////////
  //
  // wyjście z funkcji
  //
  ////////////////////////////////////////////////////////

  return CMutagen::MUTAGEN_ERR_SUCCESS;
}
```

Test

Aby przetestować kod, zaszyfrujemy ciąg tekstowy naszym silnikiem i wywołany wygenerowany kod funkcji deszyfrującej. Należy pamiętać, aby pamięć, w której znajdzie się kod, była zaalokowana z odpowiednimi flagami wykonywania. W przeciwnym wypadku mechanizmy ochrony systemu operacyjnego, takie jak np. stosowany w Windows system *DEP* (ang. *Data Execution Prevention*), spowodują, że próba uruchomienia funkcji w pamięci bez praw do wykonywania kodu zakończy się wyjątkiem.

Listing 17. Testowanie silnika polimorficznego.

```
#include <conio.h>
#include "mutagen\mutagen.h"
#include "mutagen\mutagen_spe.h"

// deklaracja prototypu funkcji deszyfrującej
typedef DWORD(__stdcall *DecryptionProc)(PVOID);

int __cdecl main()
{
  // dane wejściowe (ciąg tekstowy, lecz
  // może to być dowolny inny bufor z
  // danymi)
  char szHelloWorld[] = "Hello world!";

  // nowa instancja silnika polimorficznego
  CMutagenSPE *speEngine = new CMutagenSPE();

  // tutaj znajdzie się wskaźnik do
  // wygenerowanej funkcji deszyfrującej
  PBYTE lpcDecryptionProc = NULL;

  // tutaj zostanie zapisany rozmiar
  DWORD dwDecryptionProcSize = 0;

  // szyfruj dane wejściowe i utwórz
  // dynamicznie wygenerowaną funkcję
  // deszyfrującą
  speEngine->PolySPE(reinterpret_cast<PBYTE>(szHelloWorld), \
             sizeof(szHelloWorld), \
             &lpcDecryptionProc, \
             &dwDecryptionProcSize);

  // zapisz wygenerowaną funkcję na dysku
  FILE *hFile = fopen("polymorphic_code.bin", "wb");

  if (hFile != NULL)
  {
    fwrite(lpDecryptionProc, dwDecryptionProcSize, 1, hFile);
    fclose(hFile);
  }
```

```
  // ustaw poprawny wskaźnik do funkcji
  // deszyfrującej
  DecryptionProc lpDecryptionProc =
reinterpret_cast<DecryptionProc>(lpcDecryptionProc);

  // bufor wyjściowy na odszyfrowane dane
  char szOutputBuffer[128] = { 0xCC };

  // wywołaj funkcję deszyfrującą z pamięci
  DWORD dwOutputSize = lpDecryptionProc(szOutputBuffer);

  // wyświetl odszyfrowany ciąg tekstowy,
  // powinien wyświetlić "Hello world!"
  printf(szOutputBuffer);


  return 0;
}
```

Jak wygląda wygenerowany kod?

Wygenerowany kod wygląda za każdym razem inaczej. To główny cel silnika polimorficznego. Poniżej prezentuję dwie różne funkcje deszyfrujące, które zostały wygenerowane naszym silnikiem.

Listing 18. Funkcja deszyfrująca.

```
; początek funkcji deszyfrującej (prolog)

    enter   0, 0


; zachowanie wrażliwych rejestrów (stdcall)

    push    esi

    push    ebx

    push    edi


; pobranie 1 parametru funkcji

    mov     ecx, [ebp+8]


    call    delta_offset


; nieużywane instrukcje

    xor     eax, eax

    leave

    retn    4
```

```
delta_offset:

; w EDX znajdzie się adres labela delta_offset
    mov     ebx, [esp]
    add     esp, 4

; korekta adresu wskaźnika do zaszyfrowanych danych
    add     ebx, 51h

; ustawianie klucza deszyfrującego
    mov     edx, 918D3D1Bh
    add     edx, 78170F2Ah

; ilość bloków do odszyfrowania
    mov     eax, 4

; pętla deszyfrująca
decryption_loop:

; pobranie bloku zaszyfrowanych danych
    mov     esi, [ebx]

; instrukcje deszyfrujące (losowe)
    xor     esi, 138E6781h
    sub     esi, edx
    not     esi
    xor     esi, edx

; zapis odszyfrowanego bloku do bufora wyjściowego
    mov     [ecx], esi
```

```
    add     ebx, 4
    add     ecx, 4
    dec     eax
    jnz     short decryption_loop

; ustawianie wartości zwracanej przez funkcję
; czyli rozmiaru odszyfrowanych danych
    mov     eax, 0Dh

; epilog funkcji i powrót do kodu wywołującego
    pop     edi
    pop     ebx
    pop     esi
    leave
    retn    4

; wyrównanie (składające się z instrukcji nop)
    db 5 dup(90h)

; zaszyfrowane dane
    db 0B6h, 044h, 052h, 0B0h, 09Bh, 087h, 05Eh, 0B1h
    db 08Ch, 04Bh, 06Ah, 0F3h, 07Eh, 0ACh, 0B5h, 057h
```

Listing 19. Funkcja szyfrująca w innym wariancie.

```
; początek funkcji deszyfrującej (prolog)
    push    ebp
    mov     ebp, esp

; zachowanie wrażliwych rejestrów (stdcall)
    push    edi
    push    ebx
    push    esi
```

```
; pobranie 1 parametru funkcji
    mov     esi, [ebp+8]

    call    delta_offset

; nieużywane instrukcje
    xor     eax, eax
    leave
    retn    4

delta_offset:

; w EDX znajdzie się adres labela delta_offset
    mov     ecx, [esp]
    add     esp, 4

; korekta adresu wskaźnika do zaszyfrowanych danych
    add     ecx, 4Ah

; ustawianie klucza deszyfrującego
    mov     ebx, 76F71EBFh
    sub     ebx, 50531439h

; ilość bloków do odszyfrowania
    mov     edi, 4

; pętla deszyfrująca
decryption_loop:

; pobranie bloku zaszyfrowanych danych
    mov     eax, [ecx]
```

```
; instrukcje deszyfrujące (losowe)
    neg     eax
    sub     eax, 4B1A7C17h
    not     eax
    xor     eax, ebx
    neg     eax

; zapis odszyfrowanego bloku do bufora wyjściowego
    mov     [esi], eax

    add     ecx, 4
    add     esi, 4
    dec     edi
    jnz     short decryption_loop
; ustawianie wartości zwracanej przez funkcję
; czyli rozmiaru odszyfrowanych danych
    mov     eax, 0Dh

; epilog funkcji i powrót do kodu wywołującego
    pop     esi
    pop     ebx
    pop     edi
    leave
    retn    4

; wyrównanie (składające się z instrukcji int3)
    db 5 dup(0CCh)

; zaszyfrowane dane (wyrównane do 4)
    db 028h, 014h, 01Dh, 06Ah, 001h, 059h, 012h, 06Bh
    db 0F2h, 01Ch, 025h, 0ADh, 070h, 0C2h, 07Ch, 0CAh
```

Co dalej?

Nasz silnik polimorficzny jest w tym momencie całkiem prosty, można go rozbudować o dodatkowe elementy, jak np.:

- generowanie instrukcji zaśmiecających, tzw. *junks* – są to zwykle bloki kodu utrudniające zrozumienie kodu pod debuggerem lub deasemblerem
- generowanie białego szumu – czyli instrukcji, które nie mają wpływu na działanie funkcji (np. dodawanie do losowego rejestru wartości liczbowej, a później jej odejmowanie)
- generowanie równoznacznych instrukcji (mutacji) w różnych wariantach, połączonych losowym porównaniem i skokami
- generowanie dodatkowych funkcji pomocniczych, np. zwracające wartości, które pierwotnie są ustawiane bezpośrednio w kodzie głównej funkcji
- zmiana systemu przekazywania wartości, np. z konwencji *stdcall* na *cdecl* (lub na dowolnie inną)
- modyfikacja kodu, aby instrukcje nie były wykonywane linearnie
- i najbardziej zaawansowane – wielowarstwowe szyfrowanie, czyli wygenerowany kod zawierający kolejno zaszyfrowane warstwy deszyfrujące

Całość można bardzo rozbudować, zwłaszcza korzystając z takiej biblioteki jak *AsmJit*, która w prosty sposób udostępnia interfejsy do budowania dynamicznego kodu assemblera z wykorzystaniem wszystkich obecnych rozszerzeń procesorów z rodziny x86 i x64.

Źródła

- Blokowe algorytmy szyfrowania - http://en.wikipedia.org/wiki/Block_cipher
- Strumieniowe algorytmy szyfrowania - http://en.wikipedia.org/wiki/Stream_cipher
- Program PGP - http://en.wikipedia.org/wiki/Pretty_Good_Privacy
- Algorytm RSA - http://en.wikipedia.org/wiki/RSA_(algorithm)
- Konkurs firmy RSA na łamanie kluczy - http://en.wikipedia.org/wiki/RSA_Factoring_Challenge
- Szyfrowanie ElGamal - http://en.wikipedia.org/wiki/ElGamal_encryption
- Szyfrowanie z wykorzystaniem krzywych eliptycznych - http://en.wikipedia.org/wiki/Elliptic_curve_cryptography

- Narzędzie do faktoryzacji kluczy publicznych RSATool - http://www.woodmann.com/collaborative/tools/index.php/RSA-Tool_2
- Narzędzie do faktoryzacji Msieve - http://sourceforge.net/projects/msieve/
- Narzędzie ECCTool - http://www.woodmann.com/collaborative/tools/index.php/ECCTool
- Framwork Metasploit - http://www.metasploit.com/
- Historia kodu silników polimorficznych - http://en.wikipedia.org/wiki/Polymorphic_code
- Śledztwo w sprawie wirusa Virut - http://krebsonsecurity.com/2013/01/polish-takedown-targets-virut-botnet/
- Mechanizm Data Execution Prevention (DEP) - http://en.wikipedia.org/wiki/Data_Execution_Prevention
- Szczegółowy opis ramki stosu - http://en.wikipedia.org/wiki/Call_stack
- Biblioteka AsmJit - http://code.google.com/p/asmjit/
- Biblioteka JitAsm - http://code.google.com/p/jitasm/

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.