Praktyka

Praktyczna aplikacja do analizy Malware

Rubén Santamarta

stopień trudności

Dzień za dniem badacze malware, analitycy sądowi czy administratorzy muszą stawiać czoła zagrożeniu bezpieczeństwa systemów informatycznych. Ich celem może być wyjaśnienie nieautoryzowanych ingerencji, ochrona użytkowników przed wirusem lub unikanie wystawiania sytemu na niebezpieczeństwo. Aby osiągnąć te cele, konieczna jest maksymalnie szczegółowa analiza działania złośliwego oprogramowania, z którym mamy doczynienia; tu właśnie do gry wkracza inżynieria wsteczna (ang. reverse engeenering).

Twórcy malware (wirusów, trojanów, rootkitów) próbują utrudnić tę analizę do maksimum, przy użyciu między innymi technik przeciwdziałających debugowaniu, technik polimorficznych, techniki stealth lub programów do kompresji plików; te ostatnie oprócz tego, że zmniejszają rozmiar plików wykonywalnych, jednocześnie dodają bardziej lub mniej skomplikowaną dodatkową warstwę ochronną.

W takich sytuacjach liczy się przede wszystkim czas, nie ma wątpliwości, że dzięki spokojnej i wyczerpującej analizie, prędzej czy później, osiągniemy nasz cel i poznamy wszystkie szczegóły zagrożenia. Niestety, zdarzają się sytuacje, w których nie dysponujemy odpowiednią ilością czasu i należy wtedy zoptymalizować działania związane z przeprowadzaniem analizy. Wobraźmy sobie robaka wykorzystującego nie znany błąd programu, aby rozesłać się za pomocą internetu, czas zainwestowany w analizę i zrozumienie funkcjonowania tego robaka wyznaczy granicę pomiędzy prawdziwą katastrofą dla użytkowników, a zneutralizowanym i zmniejszonym zagrożeniem.

Z tego też względu powinniśmy uzbroić się w środki wystarczające do rozwiązania jakiejkolwiek trudności, z jaką się zetkniemy.

Hooking

Jak można zauważyć, istnieje bardzo wiele sztuczek, których celem jest utrudnienie nam użycia debuggera (zarówno w Ring0, jak i w Ring3), będącego podstawowym narzędziem inżynierii wstecznej. Z tego powodu powinni-

Z artykułu dowiesz się...

- Jak stosować hooking do analizy złośliwego oprogramowania - malware
- Jak zastosować Structure Exception Handling do stworzenia disassamblera rozmiaru.

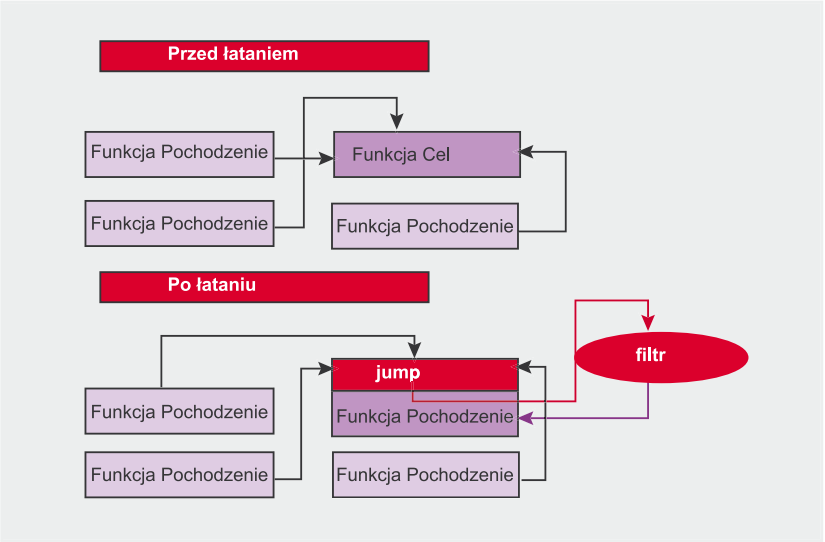
Powinieneś wiedzieć...

- znać Asembler x86 y C
- mieć znajomość Win32 Api oraz Structure Exception Handling
- mieć podstawową znajomość technik stosowanych przez malware i wirusy.

Techniki stosowane przeciwko dissasamblerom i debuggerom

Przez lata twórcy programów typu malware,autorzy wirusów czy nawet sami programiści oprogramowania komercyjnego wyposażali swoje dzieła w metody anti-debug oraz anti-disasm. Większość z nich jest przeznaczona do wykrywania, czy dany program jest obserwowany przez debugger. W przypadku potwierdzenia tej sytuacji podejmowane przez program działania mogą być bardzo różne, zaczynając od gwałtownego przerwania uruchamiania, a kończąc na ponownym uruchomieniu komputera, czy nawet czymś jeszcze bardziej agresywnym, co na szczęście jest mało popularne.

- Starym trikiem stosowanym (jeszcze ciągle) do wykrycia obecności SoftIce, najbardziej znanego debuggera Ring0, używanego na całym świecie w inżynierii wstecznej, była próba uzyskania dostępu do mechanizmów utworzonych przez jeden z jego sterowników - Ntlce.
- Instrukcja w Asemblerze x86 RDTSC: skrót mnemoniczny od Read Time−Stamp Counter. Instrukcja ta przechowuje w EDX:EAX (64 bity) wartość timestamp procesora. Wyobraźmy sobie, że RDTSC uruchamiana jest na początku bloku kodu, a wartość zwrotna jest magazynowana. Na końcu tego bloku kodu ponownie uruchamiamy RDTSC i odejmujemy uzyskaną wartość od wcześniej zachowanej. W normalnych warunkach uruchamiania, wynik tej operacji można określić w racjonalnych wartościach, oczywiście będzie miała na to wpływ szybkość i obciążenie procesora, ale jeżeli oczyścimy ten blok kodu, wzrost timestamp między obydwoma odczytami będzie bardzo rozstrzelony w stosunku do tego co odkrylibyśmy w debuggerze.
- Manipulowanie przerwami, aby zmienić strumień kodu. Potężną właściwością architektury Win32 jest funkcja Structure Exception Handling (SEH), która pozwala nam ustalić schematy funkcji callback, w celu kontrolowania wyjątków. Ze względu na to, że debuggery zwykle zajmują się jakimkolwiek wyjątkiem, który powstał w czasie pracy programu, nigdy nie zostanie zastosowany sposób postępowania ustalony przez programistę do obsługi wyjątków. Załóżmy, że oparliśmy strumień naszego programu na procedurze tego typu, jeżeli po umyślnym sprowokowaniu wyjątku (przy użyciu np.: xor eax,eax a następnie mov [eax],eax), nie dotrzemy do założonego obszaru kodu, prawdopodobnie znajdujemy się pod obserwacją debuggera.
- Inne mniej dopracowane sztuczki bazują na swoistych właściwościach każdego debuggera. Czy to poprzez próbę odnalezienia określonych rodzajów lub tytułów okien zarejestrowanych przez program, czy po prostu poprzez szukanie kluczy w rejestrze Windows, które mogłyby go zadenuncjować.



Rysunek 1. *Podstawowy schemat Inline Hooking*

śmy opracować metodę, która w odpowiednich warunkach, pozwoliłaby nam oddziaływać na zachowanie pliku wykonywalnego, który badamy i modyfikować go.

Jedną z częściej używanych technik, służących do osiągnięcia tego celu jest hooking.

Moglibyśmy krótko sklasyfikować różne techniki zahaczania z uwzględnieniem miejsca, w którym powstaje. Każdy rodzaj nastawiony jest na różne aplikacje. W ten sposób otrzymujemy następujące rodzaje:

- Inline Hooking
- Import Address Table hooking
- System Service Table hooking (Ring0)
- Interrupt Descriptor Table hooking (Ring0)
- IRP hooking (Ring0)
- Filter drivers (NDIS,IFS...Ring0)

Metodą, którą zastosujemy będzie Inline Hooking. Powodem zastosowania tej techniki jest fakt, że przy jej użyciu bezpośrednio łatamy funkcję, którą chcemy przechwycić, kiedy jest ona załadowana w pamięci. W ten sposób nie martwimy się, z którego miejsca pochodzą odwołania lub ile razy to nastąpiło i bezpośrednio atakujemy rdzeń. Jakiekolwiek odwołanie do funkcji zostanie przechwycone przez nasz hak.

Przechwytywanie i modyfikowanie strumienia kodu

Załóżmy, że chcemy przechwycić wszystkie wywołania *API CloseHandle*, powstające w trakcie uruchomiania programu. Wspomniany API znajduje się w kernel32.dll, spójrzmy jakie są jego pierwsze instrukcje:

```
01 8BFF    mov  edi,edi
02 55      push ebp
03 8BEC    mov  ebp,esp
04 64A118000000
           mov  eax,fs:[00000018]
05 8B4830 mov  ecx,[eax][30]
06 8B4508 mov  eax,[ebp][08]
```

Ten blok kodu przedstawia pierwsze bajty punktu wejścia *CloseHandle*, to znaczy, że jakiekolwiek wywołanie wspomnianej funkcji nieuchronnie uruchomi poprzedni kod. Stosując się do schematu *Inline Hooking*, zastąpimy te pierwsze instrukcje naszym własnych hakiem, który zmodyfikuje normalny strumień funkcji w kierunku filtra.

Różne możliwości dla tego samego celu

Metody zmiany biegu strumienia w kierunku naszego kodu mogą być różne. Najprostsza ze wszystkich to zmiana pierwszych bajtów CloseHandle jednym skokiem bezwarunkowym.

```
01 E9732FADDE     jmp  0DEADBEEF
02 64A118000000   mov  eax,fs:[00000018]
```

Nadpisaliśmy 5 pierwszych bajtów jednym skokiem do adresu 0xDEADBEEF, oczywiście ten adres jest nieważny, jako że pracujemy w trybie użytkownika. Pod tym adresem znalazłby się kod, który wprowadziliśmy w przestrzeń adresową pliku wykonywalnego.

Jako że jest to najzwyklejszy sposób, jest on jednocześnie najłatwiejszy do wykrycia, ze względu na to, że jest wysoce podejrzane, kiedy entry point funkcji systemowej zawiera bezwarunkowy skok do innego adresu w pamięci. Możemy użyć innej opcji, kombinacji `PUSH` + `RET`.

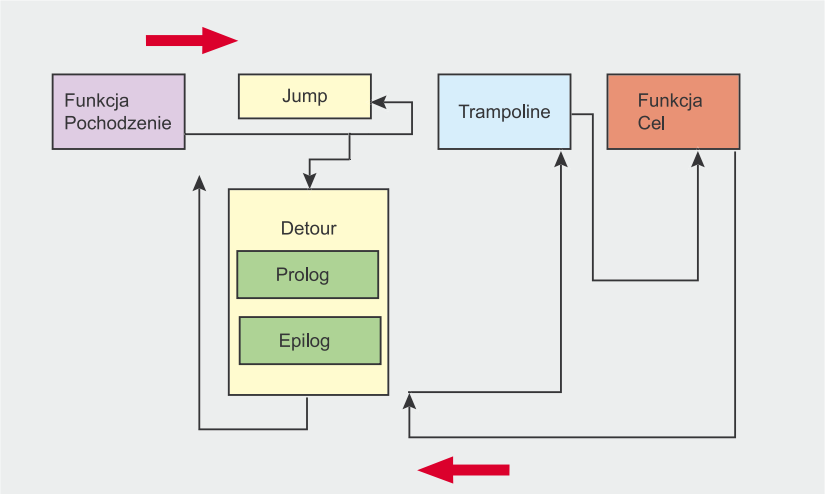
Tabla 1. *Dane dostępne dla obsługi wyjątków, jeżeli jest ona aktywna.*

W	Dane
ESP+4	Wskaźnik struktury EXCEPTION_RECORD
ESP+8	Wskaźnik struktury ERR
ESP+C	Wskaźnik struktury CONTEXT_RECORD

```
01 68EFBEADDE push 0DEADBEEF
02 C3         retn
03 A118000000 mov  eax,[00000018]
```

W tym przypadku nadpisujemy pierwszych 6 bajtów. Jeżeli przyjrzymy się uważnie, zdamy sobie sprawę z tego, że oryginalny kod *CloseHandle* istotnie się zmienił, ale nie tylko ze względu na dodane instrukcje, ale także dlatego, że niektóre z już istniejących zostały utracone, ponieważ zostały nadpisane, a kolejne stały się całkowicie różne. Jawi się to jako zagadnienie do poważnego rozważenia z uwagi na to, że wprawdzie osiągnęliśmy nasz cel, którym było przechwycenie wszystkich wywołań funkcji, to także modyfikacja jej kodu źródłowego była na tyle istotna, żeby spowodować anomalie w zachowaniu, co w rezultacie z całą pewnością doprowadzi do niespodziewanego zakończenia programu przy pierwszym wywołaniu *CloseHandle*.

Dlatego też konieczne jest rozwinięcie techniki możliwie najmniej agresywnej w stosunku do kodu źródłowego, która pozwoliłaby "zahaczanej" funkcji na normalne funkcjonowanie, a jednocześnie była stale pod kontrolą. Technikę tę znamy pod nazwą Detour (zaprezentowana przez Galena Hunta i Douga Brubachera z laboratoriów Microsoft).



Rysunek 2. *Podstawowy schemat techniki Detour*

Detour

W stosunku do Inline hooking, technika Detour wprowadza dwie nowe koncepcje: Funkcję Detour i funkcję Trampoline.

- *Funkcja Detour* powinna składać się z pierwszej części, w której realizowane są pierwsze operacje na otrzymanych danych, następnie odwołania do funkcji Trampoline i na koniec część kodu, który zostanie uruchomiony po zakończeniu funkcji Trampoline.
- *Funkcja Trampoline* zawiera zarówno instrukcje funkcji docelowej, całkowicie nadpisane przez bezwarunkowy skok (JUMP), jak i te, które zostały częściowo zmienione. Dalej nastąpi skok do kolejnej instrukcji, która odpowiadałaby funkcji docelowej.

W ten sposób rozwiązaliśmy problem utraconych lub zmienionych instrukcji, który powstał przy *Inline Hooking*. Kluczem jest ocalenie tych instrukcji w *Funkcji Trampoline*, aby mogły one zostać uruchomione. Następnie przeskoczymy do kolejnej instrukcji, gdzie funkcja docelowa będzie działać normalnie. Po zakończeniu funkcji docelowej, odzyskamy kontrolę nad zakończeniem *Funkcji Detour*. Ma ona możliwość odtworzenia ścieżki uruchomieniowej, przywracając kontrolę funkcji źródłowej lub możliwość realizacji innego rodzaju operacji.

A więc, skąd wiedzieć, ile instrukcji powinniśmy skopiować do *Funkcji Trampoline* z funkcji docelowej? Każda funkcja docelowa będzie inna, z powodu czego nie możliwe jest skopiowanie ustalonej ilości bajtów, ponieważ moglibyśmy przycinać in-

Tabla 2. *Pola struktury EXCEPTION_RECORD*

Offset	Dane
+ 0	ExceptionCode
+ 4	ExceptionFlag
+ 8	NestedExceptionRecord
+ C	ExceptionAddress
+ 10	NumberParameters
+ 14	AdditionalData

strukcje. Problem ten rozwiązuje się przy użyciu disassemblera rozmiaru.

Disassemblery rozmiaru

Disassemblery rozmiaru różnią się od zwykłych disassemblerów tym, że ich jedyną misją jest uzyskanie długości instrukcji, a nie przedstawienie ich. Ten rodzaj disassemblerów był tradycyjnie używany przez wirus *cavity*, polimorfizmy itp. Dlatego, że dwa z disassemblerów rozmiarów (prawdziwe perełki ekstremalnej optymalizacji), które są najczęściej używane, zostały zaprogramowane przez znanych twórców wirusów: *Zombie* i *RGB*.

Oparte są one na statycznym disassemblingu instrukcji. Używają do tego tabel kodów operacji architektury na której działają, w tym przypadku x86.

Oprócz użycia ich do tworzenia skomplikowanych wirusów, wykorzystywane są także do hookingu, ze względu na to, że dzięki nim możliwe jest rozwiązanie wcześniej omawianego problemu.

Dlatego, opierając się na potencjale Structure Exception Handling, zostanie wyjaśniona innowacyjna technika tworzenia dynamicznych disassemblerów rozmiaru. Zatem do dzieła.

Stosowanie Structure Exception Handling (SEH)

Należy zadać sobie pytanie, jakie informacje może przekazać nam SEH. Po pierwsze powinniśmy się skupić na cechach charakterystycznych problemu, dla którego chcemy opracować rozwiązanie.

- Pierwsze instrukcje funkcji docelowych, z reguły nie różnią się zasadniczo, ale na tyle jednak, by konieczne było indywidualne rozpatrzenie każdego przypadku.
- Skok bezwarunkowy (jmp) lub Push + ret nie zajmie więcej niż 6 bajtów. Trzeba będzie zanalizować maksymalnie 4 do 5 instrukcji.
- Pierwsze instrukcje z reguły wykonują operacje związane z dostosowywaniem stosu.
- Zamysł polega na tym, aby zdołać uruchomić te pierwsze instrukcje w kontrolowanym środowisku, co pozwoli nam policzyć ich długość.

Aby wyczuć to, w jaki sposób mamy skonstruować to środowisko, należy wprowadzić informacje, które przynosi SEH.

Dla każdego wyjątku, który miał miejsce w kodzie chronionym przez funkcję SEH zdefiniowaną dla wątku, wyznaczona obsługa ma do dyspozycji następujące dane.

Kiedy powstał wyjątek, system uruchamia obsługę wyjątków, aby określiła ona, co dalej zrobić. W tym momencie `esp` wskazuje różne struktury.

W strukturze EXCEPTION_RECORD zwracamy uwagę na pola ExceptionCode i ExceptionAddress

- ExceptionCode identyfikuje rodzaj powstałego wyjątku. W systemie ustalone są różne kody dla każdego typu, dodatkowo możliwe jest zdefiniowanie naszych własnych kodów, aby zindywidualizować wyjątki poprzez API RaiseException.
- ExceptionAddress to adres pamięci należący do instrukcji, wytworzonej przez wyjątek, równałaby się rejestrowi eip w momencie powstania wyjątku.

Inną podstawową strukturą, którą należy poznać jest CONTEXT. Ta struktura będzie zawierać wartości wszystkich rejestrów w momencie powstania wyjątku.

Musimy pamiętać o tym, że zawsze najważniejsza jest możliwość kontrolowania każdej uruchomionej instrukcji, tak jak w przypadku postępowania krok po kroku z debuggerem. Rzeczywiście zastosujemy nasz disassembler rozmiaru, podstawowe działanie debuggera.

Kody wyjątków

Nie można traktować w taki sam sposób wyjątku powstałego na skutek dostępu do nieważnej pozycji pamięci, jak wyjątku powstałego na skutek dzielenia przez 0. W związku z tym system jednoznacznie identyfikuje każdą sytuację, aby ułatwić pracę obsłudze wyjątków. Kilka najbardziej rozpowszechnionych kodów wyjątków wymieniono poniżej:

- C0000005h – Naruszenie praw dostępu do operacji czytania lub pisania.
- C0000017h – Brak wolnej pamięci.
- C00000FDh – Stack Overflow

Poniższe kody są kluczowe dla naszego projektu:

- 80000003h – Breakpoint wygenerowany przez instrukcję `int 3`
- 80000004h – Single step wygenerowany przez aktywację Trap Flag w rejestrze EFLAGS

Programowanie disassemblera rozmiaru

Po pierwsze należy zdefiniować środowisko, w którym uruchomimy nadzorowane funkcje; w ten sposób wywołamy instrukcje należące do funkcji docelowej, których długość chcemy poznać, aby móc następnie skopiować je w całości do Funkcji Trampoline.

W tym celu po pierwsze należy określić zakres SEH, gdzie SEH_SEHUK będzie rutynowym schematem obsługującym pojawiające się wyjątki.

Tabela 3. *Pola CONTEXT należące do rejestrów ogólnych i kontrolnych*

Offset	Rejestr
+ 9C	EDI
+ A0	ESI
+ A4	EBX
+ A8	EDX
+ AC	ECX
+ B0	EAX
+ B4	EBP
+ B8	EIP
+ BC	CS
+ C0	EFLAGS
+ C4	ESP
+ C8	SS

```
01 push  dword SEH_SEHUK
02 push  dword [fs:0]
03 mov   [fs:0], esp
```

Od tego momentu cały kod uruchomiony po tych instrukcjach będzie chroniony. Kolejnym etapem jest skopiowanie określonej ilości bajtów, które będą zawierać nadzorowane instrukcje, z funkcji docelowej do zarezerwowanego obszaru wewnątrz naszego kodu. Jego rozmiar może się zmieniać. W tym przypadku wybraliśmy 010h, jako że jest wystarczająco szeroki, by pomieścić w całości pierwsze instrukcje.

```
04 mov  esi,TargetFunction
05 mov  edi,Code
06 push 010h
07 pop  ecx
08 rep  movsb
```

Po dotarciu do tego punktu, pozostaje nam tylko jeden etap, zanim rozpoczniemy uruchamianie nadzorowanych instrukcji. Najpierw to zobaczmy:

```
09 int 3
10 Code:
11 ModCode
   times 12h db (90h)
```

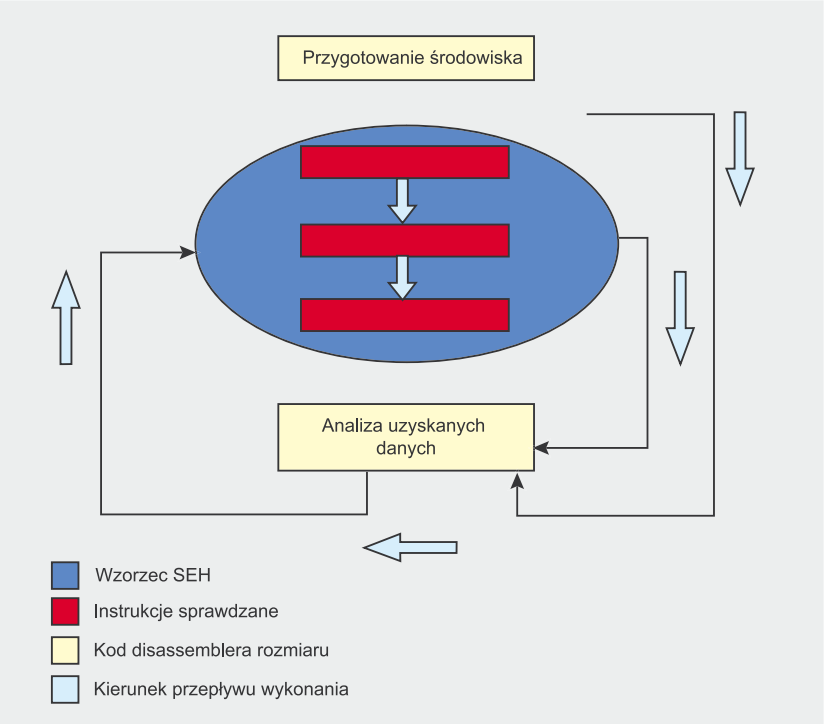
ModCode to obszar do którego skopiowaliśmy nasze nadzorowane instrukcje. Zauważyliśmy, że tuż przed osiągnięciem tego punktu, umieściliśmy pewną `int 3`. Dlaczego? Obliguje nas do tego wiele powodów:

- Jak wcześniej wspominaliśmy, pierwsze instrukcje jakiejkolwiek *funkcji docelowej* z reguły realizują operacje modyfikacji stosu. Z tego względu powinniśmy upewnić się, że nasz stos nie zostanie zniszczony przez te działania. Po uruchomieniu `int 3` wygenerujemy wyjątek, który posłuży nam do wejścia w *SEH_SEHUK* (nasza obsługa). W ten sposób wchodząc w strukturę *CONTEXT*, ocalimy rejestry *ESP* i *EBP* w celu przywrócenia stanu naszego stosu do poprzednich wartości, po zakończeniu analizy.
- Aktywowanie *Trap Flag* w rejestrze *EFLAGS*. Dzięki tej technice sprawimy, ze po uruchomieniu następnej instrukcji, zostanie automatycznie wygenerowany wyjątek *Single Step*. Dzięki czemu zostanie przywrócona kontrola. W ten sposób otrzymaliśmy tę sama informację, którą byśmy uzyskali oczyszczając program krok po kroku.

Zobaczmy te punkty utworzone w kodzie assemblera

```
12 SEH_SEHUK:
13 mov        esi, [esp + 4]
      ; EXCEPTION_RECORD
14 mov        eax, [esi]
      ; ExceptionCode
15 test       al,  03h
      ; Int3 Exception Code
16 mov        eax, [esi + 0Ch]
      ; Eip Exception
17 mov        esi, [esp + 0Ch]
      ; CONTEXT record
18 mov        edx, [esi + 0C4h]
      ; Esp Exception
19 jz         Int1h
20 mov        eax, [esi + 0B4h]
      ; Ebp Exception
21 mov        [OrigEbp], eax
22 mov        [OrigEsp], edx
23 inc        dword [esi + 0B8h]
      ; Eip++ (Int3->Weitere Anweisung)
24 mov        eax,Code
25 mov        [PrevEip],eax
26 jmp        RetSEH
```

Widzimy, że w wersie 15 porównujemy al z 03 aby spróbować ustalić, czy wyjątek sprowokowała `int 3` (kod wyjątku *80000003h*), czy wręcz przeciwnie, chodzi o wyjątek spowo-



Rysunek 3. *Schemat działania naszego disassemblera rozmiaru*

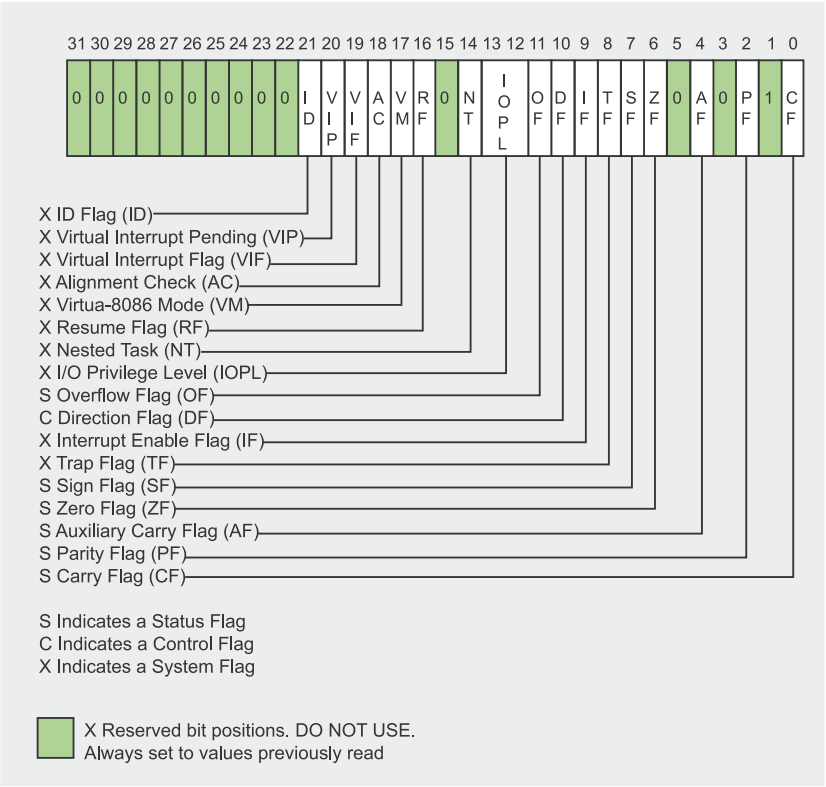
Listing 1. *Obserwacja kodu*

```
27 Int1h:
28 mov  ecx, eax
29 sub  eax, [PrevEip]
30 cmp  ax, 10h
31 jb   NoCall
32 mov  ebx, dword [edx]
33 mov  edx,ebx
34 sub  ebx, [PrevEip]
35 cmp  bl, 7
36 jbe  HabemusCall
37 mov  edi,[PrevEip]
38 inc  edi
39 inc  edi
40 mov  dword [esi + 0B8h], edi
41 mov  ecx,edi
42 jmp  NoCall
43 HabemusCall:
44 mov  dword [esi + 0B8h], edx
45 mov  ecx,edx
46 NoCall:
47 mov  [PrevEip], ecx
48 sub  ecx, Code
49 cmp  ecx, [HookLength]
50 jge  Success
51 RetSEH:
52 or   word [esi + 0C0h], 0100h
     ;Das Trap Flag aktivieren
53 xor  eax,eax
54 ret
55 Success:
56 mov  [LenDasm], ecx
     ;Die Länge der
     Anweisung
     wiederherstellen
57 mov  esp, [OrigEsp]
     ;Esp wiedergewinnen
58 mov  ebp, [OrigEbp]
     ;Ebp wiedergewinnen
59 pop  dword [fs:0]
     ;Den SEH-Bereich reinigen
60 add  esp, 4
     ;Den Stack anpassen
```

dowany przez *Trap Flag* czy jakiekolwiek inne wydarzenie. W przypadku, gdy mamy do czynienia z wyjątkiem *BreakPoint* zastosujemy czynności wytłumaczone wcześniej; wersy 20,21,22.

Konieczne jest zmodyfikowanie *EIP* kontekstu, aby w momencie przywracania kontroli w systemie, wskazał kolejną instrukcję po `int 3`, bo inaczej znaleźlibyśmy się w zamkniętym kręgu. Aby tego uniknąć, jak widzimy w wersie 23, po prostu zwiększamy o jeden jej wartość. Związane jest to z tym, że kod operacji dla `int 3` ma rozmiar 1 bajta.

W wersie 25 zapisujemy adres pamięci, gdzie zaczynają się nasze *instrukcje nadzorowane*, pomoże nam to w momencie emulowania wywołań i skoków warunkowych lub bezwarunkowych.



Rysunek 4. *Rejestr EFLAGS*

Analiza nadzorowanych instrukcji

Do tej pory wszystko co widzieliśmy można by zawrzeć w bloku przygotowanie środowiska. Rozpoczniemy obserwację kodu przynależnego do analizy nadzorowanych instrukcji.

Cel

Celem tej części kodu jest analiza długości nadzorowanych instrukcji aż do momentu odnalezienia wartości wyższej lub równej tej, która zajęłaby nasz hak, niezależnie, czy byłby to bezwarunkowy skok (`jmp` 5 bajtów) czy `push`+ `ret` (6 bajtów). Wobraźmy sobie na przykład, że rodzaj naszego haka to `push`+`ret` i staramy się zahaczyć *CloseHandle*. Wskażemy naszemu disassemblerowi rozmiaru, że nasz hak zajmuje 6 bajtów (`HookLength = 6`). Wtedy zacznie liczyć długość pierwszej instrukcji

```
01 8BFF  mov edi,edi
```

Rozmiar 2 bajty. Jako że jest mniejszy niż 6 kontynuuje jak następuje

```
02 55  push ebp
```

Rozmiar 1 bajt +2 bajty poprzedniej instrukcji = 3 bajty. W dalszym ciągu mniejszy niż 6.

```
03 8BEC  mov ebp,esp
```

Rozmiar 2 bajty +3 bajty z poprzednich = 5 bajtów. Kontynuujemy

```
04 64A118000000  mov eax,fs:[00000018]
```

Rozmiar 6 bajtów +5 bajtów z poprzednich = 11 bajtów. Gotowe!

Nasz disassembler rozmiaru zwróci nam jedenaście. Co to oznacza? Tyle, że na hak złożony z sześciu bajtów, liczba bajtów, które należy skopiować z funkcji docelowej do Funkcji Trampoline, aby nie utracić ani nie uciąć żadnej instrukcji, to jedenaście.

Analiza danych

Tutaj mamy początkowy blok analizy. Aż do wersu 46 stykamy się z algo-

rytmem, który pozwala nam emulować wywołania i skoki. Algorytm ten polega na sprawdzaniu odległości pomiędzy *EIP*, w którym powstał wyjątek, a wcześniejszą wartością rejestru. W przypadku, gdy jest to dość znaczna odległość, mamy doczynienia z wywołaniem lub skokiem, dlatego też przejdziemy do odzyskiwania kontekstu, aby wskazał na następną nadzorowaną instrukcję, zamiast prowadzić proces dalej od adresu, do którego zaprowadził nas skok lub wywołanie, zsumujemy także w naszym liczniku bajty które zajmuje ta instrukcja.

W wersie 47, 48, 49 sprawdzane jest, czy mamy zanalizowaną wystarczającą ilość instrukcji, aby odpowiednio umieścić nasz hak.

Kluczową częścią disassamblera są kolejne wersy 52, 53 i 54. W nich aktywujemy *Trap Flag* ustawiając na 1 odpowiedni bit w rejestrze *EFLAGS*. Jest do podstawa do oczyszczania krok po kroku.

Z mniej niż 256 bajtów zbudowaliśmy całkowicie działający disassembler rozmiaru. Myślę, że jesteśmy gotowi, żeby zastosować go w praktyce.

Listing 2. *ACPInject*

```
#include <stdio.h>
#include <windows.h>
typedef BOOL (WINAPI *PQUEUEAPC)(FARPROC,HANDLE,LPDWORD);
int main(int argc, char *argv[])
{
PROCESS_INFORMATION strProces;
STARTUPINFOA        strStartupProces;
PQUEUEAPC           QueueUserApc;
DWORD                                   MessageAddr,Ret1,Ret2,Długość;
char                                            *szExecutableName;
unsigned char       Snippet[]=  "\x90" /* nop */
   "\x6A\x00"  /* push NULL  */
   "\x6A\x00"  /* push NULL  */
   "\x6A\x00"  /* push NULL  */
   "\x6A\x00"  /* push NULL  */
   "\xB9\x00\x00\x00\x00"
              /* mov ecx,MessageBox*/
   "\xFF\xD1" ;
/* Call ecx   */
Długość = (DWORD) strlen( "c:\\windows\\system32\\calc.exe") + 1;
ZeroMemory( &strStartupProces, sizeof( strStartupProces) );
strStartupProces.cb = sizeof( strStartupProces );
ZeroMemory( &strProces, sizeof( strProces) );
szExecutableName = (char*) malloc( sizeof(char) * Długość);
if( szExecutableName  )  strncpy(szExecutableName,"c:\\windows\\system32\\
                    calc.exe",Długość);
else  exit(0);
_QueueUserApc = (PQUEUEAPC)GetProcAddress( GetModuleHandle ( "kernel32.dll"
                    ), "QueueUserAPC");
MessageAddr = (DWORD) GetProcAddress ( LoadLibraryA( "user32.dll") ,
                    "MessageBoxA" );
// U32!MessageBoxA
*( DWORD* )( Snippet + 10 ) = MessageAddr;
  Ret1 = CreateProcessA( szExecutableName,   NULL, NULL, NULL,
                0, CREATE_SUSPENDED,
                NULL,NULL,
                &strStartupProces,&strProces);
 Ret2 = (DWORD) VirtualAllocEx( strProces.hProcess,NULL,sizeof(Snippet),
                  MEM_COMMIT,
                   PAGE_EXECUTE_READWRITE);

  WriteProcessMemory(strProces.hProcess,(LPVOID)Ret2,  Snippet,
                    sizeof(Snippet), NULL);
 _QueueUserApc((FARPROC)Ret2,strProces.hThread,NULL);
  ResumeThread(strProces.hThread);
 return 0;
}
```

Zastosowanie w praktyce do analizy malware

Powszechnie do analizy malware z karty stosuje się rożne techniki, oto przykłady:

- Dla naszych celów stworzymy program, który wprowadzi i uruchomi kod w pliku wykonywalnym, który podamy mu jako parametr. Techniki wprowadzania kodu do procesów są dobrze znane. Istnieją różne ich rodzaje, ale wszystkie opierają się praktycznie na tych samych API.
- *VirtualAllocEx* dla zarezerwowania przestrzeni w pamięci w procesie. W tę przestrzeń zostanie wprowadzony kod. Użyjemy do tego *WriteProcessMemory*.
- W momencie uruchamiania kodu możemy wybrać pomiędzy *CreateRemoteThread* lub *SetThreadContext*.

Ale my nie użyjemy żadnej z tych form, ale zupełnie nowej: *QueueUserAPC*.

Cele zastosowania

Ten mały program wprowadza do kalkulatora windows mały kod, który powoduje wyświetlenie *Message Box*. Kalkulator także się nie pojawi po uruchomieniu tego kodu. Wyobraźmy sobie, że zamiast nieszkodliwego kodu, wprowadzony zostaje złośliwy kod przynależny do jakiegoś robaka. Wyobraźmy sobie także, że ten mały program został spakowany za pomocą programu, który zawiera wiele rodzajów ochrony *anti-debug* oraz *anti-disasm*. Musimy szybko dowiedzieć się, jak udaje mu się wniknąć do innego pliku wykonywalnego i jaki kod wprowadza. Do tego wszystkiego potrzebowalibyśmy pilnie dokonać disassemblingu pliku wykonywalnego, ale jak już powiedzieliśmy, zawiera packer, który spowalnia naszą analizę, dlatego

Listing 3. *Narzędzie do zahaczania API*

```
unsigned char HookCode[]=  "\x90"
      /* nop                */
   "\x68\x38\x03\x00\x00"
      /* push 3E8h          */
   "\x6A\x6E"
      /* push 6Eh           */
   "\xB9\x00\x00\x00\x00"
      /* mov ecx,00h        */
   "\xFF\xD1"
      /* Call K32!Beep      */
   "\x68\x00\x00\x00\x00"
      /* push dword 00h     */
   "\xB9\x00\x00\x00\x00"
      /* mov ecx,00h        */
   "\xFF\xD1"
      /* Call K32!Sleep     */
   "\x90\x90\x90\x90"
      /* Der Raum für
         die überwachten
         Anweisungen  */
   "\x90\x90\x90\x90"
   "\x90\x90\x90\x90"
   "\x90\x90\x90\x90"
   "\x90\x90\x90\x90"
   "\x68\x00\x00\x00\x00"
      /* push dword 00h     */
   "\xC3"
      /* ret                */
   "\x90";
      /* nop                */
unsigned char ExitHook[]=
   "\x68\x00\x00\x00\x00"
      /* push dword 00h     */
   "\xC3";
      /* ret                */
```

że nie można łatwo użyć debuggera. Nie pozostaje na tyle długo w pamięci, aby można było go rozpakować i zrzucić na dysk jakimś narzędziem do dekompresji i zrzucania procesów (*ProcDump*...). Faktycznie plik wykonywalny, tam gdzie wnika, pozostaje zaledwie przez dziesiąte sekundy w pamięci, uniemożliwiając także zrzucenie jego obrazu. Co możemy w związku z tym zrobić?.

Rozwiązanie przeszłoby przez zahaczenie *ExitProcessu* i w jakiś sposób zamrożenie go w pamięci do procesu (przy użyciu *Sleep*) i utrzymanie go w tym stanie odpowiednio długo, aby można było go zrzucić, a następnie zrekonstruować zrzut binarny w celu dokonania jego disassemblingu i oczyszczenia do stanu normalnego. Dlaczego zahaczyć *ExitProcess*?. 90% spakowanego malware po dotarciu do *ExitProcess* zostaje całkowicie rozpakowane w pamięci. Możemy natknąć się na wyjątki, w których jedynie pewna część pliku wykonywalnego rozpakowuje się, ale nie zdarza się to zbyt często, jako że jest to bardzo trudne do zaprojektowania. Skoncentrujemy sie na zbudowaniu narzędzia, które pozwoliłoby nam szybko zahaczyć jakiekolwiek API jakiejkolwiek dll, które wykorzystuje malware. Do tego użyjemy oczywiście naszego nowo opracowanego disassemblera rozmiaru.

Jako techniki hookingu użyjemy *Inline Hooking* z pewną wariacją w stosunku do techniki *Detour*

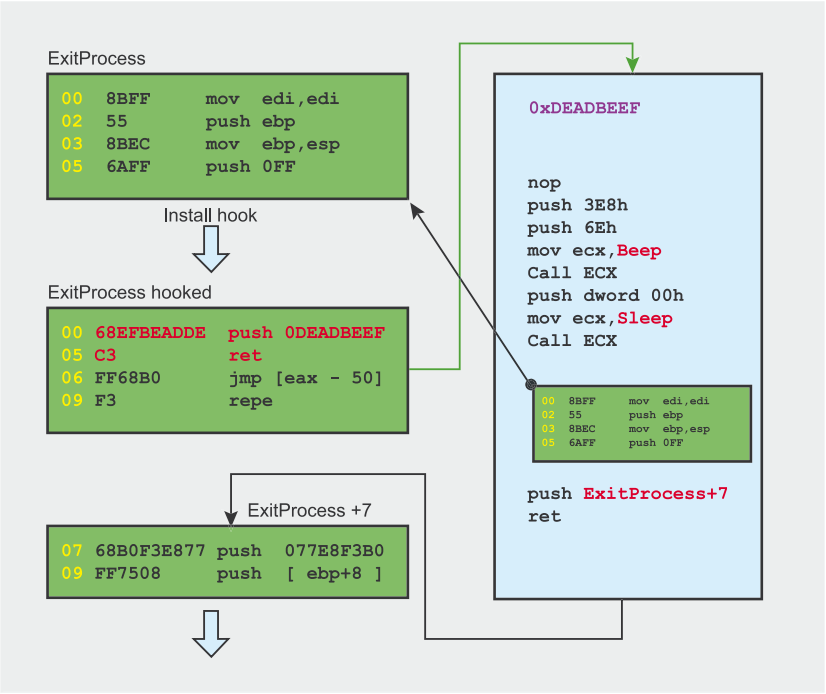
HookCode zawiera to, co byłoby Prologiem Funkcji *Detour*. Zadaniem tego prologu jest zawiadomienie nas poprzez alarm dźwiękowy, że malware dotarł do *ExitProcess*, wywołując API *Beep*. Następnie, jak wspomnieliśmy wcześniej, wywołamy *Sleep* za pomocą parametru przesłanego przez wiersz poleceń. Parametr ten będzie wystarczająco wysoki, aby umożliwić nam operacje zrzucania oraz wszystkie inne, które chcielibyśmy przeprowadzić. Po zakończeniu prologu, uruchomione zostaną pierwsze instrukcje *ExitProcess* (instrukcje nadzorowane przez disassembler rozmiaru) a następnie

Listing 4. *Budujemy nasz HookCode przy użyciu otrzymanych adresów*

```
printf("[+]Rebuilding
          HookCode...");

// K32!Beep
*( DWORD* )(
      HookCode + 9 ) =
      BeepAddr;

// Sleep param
Parameter = atoi( argv[2] );
*( DWORD* )(
      HookCode + 16 ) =
      Parameter;

// K32!Sleep
*( DWORD* )(
      HookCode + 21 ) =
      SleepAddr;

*( DWORD* )(
      HookCode + 48) =
      HookAddr + LenDasm;

printf("[OK]\n");
```

zostanie przywrócona kontrola następnej właściwej instrukcji (*ExitProcess+7*).

Później należy zrekonstruować *HookCode* i *ExitHook* przy pomocy adresów pamięci API oraz war-



Rysunek 5. *Schemat zahaczania ExitProcess*

tości uzyskanych przez disassembler rozmiaru.

Następnie tworzymy proces w trybie zawieszonym i rezerwujemy pamięć w jej przestrzeni adresów tak jak to pokazuje Listing 5.

Na koniec przywracamy *ExitHook* za pomocą adresu uzyskanego poprzez *VirtuaAllocEx*, łatamy *Entry Point* funkcji docelowej (w tym przypadku ExitProcess) przy pomocy *ExitHook*. Ilustruje to Listing 6.

Sposób wywołania programu byłby nastepujący:

```
congrio c:\acpinject.exe
10000 Kernel32.dll ExitProcess
```

- jako pierwszy argument mamy path malware
- jako drugi argument interwał w milisekundach przejścia do Sleep
- trzeci argument to DLL, która eksportuje funkcję docelową
- Funkcja docelowa w tym przypadku to ExitProcess

Mała refleksja na koniec

Jak mogliśmy zaobserwować w artykule, w szerokiej dziedzinie inżynierii wstecznej w zasadzie wszystkie jej pola w pewnym punkcie się zbiegają. Łączyliśmy różne techniki jak na przykład disassemblery romiaru, hooking oraz wprowadzanie procesów, aby wspomóc naszą analizę malware.

Paradoksalnie tych samych technik używa złośliwe oprogramowanie dla własnych korzyści, co także powoduje jednoczesny rozwój inżynierii odwrotnej. Im większy stopień skomplikowania osiągają rootkity, wirusy itp., tym staranniej badane są wykorzystane techniki oraz sposoby ich pokonania. W ten sposób tworzy się coś w rodzaju biegu przełajowego, w którym uczestniczą badacze, programiści malware czy autorzy wirusów, chociaż bez wątpienia skutki tego odczuwają miliony użytkowników, nie możemy jednak zaprzeczyć, że po obydwu stronach rozwijają się procesy badawcze oraz innowacja. ●

Listing 5. *Proces w trybie zawieszonym*

```
Ret1 = CreateProcessA( szExecutableName, NULL, NULL, NULL,0,
 CREATE_SUSPENDED,NULL,NULL,
 &strStartupProceso,&strProceso);

if( !Ret1 ) ShowError();
printf("[OK]\n");

printf("[+]Allocating remote memory...");
Ret2 = (DWORD) VirtualAllocEx( strProceso.hProcess,NULL,sizeof(HookCode),
 MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

Listing 6. *Przywracamy ExitHook*

```
 *( DWORD* )( ExitHook + 1 ) = Ret2;
 printf("[OK]->Address : 0x%x",Ret2);
 printf("\n[+]Hooking %s...",argv[4]);
 printf("\n\t[-]Reading %d bytes from %s Entry Point ...", LenDasm, argv[4]);
 /* Kopiujemy nadzorowane instrukcje do sekcji Trampoline */
 Ret1 =(DWORD) memcpy( (LPVOID)( HookCode + 27 ),(LPVOID)HookAddr, LenDasm);
 if( !Ret1 ) ShowError();
 printf("[OK]\n");
 printf( "\t[-]Hooking %s...", argv[4] );
 Ret1=0;
 while( !Ret1 )
 {
              ResumeThread(strProceso.hThread);
              Sleep(1);
              SuspendThread(strProceso.hThread);
              Ret1 = WriteProcessMemory(strProceso.hProcess,
                      (LPVOID)HookAddr,  / * Łatamy Funkcję docelową*/
                                                    ExitHook, HookLength,
                      NULL);                    /* w pamięci      */
 }
 printf("[OK]\n");
 printf("\t[-]Injecting Hook...");
 Ret1 = WriteProcessMemory(strProceso.hProcess,(LPVOID)Ret2,    /* Kopiujemy
                    kod do przestrzeni adresowej*/
      HookCode, sizeof(HookCode), NULL);/* nowo utworzonego procesu */
/* Pozwalamy procesowi toczyć się */
ResumeThread(strProceso.hThread);
```

O autorze

Autor od 16 lat interesuje się środowiskiem inżynierii wstecznej niskiego poziomu oraz ogólnie bezpieczeństwem informatycznym. Jako całkowity samouk rozpoczął w wieku 19 lat pracę programisty. Następnie rozwijał swoją karierę zawodową w obszarach związanych z niskim poziomem, programami antywirusowymi i podatnością na ataki. Obecnie koncentruje swoją działalność na tej ostatniej dziedzinie.

W Sieci

- *http://www.reversemode.com/index.php?option=com_remository&Itemid=2&func=select&id=8* – Kompletny kod źródłowy wszystkich aplikacji wymienionych w artykule. Disassembler rozmiaru. Przykład malware i aplikacji do hookingu.
- *http://research.microsoft.com/~galenh/dfPublications/HuntUsenixNt99.pdf* – Detours: Binary Interception of Win32 Functions.
- *http://msdn2.microsoft.com/en-us/library/ms253960(VS.80).aspx* – Structure Exception Handling in x86