Wydział Podstawowych Problemów Techniki
Politechniki Wrocławskiej

Wirtualny debugger-dekompiler złośliwego oprogramowania.

Michał Składnikiewicz

Praca inżynierska napisana
pod kierunkiem
dr Wiesława Cupały

Wrocław 2007

Dla Arashi Coldwind, tak po prostu.
Dla Unavowed'a, za tysiące wspólnych linii kodu.

Spis treści

1 Wstęp

Tematem pracy dyplomowej jest analiza zagadnień związanych z badaniem oprogramowania złośliwego przy wykorzystaniu technik analizy wstecznej, oraz próba zaprojektowania i zaimplementowania prototypu systemu wspomagającego analizę wsteczną kodu złośliwego przeznaczonego dla platformy x86 z systemem z rodziny Microsoft®Windows NT®. System wspomagający, zwany dalej SkySong, powinien wykorzystywać elementy emulacji procesora x86 oraz emulacji środowiska systemu Microsoft®Windows NT®.

Rozdział drugi pracy, „Wprowadzenie do architektury x86 oraz Windows NT®", zawiera omówienie architektury procesora x86 oraz systemu Microsoft® Windows®z rodziny NT, ze zwróceniem uwagi przede wszystkim na informacje ważne z punktu widzenia analizy wstecznej.

Rozdział trzeci, „Analiza wsteczna", zawiera wytłumaczenie terminu „analiza wsteczna" oraz omówienie technik analizy wstecznej, wraz z technicznymi detalami implementacji części technik na platformie x86 oraz Win32.

Rozdział czwarty, „Przeciwdziałanie analizie wstecznej", zawiera opis metod utrudniających analizę wsteczną, wraz z technicznymi detalami implementacji części metod.

Rozdział piąty, „SkySong", zawiera opis założeń, architektury, wewnętrznych mechanizmów systemu SkySong, oraz opis budowy przykładowego frontendu opartego o SkySong.

Poszczególne zagadnienia będą omawiane przede wszystkim w odniesieniu do trybu chronionego procesora Intel®x86 oraz systemu Microsoft®Windows®z rodziny NT (w skrócie Win32), czego głównym powodem jest fakt iż przeważająca część oprogramowania złośliwego tworzona jest z myślą o tej platformie.

SkySong, oraz przykładowy frontend, został zaimplementowany w języku C++ w standardzie C++ 98. Do kompilacji wykorzystany został kompilator g++, w wersji 3.4.5 (mingw special), wchodzący w skład pakietu MinGW. Dodatkowymi narzędziami użytymi podczas implementacji SkySong był edytor tekstowy GVIM w wersji 7.0, GNU Make 3.80 wchodzący w skład pakietu MinGW oraz system kontroli wersji Subversion w wersji 1.3.0 (r17949). Do testów został użyty również pakiet Netwide Assembler w wersji 0.98.38. W obecnej implementacji systemu SkySong korzysta jedynie z biblioteki CRT (ang. C RunTime). System SkySong w obecnej wersji zajmuje około 25 tys. linii kodu, w około 341 plikach źródłowych.

1.1 Przyjęta konwencja zapisu wartości liczbowych

W niniejszym dokumencie oznaczenie „0x" poprzedzające liczbę oraz sufiks „h" oznaczają iż dana liczba jest zapisana w systemie heksadecymalnym (szesnastkowym), natomiast prefiks „b" oznacza iż dana liczba jest zapisana w systemie binarnym (zero-jedynkowym). Wartości liczbowe bez prefiksów oraz sufiksów zapisane są w systemie decymalnym (dziesiętnym).
Kod w postaci binarnej zapisywany jest zawsze oktetami (bajtami) w postaci heksadecymalnej.

2 Wprowadzenie do x86 oraz Windows NT®

Celem dobrego zrozumienia informacji zawartych w niniejszym dokumencie, wymagana jest znajomość podstaw architektury procesorów z rodziny Intel®x86 pracujących w trybie chronionym, oraz znajomość podstaw architektury systemu Microsoft®Windows®z rodziny NT. Poniżej znajduje się krótkie wprowadzenie do wymienionych architektur, zawierające przede wszystkim informacje ważne z punktu widzenia analizy wstecznej. Niektóre mechanizmy są przedstawione w uproszczonej wersji, co jest zaznaczone w tekście.

2.1 Architektura procesora x86, tryb chroniony

Procesory z rodziny Intel®x86 są pseudo-CISC'owymi (ang. Complex Instruction Set Computer) procesorami pracującymi w trybach (w zależności od modelu) 16-bitowym, 32-bitowym lub 64-bitowym. Na potrzeby tej pracy omówione zostaną jedynie procesory 32-bitowe.

Procesory 32-bitowe, czyli np. Intel386™, Intel486™, Intel®Pentium™, oraz nowsze, mogą pracować w kilku możliwych trybach:[5]

1. Tryb rzeczywisty (ang. real mode, real address mode) – charakteryzujący się 16-bitowymi operacjami, bezpośrednim dostępem do 1MB pamięci, bezpośrednim dostępem do funkcji BIOS (ang. Basic Input/Output System) i sprzętu, oraz brakiem mechanizmów odpowiedzialnych za szeroko pojętą ochronę pamięci czy wielozadaniowość. W trybie rzeczywistym funkcjonował system MS-DOS®.

2. Tryb chroniony (ang. protected mode) – charakteryzujący się 32-bitowymi operacjami, bezpośrednim dostępem do 4GB pamięci, podziałem poziomów uprzywilejowania (od „ring 0", czyli tzw. trybu systemu charakteryzującego się pełnym dostępem do możliwości procesora oraz sprzętu, do „ring 3", czyli trybu użytkownika charakteryzującego się ograniczonym dostępem do pamięci, urządzeń, oraz funkcji procesora wspomagających działanie systemu), oraz mechanizmami ochrony dostępu do pamięci

(pamięć wirtualna, strony pamięci). W trybie chronionym funkcjonuje większość nowoczesnych kerneli przeznaczonych dla platformy x86, takich jak Linux, czy kernele systemów Sun™Solarisa™oraz Microsoft® Windows®.

3. Tryb wirtualny 8086 (ang. virtual-8086 mode, VM86) – tryb umożliwiający uruchamianie aplikacji 16-bitowych w trybie chronionym.

Jako ciekawostkę można podać jeszcze jeden tryb – tryb nierzeczywisty (ang. unreal mode), którego istnienie jest wynikiem błędu w procesorze. Tryb nierzeczywisty pozwala na wykonywanie kodu 16-bitowego mającego bezpośredni dostęp do 4GB pamięci. Uzyskiwany jest on poprzez wejście do trybu chronionego, ustawienie deskryptorów segmentów w tryb 32-bitowy, oraz powrót do trybu rzeczywistego. Obecnie unreal mode nie jest uznawany już za błąd, lecz za cechę procesora zachowaną w celach wstecznej kompatybilności.[17]

Z uwagi iż systemy operacyjne Microsoft®Windows®z rodziny NT działają przede wszystkim w trybie chronionym, pozostała część wprowadzenia będzie dotyczyła tego trybu.

Procesor, jako jednostka wykonująca program przygotowany przez programistę posiada szereg mechanizmów, takich jak rejestry czy stos, które umożliwiają, bądź ułatwiają oprogramowanie procesora. Procesor x86 posiada 8 rejestrów ogólnego przeznaczenia, które służą do przechowywania 32-bitowych wartości. Interpretacja zawartości zależy tylko i wyłącznie od kontekstu użycia, przykładowa wartość FFFFFFFFh może zostać zinterpretowana jako liczba całkowita -1, liczba naturalna 4294967295, adres FFFFFFFFh, lub numer indeksu w tablicy. Rejestry ogólnego przeznaczenia noszą następujące oznaczenia (niektóre z rejestrów posiadają specjalne funkcje w przypadku grup niektórych instrukcji, funkcje te podane zostały po myślniku):[2]

- EAX
- ECX – używany jako licznik w pętlach (LOOP, REP/REPNE)

- EDX
- EBX
- ESI – adres źródła dla operacji na łańcuchach danych (REP/REPNE)
- EDI – adres celu dla operacji na łańcuchach danych (REP/REPNE)
- ESP – zawiera adres ostatniego elementu stosu (instrukcje stosu)
- EBP – adres ramki stosu (ENTER/LEAVE)

Ponadto wyróżniane są dwa rejestry specjalne, które nie są bezpośrednio dostępne, jednak w których wartość można ingerować w sposób pośredni:

- EIP (ang. instruction pointer) – wskaźnik następnej instrukcji do wykonania
- EFLAGS – rejestr zawierający flagi stanu oraz flagi kontroli

Rejestr flagowy EFLAGS jest podzielony na pola o wielkość 1 (wyjątkiem jest pole IOPL, które zajmuje 2 bity). Pola te, określane mianem flag, zawierają informacje o trybie działania CPU (tj. stanie niektórych rozszerzeń procesora), rodzaju wyniku ostatniej operacji arytmetycznej lub wyznaczają działanie niektórych instrukcji. Najważniejsze flagi to:

- CF (ang. Carry Flag, bit 0, flaga stanu) – flaga przeniesienia, ustawiana gdy w wyniku operacji arytmetycznej nastąpi przeniesienie lub zapożyczenie.
- PF (ang. Parity Flag, bit 2, flaga stanu) – flaga parzystości jedynek, ustawiana gdy wynik ostatniej operacji arytmetycznej zawiera parzystą ilość jedynek.
- AF (ang. Auxiliary Carry Flag, bit 4, flaga stanu) – flaga przeniesienia, analogiczna do CF, używana przy instrukcjach BCD (ang. binary coded decimal).

- ZF (ang. Zero Flag, bit 6, flaga stanu) – flaga jest ustawiana gdy wynikiem ostatniej operacji arytmetycznej jest zero.

- SF (ang. Sign Flag, bit 7, flaga stanu) – flaga znaku, ustawiana gdy wynikiem ostatniej operacji arytmetycznej w interpretacji całkowitej jest liczba ujemna.

- TF (ang. Trap Flag, bit 8, flaga systemowa) – flaga trybu krokowego, jeśli ustawiona, procesor po wykonaniu jednej. instrukcji przerywa działanie i zgłasza przerwanie (INT 1, wyjątek trybu krokowego).

- IF (ang. Interrupt Enable Flag, bit 9, flaga systemowa) – flaga włączonej obsługi przerwań; gdy wyłączona, zgłoszone przerwania nie są obsługiwane od razu.

- DF (ang. Direction Flag, bit 10, flaga kontroli) – flaga kierunku; jeśli ustawiona, instrukcje operujące na łańcuchach danych działają w odwrotnym kierunku (tj. zakładają że kolejne dane znajdują się na adresach młodszych).

Po za podstawowymi rejestrami dostępne są rejestry segmentowe, które są selektorami przestrzeni adresowej danego segmentu. Rejestry segmentowe zawierają 16-bitową zmienną będącą indeksem w tablicy GDT lub LDT. Tablica GDT (ang. global descriptor table), dostępną tylko dla procesora oraz systemu, zawiera m.in. dopuszczalną przestrzeń adresową w pamięci wirtualnej dla danego segmentu, oraz prawa jakie dana przestrzeń posiada (zapis/odczyt). Każdy wpis dotyczący segmentu zawiera m.in. początek segmentu w pamięci wirtualnej i jego wielkość. Tak więc jeśli wartością indeksu rejestru segmentowego jest na przykład 6, to przy użyciu danego segmentu, z GDT odczytywany jest deskryptor z pozycji szóstej. Wyróżnia się 6 rejestrów segmentowych:

- CS (ang. code segment) – segment kodu, z którego odczytywane są instrukcje do wykonania.

- SS (ang. stack segment) – segment stosu.
- DS (ang. data segment), ES, FS, GS – segmenty danych.

Ponadto procesor posiada 8 rejestrów debuggera (z czego 2 są zarezerwowane) oznaczonych literami DR oraz kolejnymi cyframi (DR0, DR1, itd). Rejestry debuggera, dostępne jedynie z trybu ring 0, służą do sterowania wbudowanymi w procesor sprzętowymi breakpointami. Wyróżnia się następujące rejestry:

- DR0, DR1, DR2, DR3 – rejestry zawierające adres (wirtualny).
- DR6 – rejestr stanu.
- DR7 – rejestr kontrolujący.

Dodatkowo procesor zawiera pewną liczbę, zależną od modelu, rejestrów MSR[18] (ang. model-specific register) służących do kontrolowania rozmaitej funkcjonalności procesora. Ponadto istnieje 5 rejestrów kontrolujących tryb pracy procesora. Są one oznaczone CR0 do CR4.

Nowe procesory posiadają również rejestry związane z rozszerzeniami procesora:

- FPU – rejestry stosu FPU st0-st7 (80 bitów per rejestr)
- MMX$^{\text{TM}}$– rejestry mm0 – mm8 (64 bitów per rejestr)
- SSE, SSE2, SSE3, SSSE3, SSE4[19] – rejestry xm0 – xm8 (128 bitów per rejestr)

Oprócz wbudowanych rejestrów, procesor operuje na pamięci RAM (ang. random access memory). W trybie chronionym z punktu widzenia trybu użytkownika (ring 3) nie ma bezpośredniego dostępu do fizycznej przestrzeni adresowej (tj. takiej w której adres odpowiada bezpośrednio numerowi komórki

pamięci na kości ram), zamiast tego aplikacje operują w przestrzeni wirtualnej, modyfikowanej dodatkowo przez mechanizm segmentów. Tłumaczenie przez procesor adresu wirtualnego VA w segmencie S na adres fizyczny odbywa się w następujący sposób:[2]

1. Pobierany jest deskryptor segmentu S z GDT/LDT (jest to oczywiście objęte mechanizmem cacheowania celem optymalizacji działania) i sprawdzana jest poprawność wpisu.

2. Do adresu VA dodawany jest adres początku segmentu S pobrany z deskryptora segmentu. Uzyskany adres określa się mianem adresu liniowego (LA, ang. Linear address)

3. Z LA pobierany jest numer strony. Najczęściej wykorzystywane są strony o wielkości 4096 bajtów (1000h), których numer uzyskuje się z 20 najstarszych bitów LA (LA jest 32-bitowe).

4. Deskryptor strony jest wyszukiwany w katalogu stron (ang. Page Directory) oraz w tabeli stron (ang. Page Table). W przypadku nieznalezienia strony zgłaszany jest błąd do systemu operacyjnego (wymuszający na nim np. pobranie strony zapisanej na dysk z powrotem do pamięci).

5. Offset na stronie (najmłodsze 12 bitów LA) jest dodawany do adresu fizycznego strony pobranego z deskryptora strony. Uzyskany adres jest adresem fizycznym (PA, ang. Physical Address).

Dzięki zastosowaniu takiego mechanizmu, możliwe jest utworzenie oddzielnej wirtualnej przestrzeni adresowej dla każdego procesu (uruchomionego programu) w systemie, co umożliwia działanie procesom operującym (wirtualnie) na tym samym (pod względem wartości adresów) fragmencie pamięci, oraz fizycznie oddziela procesy od siebie.

Dowolny fragment pamięci możliwej do zaadresowania z segmentu SS (segment stosu) może zostać użyty jako stos. Stos jest normalnym fragmentem pamięci, jednak jest rozpatrywany ze względu na swoje zastosowanie, czyli opartą na tablicy elementów 32 bitowych (4 bajtowych) kolejkę LIFO (ang. Last In First Out). Przyjęło się że „dno" ramki stosu (lub po prostu „dno stosu") jest wskazywane przez rejestr EBP, jednak nie jest to konieczne (obecne kompilatory optymalizujące odchodzą od tego). Najnowszy element stosu wskazywany jest przez rejestr ESP. Do operowania na stosie służą przede wszystkim dwie instrukcje:[4]

- PUSH – zmniejsza adres zawarty w ESP o 4 bajty i w uzyskane miejsce wpisuje nowy element stosu.
- POP – zwraca wartość wskazaną przez ESP, po czym zwiększa ESP o 4.

Stos „rośnie" w stronę młodszych adresów, czyli pierwszy element dodany na stos będzie pod adresem starszym (większym) niż kolejny element.

Oprócz powyższych instrukcji, na stosie operują również instrukcje:

- CALL – skok pod podany adres, adres powrotu jest umieszczany na stosie
- RET – powrót pod adres uprzednio wrzucony na stos (para do instrukcji CALL)
- RETN – powrót pod adres uprzednio wrzucony na stos, oraz usunięcie ze stosu n elementów (para do instrukcji CALL)
- PUSHF – umieszcza rejestr flagowy EFLAGS na stosie
- POPF – pobiera wartość ze stosu do rejestru flagowego EFLAGS (para do instrukcji PUSHF)
- PUSHA – umieszcza wszystkie rejestry ogólnego przeznaczenia na stosie

- POPA – pobiera kolejne 8 wartości ze stosu i umieszcza je w rejestrach ogólnego przeznaczenia (para do instrukcji PUSHA)
- ENTER – stworzenie ramki stosu
- LEAVE – usunięcie ramki stosu

Stos jest używany zazwyczaj przez do zapamiętywania stanu rejestrów, przekazywania argumentów podprogramom (funkcjom), zapamiętywania adresu powrotu, oraz pobierania adresu EIP oraz bezpośredniej modyfikacji rejestru flagowego.

Ostatnim ważnym, z punktu widzenia analizy wstecznej mechanizmem, jest mechanizm przerwań oraz wyjątków. W przypadku wystąpienia błędu (np. próby odczytania pamięci przeznaczonej tylko do zapisu, lub np. dzielenia przez zero), wygenerowania przez programistę za pomocą instrukcji przerwania (np. korzystając z instrukcji INT) lub błędu (np. instrukcja UD2 służy do wywołania wyjątku #UD[6] – nieprawidłowa instrukcja (ang. invalid opcode)), lub wygenerowania przez sprzęt jakiegoś zdarzenia (np. nadejście nowych danych), przerywane jest działanie obecnego procesu, i wywoływana jest odpowiednia procedura obsługująca wyjątek/przerwanie. Adres procedury pobierany jest z tabeli IDT (ang. interrupt descriptor table), numer procedury zależy od typu wyjątku lub numeru przerwania. Procedury te dostarczane są przez system operacyjny. Przykładowymi błędami generującymi wyjątki są:

- Dzielenie przez zero.
- Próba dostępu do pamięci do której nie ma się odpowiednich praw.
- Wywołanie instrukcji do której nie ma się praw (np. wymagającej trybu ring 0 z poziomu ring 3).

Przykładowymi instrukcjami generującymi przerwanie są:

- INT n – generuje przerwanie numer n.

- INT3 – przerwanie debuggera.
- INT0 – przerwanie 0.

Przykładowymi instrukcjami generującymi wyjątek są:

- UD2 – generuje wyjątek „nieprawidłowa instrukcja".
- dowolna nieprawidłowa instrukcja – jw.

Należy wspomnieć jeszcze o samym mechanizmie przekazywania kodu do wykonania procesorowi. W uproszczonej wersji, procesor odczytuje następną instrukcję do wykonania z segmentu CS, adresu EIP. Instrukcja może mieć wielkość od 1 do 15 bajtów, powyżej tej wielkości zgłaszany jest wyjątek #UD. Instrukcja jest następnie rozkodowywana przez procesor i wykonywana.

Instrukcja zbudowana jest z następujących elementów:[3] [1]

1. Instruction Prefix – opcjonalne prefiksy instrukcji. Istnieją cztery grupy prefiksów:
   (a) LOCK (F0) / REPNE (F2) / REPE (F3)
   (b) Przeciążenie segmentu (CS (2E), DS (3E), ES (26), FS (64), GS (65), SS (36)) lub podpowiedź rozgałęzienia (skok będzie wykonany (2E), skok nie będzie wykonany (3E))
   (c) Prefiks przeciążenia wielkości operandu (66)
   (d) Prefiks przeciążenia wielkości adresu (67)
2. Opcode – 1, 2 lub 3 bajtowy identyfikator rozkazu
3. ModR/M – opcjonalny bajt opisujący dodatkowy operand oraz odwołanie do pamięci
4. SIB – opcjonalny bajt zawierający dodatkowe informacje odnośnie odwołania do pamięci

5. Displacement – opcjonalna 1, 2 lub 4 bajtowa wartość określająca przemieszczenie w pamięci

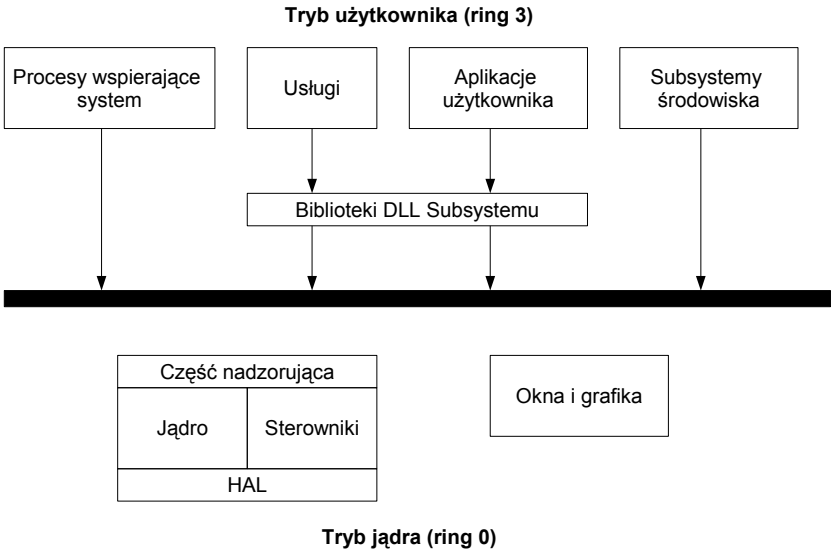
6. Immediate – opcjonalna wartość zawierająca stałą wartość liczbową

Większość parametrów jest opcjonalna, a ich wystąpienie zależy od potrzeb programisty (prefiksy), bądź budowy rozkazu (ModR/M, SIB, Displacement, Immediate). Ilość różnych identyfikatorów rozkazów jest duża, przykładowo w procesorze Intel®Pentium®MMX™(który zawierał wbudowany koprocesor FPU, oraz rozszerzenie MMX™) było ponad 700 różnych rozkazów. Interpretacja poszczególnych argumentów (np. czy 0 w polu ModR/M Reg oznacza EAX czy MM0) zależy od danego opkodu.

Po wykonani instrukcji, EIP zwiększane jest o całkowitą wielkość instrukcji. Wyjątkiem są skoki, które zmieniają wartość EIP na podaną.

2.2 Architektura systemu Microsoft®Windows®z rodziny NT

System Microsoft®Windows®jest flagowym produktem firmy Microsoft®. Nazwa Windows®pojawiła się po raz pierwszy w listopadzie 1985. Wówczas była to graficzna nakładka na system operacyjny MS-DOS®, powstała w odpowiedzi na rosnącą popularność graficznych interfejsów użytkownika prezentowanych przez inne firmy. Wraz z biegiem lat Microsoft®Windows®stał się najpierw 16-bitowym środowiskiem operacyjnym (Windows 1.0®, Windows®2.0), następnie hybrydowym 16/32-bitowym środowiskiem operacyjnym (Windows 3.0®, Windows®3.1), potem hybrydowym 16/32- bitowym systemem operacyjnym (Windows®3.11, Windows®95, Windows®98, Windows®ME), a w końcu w pełni 32-bitowym systemem operacyjnym (Windows NT®3.1, Windows NT®3.5, Windows NT®3.51, Windows NT®4.0, Windows®2000, Windows XP®, Windows Vista™). Rodzina NT zagościła również na procesorach 64-bitowych.[20]



Rysunek 1: Uproszczona architektura systemu Windows NT®[9]

System Microsoft®Windows NT®pracuje w dwóch ringach:

- ring 0 - w którym mieści się kernel (jądro systemu), sterowniki, część nadzorująca (dispatcher, memory manager, etc), HAL, oraz procedury odpowiedzialne za okna i grafikę.
- ring 3 – w którym oprócz procesów użytkownika działają niektóre procesy wspierające pracę systemu, usługi (np. spooler), oraz subsystemy.

Pozostałe tryby, czyli ring 1 oraz ring 2 są nieużywane. Część działająca w trybie ring 0 składa się z następujących komponentów:[9]

- HAL (ang. Hardware Abstraction Layer) – Abstrakcyjna warstwa nałożona na sprzęt, jej implementacja zależna jest od platformy na której system operuje. Dzięki HAL reszta jądra jest odizolowana od sprzętu i niewrażliwa na niewielkie różnice w sprzęcie (np. różnice w płytach głównych). Implementacja HAL znajduje się w pliku \Windows\System32\HAL.dll.

- Jądro systemu (ang. kernel) – Zawiera niskopoziomowe funkcje systemu operacyjnego, takie jak kolejkowanie wątków (ang. scheduling), dysponowanie przerwaniami i wyjątkami czy synchronizowanie wielu procesorów. Kernel zawiera również implementacje końcową niektórych funkcji API (ang. Aplication Programming Interface, Interfejs Programistyczny) wykorzystywanych przez procesy działające w trybie ring 3. Jądro znajduje się w pliku \Windows\System32\ntoskrnl.exe (lub ntkrnlpa.exe w wersji obsługującej rozszerzenie procesora PAE).

- Sterowniki (ang. drivers) – Sterowniki odpowiadają za obsługę sprzętu, umożliwienie programom trzecim wykorzystanie sprzętu, oraz implementują niektóre funkcje systemowe, takie jak system plików czy warstwę sieciową. Każdy sterownik jest modułem opartym o system callbacków. Sterowniki mają zazwyczaj rozszerzenie „.sys".

- Część nadzorująca (ang. executive) – Komponenty systemowe takie jak menadżer pamięci, menadżer procesów i wątków, komunikacja między procesowa, itp. Część nadzorująca znajduje się tym samym pliku co kernel.

- Okna i grafika – Część systemu odpowiedzialna za system okien oraz rendering graficzny interfejsu użytkownika. Całość znajduje się w pliku \Windows\System\win32k.sys.

Drugą część systemu stanowią komponenty pracujące w trybie użytkownika. Są to:[9]

- Procesy wspierające system – Powiązane bezpośrednio z systemem procesy takie jak proces logon czy menadżer sesji. Do tej kategorii zalicza się każdy proces systemowy który nie jest usługą.

- Usługi – Odpowiednik Unixowych daemonów. Usługi systemowe dostarczają systemowi pewnej określonej funkcjonalności (np. spooler lub serwer SQL).

- Aplikacje użytkownika – Jeden z kilku typów aplikacji użytkownika, działających pod jednym z subsystemów.

- Subsystemy – Procesy serwerowe subsystemów dostarczają środowisko, API oraz wymagane mechanizmy aby program napisany pod dany subsystem funkcjonował poprawnie. Windows NT®dostarczany był z trzema subsystemami: Windows®, POSIX oraz OS/2®. Z OS/2® zrezygnowano w Windows®2000. Windows®XP był dostarczany jedynie z subsystemem Windows®, ale subsystem POSIX można było pobrać za darmo ze strony Microsoftu®. Oddzielnym subsystemem jest równie NTVDM (ang. NT Virtual DOS Machine).

API systemu Windows NT®zawarte jest przede wszystkim na czterech modułach DLL:[9]

- Kernel32.dll – Zawierający m.in. funkcje do zarządzania procesami, wątkami, plikami, potokami, oraz systemem komputerowym. Moduł ten, z uwagi na nazwę, często jest błędnie brany z kernel systemu Windows.

- User32.dll – Frontend systemu okien. Zapewnia funkcje do tworzenia oraz operowania na oknach.

- Gdi32.dll – Biblioteka zapewniająca procedury graficzne, potrzebne do rysowania interfejsu użytkownika.

- Advapi32.dll – Biblioteka zawierająca funkcje do operowania na rejestrze systemowym, oraz dostarcza rozwiązania kryptograficzne.

Należy dodać iż moduły te zależą od biblioteki Ntdll.dll, która zawiera w sobie m.in. wrappery na wywołania funkcji jądra systemu (syscalle). Większość funkcji oferowanych przez Ntdll.dll nie jest oficjalnie udokumentowana. Przykładowo, jeśli programista chce utworzyć nowy plik, wywołuje funkcję CreateFile z modułu Kernel32.dll. W tej funkcji następuje wstępne przeanalizowanie argumentów, po czym zostaje wywołana funkcje NtCreateFile zawarta w Ntdll.dll. Z kolei funkcja NtCreateFile ustawi numer syscall (rejestr

EAX) na 25h (NtCreateFile w przypadku Windows®XP), po czym wykona instrukcję procesora SYSENTER. W tym momencie następuje przejście z ring 3 do ring 0, i wykonanie przejmuje procedura obsługująca SYSENTER, znajdująca się w ntoskrnl.exe. Procedura ta wywołuje wewnętrzną funkcje kernela NtCreateFile, która następnie odwoła się do odpowiedniego sterownika odpowiedzialnego za system plików w miejscu gdzie plik ma zostać utworzony. Sterownik być może odwoła się do kolejnego sterownika odpowiedzialnego za fizyczny zapis danych na nośniku, przez co plik zostanie fizycznie stworzony.

Z punktu widzenia analizy wstecznej bardzo ważna jest budowa pliku wykonywalnego. W systemie Microsoft®Windows NT®obowiązują pliki wykonywalne PE (ang. Portable Executable). Format tych plików jest złożony, przedstawię więc jego uproszczoną wersję. Plik PE składa się z kilku części:[15]

1. Nagłówek DOS (struktura IMAGE_DOS_HEADER) - Nagłówek używany w plikach wykonywalnych MZ używanych w systemie MS-DOS®. Obecnie używany jest jedynie dla zachowania kompatybilności wstecz, oraz umożliwienie wykonania krótkiego podprogramu DOS w razie próby wykonania pliku PE na systemie MS-DOS®.

2. Podprogram DOS (DOS stub, opcjonalne) – Krótki program przeznaczony dla systemu MS-DOS®, wypisujący komunikat w stylu „Ten program wymaga systemu Microsoft®Windows®". Możliwe jest oczywiście modyfikowanie tego podprogramu.

3. Nagłówek PE (struktura IMAGE_NT_HEADERS) – Nagłówek PE, podzielony na dwie sekcje: FileHeader (struktura IMAGE_FILE_HEADER), oraz OptionalHeader (struktura IMAGE_OPTIONAL_HEADER). Nagłówki zawierają m.in. informacje o ilości sekcji w pliku, lokalizacji importów, eksportów, zasobów, oraz informacje wymagane do uruchomienia programu, takie jak adres docelowy załadowania obrazu czy adres startowy.

4. Nagłówki sekcji (structura IMAGE_SECTION_HEADER) – Informacje o każdej sekcji pliku, takie jak nazwa (opcjonalna), wielkość sekcji w pliku, lokalizacja sekcji w pliku, wielkość sekcji w pamięci, lokalizacja sekcji w pamięci czy prawa dostępu do pamięci sekcji.

5. Sekcje – Binarne dane sekcji. Może to być kod, zainicjowane dane, niezainicjowane dane (w takim wypadku zazwyczaj wielkość sekcji w pliku to 0), zasoby, lub inne wybrane przez programistę informacje.

6. IAT (ang. Import Address Table, opcjonalne) – Lista modułów DLL oraz nazw lub numerów (ang. ordinal) funkcji (z poszczególnych modułów) wymaganych przez program do działania. Po uruchomieniu programu, IAT (w pamięci) znajdują się również adresy poszczególnych funkcji w pamięci. IAT może znajdować się w dowolnym miejscu.

7. EAT (ang. Export Address Table, opcjonalne) – Lista funkcji eksportowanych (udostępnianych innym modułom/programom) przez dany program. Lista zawiera nazwę funkcji, jej numer identyfikacyjny (ordinal) oraz jej adres. EAT może znajdować się w dowolnym miejscu.

8. Zasoby (ang. Resources, opcjonalne) – Hierarchicznie zorganizowany system zasobów. Zawierać może m.in. ikony, dane binarne, wygląd okien, dane tekstowe, itp. Zazwyczaj znajduje się w sekcji o nazwie „.rsrc" (nazwa sekcji wymagana jest przez jeden moduł DLL odpowiedzialny za mechanizm OLE).

9. Inne.

Proces uruchamiania programu, w wersji uproszczonej, wygląda następująco:

1. System tworzy nowy proces, a w nim nowy wątek który realizuje dalszą część procesu uruchomienia (część ta znajduje się w Ntdll.dll).

2. Nagłówek PE jest odczytywany i analizowany.

3. Następuje alokacja miejsca w pamięci pod adresem zwartym w OptionalHeader.ImageBase, o wielkości zawartej w OptionalHeader.SizeOfImage.

4. Do zaalokowanej pamięci kopiowane są dane z poszczególnych sekcji (pod odpowiednie adresy).

5. Ładowane są moduły wg. listy importów. Po załadowaniu modułu, adresy wymaganych funkcji zapisywane są w IAT.

6. Ustawiane są prawa do pamięci wg. danych z nagłówków sekcji.

7. Następuje przejście pod adres zawarty w OptionalHeader.AddressOfEntryPoint i rozpoczyna się wykonanie właściwego procesu.

Zakończyć proces można za pomocą funkcji ExitProcess, TerminateProcess, ExitThread (zamknięcie ostatniego wątku procesu) lub poprzez wyjście (RET) z głównej procedury programu.

3 Analiza wsteczna

3.1 Czym jest analiza wsteczna?

> **Analiza wsteczna**, **technika odwracania**, **inżynieria odwrotna**, **inżynieria wsteczna** *(ang.* **Reverse engineering**, **reversing***) to proces badania produktu (urządzenia, programu komputerowego) w celu ustalenia jak dokładnie działa, a także w jaki sposób i jakim kosztem został wykonany.*[21]

Termin „analiza wsteczna" użyty w stosunku do oprogramowania, oznacza proces jego dekompozycji w celu ustalenia jego budowy, działania, narzędzi użytych do jego stworzenia, zastosowanych technik implementacyjnych, oraz ustalenia formatów danych (plików, protokołów sieciowych) używanych przez dany program. Analiza kodu oprogramowania często określane jest mianem RCE (ang. Reverse Code Engineering).

W niniejszej pracy stosowane są zamiennie terminy „analiza wsteczna", „inżynieria odwrotna", jak i ich angielski odpowiednik „reverse engineering", również w jego skróconej wersji „RE" lub „RCE".

Z technicznego punktu widzenia analiza wsteczna polega na badaniu działania oraz budowy programu na podstawie jego źródła, jeśli jest ono w posiadaniu, źródła powstałego w wyniku deasemblacji lub dekompilacji (tzw. deadlistingu), lub analizy behawioralnej (przy wykorzystaniu debuggerów, oraz monitorów funkcji/wywołań systemowych). Wg niektórych autorów analiza własnego kodu źródłowego wykonywana dłuższy czas po jego napisaniu również jest reverse engineeringiem.[7]

Analiza wsteczna wykorzystywana jest:

- Przy tworzeniu lokalizacji oprogramowania. Często zdarza się iż twórca oprogramowania chce by jego produkt został całkowicie przetłumaczony na daną wersje językową przez lokalnego dystrybutora jednocześnie

nie chcąc udostępniać ani kodu źródłowego, ani specyfikacji formatów danych (przykładowo: plików graficznych zawierających tekst który powinien zostać przetłumaczony). W takim wypadku analiza wsteczna jest wykorzystywana do ustalenia użytego formatu danych, oraz ustalenia sposobu modyfikacji wersji językowej oprogramowania (w zależności od budowy danej aplikacji, może to być podmiana tekstu w pliku binarnym, podmiana zasobów (ang. resources) pliku wykonywalnego, itp.).

- Przy analizie kodu (oprogramowania) złośliwego. Ustalenie sposobu działania kodu złośliwego (ang. malware) jakim są np. wirusy, robaki czy konie trojańskie, pomaga przy jego deaktywacji, usunięciu oraz w prewencji.

- Przy dodawaniu funkcjonalności do gotowego oprogramowania. Jeśli autor oprogramowania nie udostępnił jego kodu, a jest potrzeba jego modyfikacji, analiza wsteczna jest bardzo pomocna przy ustaleniu dokładnego miejsca oraz sposobu modyfikacji. Przykładem może być dodanie obsługi ANSI Escape Codes do konsoli cmd.exe używanej w Microsoft®Windows®z rodziny NT.

- Przy usuwaniu funkcjonalności z gotowego oprogramowania. Przykładem mogą być tzw. cracki, które usuwają zabezpieczenia anty-pirackie z oprogramowania, a które są tworzone w oparciu o wiedze uzyskaną z reverse engineeringu danego produktu. Analiza wsteczna jest również wykorzystywana przy usuwaniu elementów oprogramowania szpiegującego (ang. spyware) z gotowego produktu.

- Przy odblokowywaniu ukrytej przez twórce oprogramowania funkcjonalności. Zdarza się że twórca oprogramowania dodał jakąś funkcjonalność do oprogramowania, lecz z uwagi na jej niestabilność/niedokończenie została ona zablokowana. Analiza wsteczna może zostać użyta do wykrycia i odblokowania takiej funkcjonalności.

- Do audytu bezpieczeństwa oprogramowania. Za równo autorzy opro-

gramowania, jak i specjaliści od bezpieczeństwa, szukając błędów w oprogramowaniu, które pozwoliłyby ,przykładowo, przejąć kontrole nad komputerem korzystającym z danego oprogramowania (buffer overflow, race condition, integer overflow, itd.), lub spowodować błąd w programie / odmowę poprawnego wykonania (Denial of Service), korzystają między innymi z analizy wstecznej.

- Przy tworzeniu oprogramowania kompatybilnego z innym oprogramowaniem którego dokumentacja techniczna nie została udostępniona. Przykładowo w celu stworzenia alternatywnego klienta danej aplikacji internetowej (gry, komunikatora), lub alternatywnego serwera aplikacji (emulatory serwerów gier).

- Przy zmianie platformy przeznaczenia dla danej aplikacji, nie mając dostępu do jej kodu źródłowego. Przykładem może być gra DOS'owa „Frontier: First Encounters" która w projekcie JJFFE (http://jaj22.org.uk/jjffe/) została zreversowana oraz przeniesiona na platformy wspierane przez SDL (ang. Simple Directmedia Layer, http://libsdl.org/), takie jak np. Microsoft® Windows®czy systemy oparte o kernel Linux.

Warto wspomnieć w tym miejscu o metodzie chińskiego muru (ang. Chinese wall, Clean room design), która zakłada istnienie dwóch oddzielnych grup specjalistów. Pierwsza grupa dostaje gotowy produkt innego producenta, dokładnie go analizuje (wykorzystując RE) tworząc bardzo szczegółową dokumentacje. Dokumentacja jest przekazywana drugiej grupie, która na jej podstawie tworzy nowy produkt, który pod względem działania i budowy jest niejako powieleniem poprzedniego, jednak pod względem prawno-formalnym nie ma z poprzednim produktem nic wspólnego (np. nie korzysta z jego kodu). Metoda ta jest stosowana aby obejść prawa autorskie związane z oryginalnym produktem. Trzeba jednak zaznaczyć iż metoda ta nie umożliwia obejścia patentów, na szczęście na terenie Unii Europejskiej patenty na oprogramowanie nie są przydzielane.[22]

3.2 Deasemblacja

Deasemblacja jest procesem bezpośredniego tłumaczenia instrukcji języka maszynowego (tzw. opkodów procesora) na ich odpowiedniki w języki assembler (tzw. mnemoniki). Termin deasemblacja można użyć zarówno do języków uruchamianych na rzeczywistym procesorze, jak i w stosunku do języków opartych o wirtualne maszyny (Java™, Python, Perl, E-Script) których postać binarna to tak zwany kod bajtowy (ang. bytecode), a którego deasemblacja przedstawiana jest w postawi pseudo-assemblerowych mnemoników. Należy pamiętać że deasemblacja jest prostym procesem, który jedynie przypisuje rozkazom maszynowym tekstową postać bardziej zrozumiałą dla człowieka.[10]

Przykładowo poniższy kod maszynowy dla procesora z rodziny x86 (tryb chroniony 32bit) zapisany heksadecymalnie

```
55 89 E5 8B 45 0C 03 45  08 5D C3
```

zostanie zdeasemblowany do postaci:

```
push    ebp
mov     ebp, esp
mov     eax, [ebp+0Ch]
add     eax, [ebp+08h]
pop     ebp
retn
```

Pewnym problemem w deasemblacji jest odseparowanie danych od kodu, oraz, w niektórych przypadkach, stwierdzenie miejsca rozpoczęcia instrukcji. Stosowane są dwa podejścia:

- Deasemblowane jest wszystko od początku do końca programu, ewentualne nieprawidłowe instrukcje oznaczane są jako dane (zazwyczaj „db liczba")
- Deasembler rozpoczyna deasemblacje w miejscu wejścia, i śledzi kod pod względem rozgałęzień oraz skoków, tzn jeśli w kodzie jest skok

bezwzględny (taki jak np. JMP, RET lub RETN) deasembler nie tłumaczy instrukcji które występują bezpośrednio po tym kodzie, a śledzi skok i kontynuuje deasemblacje w miejscu docelowym. W przypadku skoków warunkowych deasembler zapamiętuje adres docelowy skoku celem późniejszego wznowienia deasemblacji w miejscu docelowym.

Interaktywne deasemblery mają możliwość zmiany przez użytkownika typu danych zawartych w danym miejscu (dane/kod).

Najprostszym w pełni funkcjonalnym deasemblerem na platformę x86 jest ndisasm z pakietu Netwide Assembler (http://nasm.sourceforge.net/). Pozwala on przetłumaczyć kod maszynowy zapisany w postaci binarnej do postaci mnemonicznej. Obsługuje za równo tryb rzeczywisty procesora x86 (16-bit), jak i tryb chroniony (32bit). Wadą ndisasm jest brak wsparcia dla standardowych plików wykonywalnych, takich jak Win32 PE czy ELF (ang. Executable and Linkable Format, wcześniej Extensible Linking Format, format plików wykonywalnych stosowany w systemach POSIX'owych, takich jak systemy oparte o kernel Linux, system Solaris™, oraz inne). Z uwagi na to, wykorzystanie ndisasm ogranicza się do plików .com (ang. command, format plików wykonywalnych używanych w systemie MS-DOS®, charakteryzuje się brakiem jakichkolwiek nagłówków, oraz ograniczeniem wielkości do jednego segmentu, czyli do 65536 bajtów), oraz plików zawierających kod bez nagłówków. Kod ndisasm jest często używany do pisania dedykowanych deasemblerów obsługujących niestandardowe formaty, oraz do tworzenia translatorów kodu z jednej platformy na inną.

Przykład użycia ndisasm:

```
>ndisasm -b 32 func.bin
00000000 55      push ebp
00000001 89E5    mov ebp,esp
00000003 8B450C  mov eax,[ebp+0xc]
00000006 034508  add eax,[ebp+0x8]
```

```
00000009 5D      pop ebp
0000000A C3      ret

>
```

Deasembler ndisasm, z uwagi na jego prostotę, jest dobrym narzędziem do weryfikacji wyników otrzymywanych z własnego deasemblera.

Najpopularniejszym obecnie deasemblerem jest IDA belgijskiej firmy DataRescue (http://www.datarescue.com/), dostępny na platformy Microsoft®Windows® oraz systemy oparte o kernel Linux (tylko z interfejsem tekstowym). IDA, w przeciwieństwie do ndisasm, jest bardzo rozbudowaną aplikacją obsługującą wiele rodzin procesorów (np. Intel®x86, ARM™Risc, Motorola®68xxx, Intel®i860, Hitachi SH3/SH4, i inne) oraz wiele formatów plików wykonywalnych, włączając w to Win32 PE, ELF oraz pliki .com, jak i formatów plików obiektowych, np. COFF. Jako ciekawostkę należy dodać iż IDA obsługuje również kod bajtowy Java™, co umożliwia dekompilacje plików wykonywalnych Java™.class do postaci kodu pseudo-assemblera. Dodatkowo deasembler IDA zawiera wiele pomocnych funkcji umożliwiających np. reprezentowanie kodu na grafach (od wersji 5.0), możliwość podpięcia pluginów, możliwości pisania skryptów (w języku IDC oraz, za pomocą pluginów, w Perlu lub Pythonie), dodawania komentarzy bezpośrednio w kodzie, wyszukiwanie standardowych funkcji bibliotecznych za pomocą systemu filtrującego (FLIRT) działającego analogicznie do skanerów antywirusowych, jak i wiele innych. IDA zawiera również wbudowany debugger, który umożliwia m.in. debugowanie zdalne programów napisanych pod Linuxa z poziomu Microsoft® Windows®. Debugger jest ubogi, ale jego funkcjonalność jest wystarczająca do rozwiązywania prostych problemów.

3.3 Dekompilacja

Dekompilacja jest procesem tłumaczenia języka niskopoziomowego na język wysokopoziomowy, na przykład kodu maszynowego na język C lub

C++. Dekompilacja, w przeciwieństwie do deasemblacji, wymaga analizy kodu również z logicznego punktu widzenia, utworzenia grafu wykonania, a następnie przełożenia go na odpowiednie instrukcje warunkowe, skoki oraz pętle[13]. Dekompilacja jest skomplikowanym procesem, który w przypadku bardzo złożonego języka wejściowego (np. język maszynowy procesorów z rodziny x86 ma ponad 300 różnych opkodów) jest często zawodny. Doskonale nadaje się jednak do tłumaczenia języków o małym skomplikowaniu, jak na przykład kod bajtowy Java™, do postaci wysokopoziomowej. Należy zaznaczyć jednak iż celem dekompilacji nie musi być otrzymanie kodu bliskiego oryginałowi, a jedynie kodu zapisanego w języku wysokopoziomowym, którego analiza będzie szybsza oraz prostsza niż analiza kodu zapisanego w języku niskopoziomowym.

Dekompilacja kodu maszynowego z poprzedniego przykładu do języka C wyglądała by następująco:

```
unsigned int func(unsigned int a, unsigned int b)
{
  return a+b;
}
```

Kod ten mógł by jednak również zostać zdekompilowany do nieco innej postaci:

```
int func(int a, int b)
{
  return a+b;
}
```

Powyższy przykład ilustruje że nawet najprostszy kod może okazać się niejednoznaczny w przypadku dekompilacji, powstaje więc problem wyboru najlepszego wariantu. Dodatkowym problemem jest brak jednoznacznego standardu kompilacji kodu źródłowego do kodu maszynowego na procesorach CISC (lub pseudo-CISC, np. x86), przez co kompilatory różnych firm

generują bardzo różny kod, co utrudnia stworzenie uniwersalnego dekompilatora który współpracował by z plikami wykonywalnymi wygenerowanymi przez wszystkie kompilatory. Problemem jest również optymalizowanie kodu wynikowego przez kompilatory na poziomie kodu maszynowego. Przykładowo z uwagi na złożoność każdej instrukcji w procesorach z rodziny Intel, może być ona wykorzystywana do różnych celów w różnych momentach programu. Przykładowo instrukcja DEC (ang. decrement, zmniejsz o jeden) samodzielnie może być użyta do zmniejszenia zmiennej o jeden, a złożona z instrukcją JZ (ang. jump if zero, wykonaj skok jeśli wynikiem ostatniej operacji było zero) może posłużyć do sprawdzenia czy dana zmienna była równa jeden (wynikiem dekrementacji w takim wypadku było by zero, więc skok by się wykonał). Sprawę dodatkowo komplikuje fakt iż JZ wcale nie musi wystąpić bezpośrednio po DEC.

Z grupy dekompilatorów języka maszynowego z rodziny x86 do kodu w języku C, wartym wspomnienia jest dekompilator REC (Reverse Engineering Compiler, http://www.backerstreet.com/rec/), dostarczany z IDE RecStudio. Mimo iż posiada on nadal wiele wad, i rzadko jest używany do analizy wstecznej, to jest jednak pewnym krokiem naprzód.

Przykładowo poniższa funkcja skompilowana przy użyciu kompilatora MinGW GCC 3.4.5 (http://mingw.org)

```
int test(int a, int b)
{
  if(a > b)
  {
    a -= b;
    return a * 2;
  }
  else if(a == b)
  {
```

```
    return a - 3;
  }
  return b;
}
```

Została zdekompilowana do postaci:

```
/* Procedure: 0x004012E0 - 0x0040131B
 * Argument size: -4
 * Local size: 4
 * Save regs size: 0
 * Called by:
 * L0040131C()
 */

L004012e0(A8, Ac)
/* unknown */ void  A8;
/* unknown */ void  Ac;
{
/* unknown */ void  Vffffffc;

    if(A8 > Ac) {
        A8 = A8 - Ac;
        Vffffffc = A8 + A8;
    } else {
        Vffffffc = A8 == Ac ? A8 - 3 : Ac;
    }
    return Vffffffc;
}
```

Dekompilacja pod względem logicznym jest prawidłowa, jednak mimo trywialności funkcji, dekompilatorowi nie udało ustalić się typu argumentu funkcji.

Sytuacja o wiele lepiej ma się w świecie procesorów i maszyn wirtualnych typu RISC (Reduced Instruction Set Computer). Przykładem może być dekompiler kodu bajtowego Java™ do postaci języka Java™ JAD (http://www.kpdus.com/jad.html), rozwijany od roku 1999. Testem dla niego niech będzie ta sama funkcja co dla REC, jednak zapisana w Javie™:

```
public static int test(int a, int b)
{
  if(a > b)
  {
    a -= b;
    return a * 2;
  }
  else if(a == b)
  {
    return a - 3;
  }
  return b;
}
```

Wynikiem dekompilacji dekompilatorem JAD jest następująca funkcja:

```
public static int test(int i, int j)
{
    if(i > j)
    {
        i -= j;
        return i * 2;
    }
    if(i == j)
        return i - 3;
    else
        return j;
```

```
}
```

Jak można zauważyć, funkcja została zdekompilowana prawie perfekcyjnie, a jedyne zmiany względem oryginału dotyczą nazw zmiennych oraz dodatkowego „else" (który konstrukcji logicznej w żaden sposób nie zmienia. Kod uzyskany przy pomocy dekompilatora JAD może zostać bez zmian (w większości przypadków) skompilowany dowolnym kompilatorem Javy™.

Warto wspomnieć również o hybrydowym deasemblerze-dekompilatorze DeDe (http://www.balbaro.com/), którego głównym celem jest analiza budowy plików wykonywalnych Win32 PE utworzonych przez kompilator Borland® Delphi™. DeDe skupia się na poznaniu ogólnej struktury funkcji, klas oraz zdarzeń, natomiast nie tłumaczy kodu maszynowego na Delphi™, jedynie go deasembluje. Mimo tego, DeDe dostarcza analizującej osobie więcej informacji niż zwykły deasembler.

3.4 Analiza behawioralna

W przeciwieństwie do deasemblacji oraz dekompilacji, w których źródłem informacji o programie był statyczny plik binarny zawierający kod maszynowy, analiza behawioralna opiera się na informacjach zebranych podczas śledzenia wykonania programu oraz analizy komunikacji programu z systemem operacyjnym, innymi programami, operacji na plikach, rejestrze systemowym (w przypadku Microsoft®Windows®), oraz operacjach sieciowych. Narzędziami używanymi do analizy behawioralnej są przede wszystkim debuggery, oraz programy monitorujące komunikację. Dodatkowo, ze względów bezpieczeństwa podczas analizy kodu złośliwego, używane są środowiska chronione stworzone w oparciu o szeroko rozumianą wirtualizację, emulację, lub ograniczenie praw do operacji na systemie operacyjnym.

3.4.1 Debugger

Debugger jest programem, często interaktywnym, wspomagającym lokalizację oraz poznanie przyczyny błędu występującego w programie. Działanie debuggera opiera się o wykorzystanie możliwości oferowanych przez procesor oraz/lub system operacyjny w celu kontrolowania stanu wykonującego się programu. Większość debuggerów dla aplikacji działającej na poziomie użytkownika (tzw. ring 3) dla platformy x86/Win32 oferuje następujące możliwości:

- Zatrzymanie procesu gdy wykonanie dojedzie do określonego adresu w pamięci bądź linii kodu w określonym pliku (ang. breakpoint).
- Zatrzymanie procesu gdy odwoła się do określonego miejsca w pamięci w celu zapisu lub odczytu (np. dana zmienna zostanie zmodyfikowana).
- Wskazanie miejsca w którym wystąpił wyjątek (ang. exception, błąd krytyczny).
- Podejrzenie rejestrów procesora, pamięci procesu, oraz zmiana tychże.
- Podejrzenie listy załadowanych modułów (DLL, ang. Dynamic Link Library).
- Wykonanie krokowe programu, tj. zatrzymanie co jedną instrukcje.
- Utworzenie zapisu przebiegu programu (ang. trace).
- Wyświetlenie części programu jako kod natywnego języka bądź kod assemblera (jeśli debugger posiada wbudowany mini-deasembler).

Z technicznego punktu widzenia do realizacji powyższej funkcjonalności wykorzystywane jest połączenie funkcjonalności udostępnianych przez procesor oraz system operacyjny Microsoft®Windows®. W systemie Microsoft®Windows® debugger rozpoczynający działanie ma dwie możliwości:[24]

1. Uruchomić dany program jako tzw. proces debugowany, przy pomocy funkcji CreateProcess z flagą DEBUG_PROCESS.

2. Rozpocząć debugowanie istniejącego procesu (ang. attach), przy pomocy funkcji DebugActiveProcess.

Po uruchomieniu nowego procesu bądź podpięcia (ang. attach) się do procesu istniejącego, debugger wchodzi w główną pętle debuggera, w której przy pomocy funkcji WaitForDebugEvent oczekuje na informacje o pewnych ważnych dla debuggera zdarzeniach w procesie debugowanym, odpowiednio je obsługuje, a następnie kontynuuje wykonanie w procesie debugowanym przy pomocy funkcji ContinueDebugEvent, lub kończy działanie programu.[25]

Najważniejszymi wiadomościami dla debuggera są:

- EXCEPTION_DEBUG_EVENT – W procesie debugowanym wystąpił wyjątek (błąd krytyczny).
- CREATE_THREAD_DEBUG_EVENT – Proces debugowany stworzył nowy wątek.
- CREATE_PROCESS_DEBUG_EVENT – Proces debugowany stworzył nowy proces.
- EXIT_THREAD_DEBUG_EVENT – Jeden z wątków w procesie debugowanym zakończył działanie.
- EXIT_PROCESS_DEBUG_EVENT – Proces debugowany zakończył działanie.
- LOAD_DLL_DEBUG_EVENT – Proces debugowany załadował dodatkowy moduł DLL.
- UNLOAD_DLL_DEBUG_EVENT – Proces debugowany zwolnił moduł DLL.
- OUTPUT_DEBUG_STRING_EVENT – Proces debugowany prosi o wypisanie w debuggerze wiadomości.
- oraz inne.

Na czas obsłużenia wiadomości przez debugger, działanie wątku procesu debugowanego który wygenerował zdarzenie jest wstrzymywane. Debugger przy pomocy funkcji GetThreadContext oraz SetThreadContex może odczytywać oraz nadpisywać zawartość wszystkich rejestrów danego wątku w procesie debugowanym, łącznie z rejestrem EIP (wskaźnik instrukcji która będzie wykonana jako następna), oraz rejestrami debuggera DR0, DR1, DR2, DR3, DR6 oraz DR7. Dodatkowo przy pomocy funkcji ReadProcessMemory orz WriteProcessMemory debugger odczytywać oraz zapisywać dowolny fragment pamięci procesu debugowanego. Debugger posiada również prawa do tworzenia nowego wątku w kontekście procesu debugowanego, przy pomocy funkcji CreateRemoteThread, alokowania i zwalniania pamięci przy pomocy funkcji VirtualAllocEx oraz VirtualFreeEx, oraz zmieniania praw do odczytu/zapisu/wykonania konkretnym fragmentom pamięci dzięki funkcji VirtualProtectEx. Standardowa funkcjonalność breakpointów, czyli zatrzymywania programów we wskazanym miejscu, realizowana jest w następujący sposób:[25]

- Software Breakpoint (on execution) - przerwanie działania programu w konkretnym wskazanym miejscu, metoda programowa. Debugger przy pomocy ReadProcessMemory odczytuje jeden bajt kodu programu ze wskazanego miejsca, zapamiętuje go, a następne w to miejsce zapisuje opkod 0xCC, odpowiadający instrukcji INT3, czyli przerwaniu debuggera, korzystając z funkcji WriteProcessMemory. Gdy któryś z wątków wykona instrukcje INT3, zostanie zatrzymany, a debugger otrzyma wiadomość EXCEPTION_DEBUG_EVENT z parametrem EXCEPTION_BREAKPOINT. Za pomocą funkcji GetThreadContext debugger pobiera wartość rejestru EIP, uzyskując adres breakpointu, a następnie zapisuje pod uzyskany adres zapamiętany bajt (zastępując bajt INT3) przy pomocy WriteProcessMemory. Ilość programowych breakpointów jest w żaden sposób nie ograniczona. W celu ustawienia breakpointu na przedział adresów, zamiast na jeden konkretny adres, debugger nadpisuje dany przedział pamięci bajtami 0xCC (INT3), up-

rzednio je zapamiętując. Pewne wariacje tej funkcji zakładają użycie instrukcji innej niż INT3, która powoduje inny wyjątek, np. MOV [0], EAX spowoduje próbę zapisania zawartości rejestru EAX pod adres 0, do którego proces nie ma dostępu, co spowoduje wyjątek EXCEPTION_ACCESS_VIOLATION, o czym zostanie powiadomiony debugger.

- Hardware Breakpoint (on execution) – sprzętowe przerwanie działania programu we wskazanym miejscu. Debugger przy pomocy GetThreadContext pobiera zawartość rejestrów DR0, DR1, DR2, DR3, DR6 oraz DR7, a następnie, przy pomocy funkcji SetThreadContext, w wolny rejestr DR0-DR3 zapisuje adres w którym ma nastąpić przerwanie, a w rejestrze DR7 ustawia odpowiadającą wybranemu rejestrowi DR0-DR3 flagę L0-L3 (ang. local breakpoint enable), oraz ustawia tryb „Przerwij w wypadku wykonania" (ang. Break on instruction execution only) dla wybranego rejestru DR0-DR3 poprzez ustawienie pola R/W0-R/W3 na wartość b001. Z uwagi na dostępność jedynie czterech rejestrów dla każdego wątku procesu debugowanego, można ustawić jedynie cztery breakpointy sprzętowe. Metoda ta nie funkcjonuje pod systemami z rodziny Microsoft®Windows®9x (oprócz ME).

- NX bit / XD bit Breakpoint (on execution) – przerwanie działania programu gdy wykonanie dojdzie do danego miejsca. W nowych procesorach firmy AMD™oraz Intel®zostało wprowadzone nowe prawo dla strony w pamięci, oznaczone u powyższych producentów odpowiednio NX (ang. No eXecution) i XD (ang. eXecution Disabled). Możliwe jest cofnięcie prawa do wykonania danemu fragmentowi kodu procesu debugowanego przez debugger, przez co próba wykonania kodu w danym fragmencie spowoduje wystąpienie wyjątku, o czym zostanie powiadomiony debugger. Metoda ta wymaga jednak odpowiednio nowego procesora oraz włączenia DEP w systemie (ang. Data Execution Prevention).

- Software Memory Breakpoint (on read/write) – przerwanie działania programu gdy program odczyta dany fragment pamięci, lub gdy zapisze do niego. Debugger przy pomocy funkcji VirtualProtectEx odbiera prawo do odczytu/zapisu danemu fragmentowi pamięci (w przypadku procesorów Intel®x86 istnieje możliwość nadania danego prawa jedynie całej stronie, w większości przypadków strona ma wielkość 4096 bajtów (0x1000)). Gdy program debugowany będzie chciał zapisać lub odczytać dany fragment pamięci, wystąpi wyjątek EXCEPTION_ACCESS_VIOLATION (#PF (Page Fault) w terminologii Intela®), o czym zostanie powiadomiony debugger. Debugger może przywrócić prawa do danych stron pamięci korzystając ponownie z VirtualProtectEx.

- Hardware Memory Breakpoint (on read/write) – przerwanie działania programu gdy program odczyta dany fragment pamięci, lub gdy zapisze do niego. Debugger postępuje analogicznie jak w przypadku Hardware Breakpoint (on execution). Jedyną różnicą jest inne ustawienie pola (b01 – zapis, b11 - zapis/odczyt) R/W0-R/W3.

- Step – „wykonaj jedną instrukcję i się zatrzymaj". Procesory z rodziny Intel®x86 w rejestrze flagowym EFLAGS posiadają flagę TF (ang. Trap Flag, Flaga Pułapki), po której ustawieniu, procesor wykona jedną instrukcję, po czym zgłosi przerwanie debuggera (#DB (Debug) w terminologii Intela). Debugger korzystając z SetThreadContext może ustawić tą flagę, a następnie odczekiwać na wyjątek EXCEPTION_SINGLE_STEP. Inną możliwością jest tworzenie breakpointu (on execution) na następną instrukcję, lub instrukcję która nastąpi po skoku, jednak jest to metoda bardziej kosztowna.

Dobre debuggery pozwalają również tworzyć breakpointy warunkowe, czyli do powyższej funkcjonalności dodawać możliwość postawienia warunku, takiego jak na przykład „tylko gdy EAX jest równe 10" lub „gdy flaga ZF jest ustawiona".

Najczęściej używanym obecnie interaktywnym debuggerem ring 3 jest OllyDbg autorstwa Oleha Yuschuka (http://www.ollydbg.de/), obecnie w wersji 1.10 (wersja 2.0 jest w przygotowaniu). Debugger ten oferuje zarówno pracę z plikami wykonywalnymi (w formie procesu), jak i z bibliotekami DLL. Z ciekawszych funkcji debuggera można wymienić:

- możliwość rozszerzenia funkcjonalności poprzez system wtyczek (ang. plug-in)
- podgląd załadowanych modułów DLL
- podgląd mapy pamięci zaalokowanej na potrzeby procesu
- operowanie na dowolnym wątku procesu
- śledzenie okien należących do procesu
- śledzenie uchwytów (ang. handle) należących do procesu
- podgląd łańcucha SEH (ang. structured exception handling – mechanizm obsługi wyjątków w systemach Microsoft Windows)
- podgląd stosu wywołań (ang. call stack)
- tworzenie trace-logu
- możliwość zmieniania kodu programu w trakcie działania (wbudowany assembler)
- możliwość wydawania poleceń za równo z GUI (ang. graphical user interface, graficzny interfejs użytkownika), jak i z wbudowanej linii poleceń
- oraz inne

Debugger OllyDbg wykorzystuje ponadto heurystykę oraz logikę rozmytą do analizy kodu programu, celem wyszukania i wyodrębnienia funkcji w analizowanym programie, udostępnia mechanizmy kolorowania składni assemblera, oraz wykrywa odwołania do zmiennych lokalnych, argumenty znanych funkcji, oraz wywołania funkcji importowanych.

Inną rodziną są debuggery ring 0, czyli działające na poziomie systemu operacyjnego, a nie pojedynczego procesu. Wymienić tutaj należy przede wszystkim popularnego niegdyś debuggera SoftICE firmy NuMega Technologies, oraz debugger systemowy Microsoftu®WinDbg. Jednak z uwagi na fakt iż debuggery te obejmują swoim działaniem cały system, a nie pojedynczy proces, są rzadziej wykorzystywane przy analizowaniu aplikacji.

3.4.2 Programy monitorujące

Programy monitorujące służą do monitorowania zachowania procesu pod względem wywołań funkcji systemowych, korzystania z plików, rejestru, usług sieciowych, przesyłania wiadomości między procesem a innymi aplikacjami / systemem, monitorowania posiadanych przez proces zasobów, oraz innych interesujących z punktu widzenia analizy zdarzeń.

Z technicznego punktu widzenia, programy monitorujące korzystają z funkcjonalności udostępnionej przez system operacyjny (np. hooków systemowych), bądź zawierają własne mechanizmy pozwalające na śledzenie przebiegu informacji, poprzez przykładowo podpięcie się do śledzonego procesu i podmianie funkcji systemowych na funkcje logujące, skorzystanie z możliwości oferowanych przez działanie w trybie ring 0 (systemowym) i podmianę funkcji systemowych (ang. syscall hook), bądź skorzystanie z innego, np. autorskiego, sposobu.

Poszczególne techniki mogą być realizowane w następujące sposoby:

Technika śledzenia wiadomości odbieranych przez aplikację może zostać zrealizowana w następujący sposób:

1. Program monitorujący tworzy globalny hook systemowy, przy pomocy funkcji SetWindowsHookEx, podając jako parametry WH_GETMESSAGE oraz własny moduł DLL zawierający funkcję która obsłuży dany hook.

2. Gdy dowolna aplikacja wywoła funkcję GetMessage lub PeekMessage (służące do odbierania wiadomości), działanie programu zostanie na chwile wstrzymane, i zostanie wywołana funkcja obsługująca hook WH_GETMESSAG w kontekście aplikacji odbierającej wiadomość.

3. Funkcja obsługująca hook loguje informacje o typie wiadomości, jej parametrach, oraz oknie i aplikacji do której wiadomość przyszła.

4. Program jest kontynuowany.

Technika przekierowania funkcji systemowej może zostać zrealizowana w następujący sposób:

1. Program monitorujący otwiera (OpenProcess) lub tworzy nowy proces (CreateProcess), po czym go wstrzymuje (tj wszystkie jego wątki).

2. Program monitorujący alokuje w kontekście procesu fragment pamięci z pełnymi prawami dostępu (odczyt/zapis/wykonanie) za pomocą funkcji VirtualAllocEx, po czym kopiuje swój wcześniej przygotowany kod w zaalokowane miejsce.

3. Za pomocą funkcji CreateRemoteThread program tworzy nowy wątek w procesie monitorowanym, w którym uruchamia wcześniej skopiowany kod.

4. Nowy wątek, działający w kontekście monitorowanej aplikacji, odnajduje moduł (LoadLibrary) i adres funkcji (GetProcAddress) która go interesuje, a następnie na początku tej funkcji zapisuje instrukcje skoku (lub wywołania) do własnej funkcji.

5. W przypadku wywołania, własna funkcja zapisze parametry z jakimi funkcja została wywołana, wywoła oryginalną funkcję, po czym zapisze co funkcja zwróciła i powróci do miejsca w oryginalnym programie który funkcję monitorowaną wywołał.

Metoda ta, jeśli dobrze zaimplementowana, jest w pełni transparentna dla procesu monitorowanego.

Kolejną techniką monitorowania jest podmiana funkcji systemowych na poziomie jądra systemu (ang. syscall hook). Technika ta może zostać zrealizowana w następujący sposób:[14]

1. Program monitorujący uruchamia swój sterownik monitorujący.
2. Sterownik przy inicjacji podmienia adresy wybranych funkcji systemowych, np. NtCreateFile, w tablicy syscalli (symbol _KiServiceTable) na własną funkcję.
3. Program monitorujący otwiera kanał komunikacyjny ze sterownikiem (np. przy pomocy CreateFile), i oczekuje na informacje.
4. Gdy jakiś program wywoła (pośrednio zapewne) syscall NtCreateFile, zostanie uruchomiona funkcja monitorująca, która zapisze interesujące ją parametry, i wywoła oryginalną funkcję NtCreateFile.
5. Program monitorujący poprzez kanał komunikacyjny odbierze informacje o wywołaniu.

Powyższa technika źle zrealizowana może spowodować niestabilność systemu, włącznie z wystąpieniem BSoD (ang. Blue Screen of Death).

Często wykorzystywanymi gotowymi programami monitorującymi są:

- Filemon firmy Sysinternals – służący do śledzenia odwołań do plików.
- Regmon firmy Sysinternals – służący do śledzenia odwołań do rejestru systemowego.

- kerberos autorstwa Rustema Fasihova – służący do śledzenia wywołań funkcji zawartych w bibliotekach DLL przez program monitorowany.
- Spy++ firmy Microsoft®– służący do monitorowania wiadomości wysyłanych do i przez monitorowane okno należące do danej aplikaci.
- oSpy autorstwa Ole Andre Vadla Ravnas – służący do monitorowania komunikacji sieciowej oraz wywołań funkcji kryptograficznych przez monitorowaną aplikację.
- ProcessExplorer firmy Sysinternals – służący do zarządzania procesami uruchomionymi w procesie.
- oraz inne.

Monitorowanie zachowania procesu daje bardzo dużo informacji o nim, ale w przypadku kodu złośliwego może być niebezpieczne. Jeśli wirus zostanie uruchomiony, nawet na chwile, natychmiast może rozpocząć proces samopowielania, kradzieży informacji, czy blokowania systemu. Wymogiem stało się znalezienie mechanizmu który pozwalałby na uruchomienie kodu złośliwego celem analizy behawioralnej, a który zabezpieczałby przed szkodliwym działaniem wynikającym z natury kodu złośliwego.

3.4.3 Emulator, wirtualizer, sandbox

Emulator jest to program komputerowy, który duplikuje funkcje jednego systemu informatycznego w innym, dzięki czemu drugi system upodabnia się w działaniu z pierwszym[23]. Przykładem emulatora może być np. UAE który emuluje system komputerowy Amiga. Emulator symuluje działanie wybranej funkcjonalności, nie korzystając jednak bezpośrednio z możliwości sprzętu na którym jest uruchamiany, co umożliwia przenoszenie emulatorów między różnymi platformami. Przykładowo emulator procesora x86 nie wymaga procesora x86 żeby funkcjonować. Takie podejście powoduje niestety poważny spadek prędkości emulowanego systemu względem oryginału, ale umożliwia implementacje dodatkowej funkcjonalności, niedostępnej

nawet na oryginalnej platformie.

Przykładowymi emulatorami są:

- bochs (http://bochs.sourceforge.net/) - Emulator platformy x86, z wbudowanym debuggerem. Przeznaczony m.in. do wspomagania tworzenia i analizy systemów operacyjnych.

- DOSBox (http://dosbox.sourceforge.net/) – Emulator platformy x86 oraz systemu MS-DOS®, przeznaczony głównie do emulacji programów działających pod MS-DOS®na obecnych systemach operacyjnych.

Oba powyższe emulatory są udostępnione na licencji typu Open Source, zgodnej z polityką FSF (ang. Free Software Fundation), dzięki czemu są one używane do tworzenia dedykowanych środowisk wspomagających analizę.

Wirtualizer (ang. virtualizer) jest programem który umieszcza dodatkową abstrakcyjną warstwę między dwie inne, celem wirtualizacji pewnej funkcjonalności. Przykładowo umieszczenie abstrakcyjnej warstwy między systemem operacyjnym a programem może zostać użyte do stworzenia wirtualnego (w sensie systemu plików lub systemu sieciowego) środowiska pracy dla danej aplikacji. Wirtualizacja, w przeciwieństwie do emulacji, jest ściśle związana z daną platformą. Przykładowymi wirtualizerami są:

- Microsoft®Virtual PC – darmowy wirtualizer systemu komputerowego x86/x64, umożliwia uruchomienie kolejnych systemów operacyjnych (działających równolegle) na danym systemie komputerowym. Program jest przeznaczony dla systemów Microsoft®Windows®.

- vmware®ESX Server – wirtualizer systemu komputerowego, umożliwia uruchomienie kilku systemów operacyjnych działających równolegle na danym systemie komputerowym. Nie wymaga dodatkowego systemu operacyjnego.

- QEmu (http://fabrice.bellard.free.fr/qemu/) – darmowy, Open Source, wirtualizer systemu komputerowego. Program przeznaczony jest dla systemów opartych o kernel Linux, Microsoft®Windows®, oraz innych.

Wirtualizery, z uwagi na fakt wykorzystania istniejących, prawdziwych, elementów systemu komputerowego do funkcjonowania (np. procesora) są dużo szybsze niż emulatory. Wirtualizery są niezwykle użyteczne przy analizie kodu złośliwego. Umożliwiają stworzenie wirtualnego systemu komputerowego, na którym uruchamiany jest następnie kod złośliwy, razem ze wszelkimi dostępnymi programami monitorującymi. Mimo iż kod złośliwy może zniszczyć środowisko w którym pracuje, to i tak zniszczy tylko wirtualny system, który w przeciwieństwie do rzeczywistego, można odtworzyć w kilkanaście sekund. Nawet w wypadku zniszczenia przez kod złośliwy środowiska, wyniki monitoringu bardzo często są możliwe do odzyskania, bądź są od razu przekazywane do systemu rzeczywistego.

Sandbox jest programem, często modułem w systemie operacyjnym, wirtualizującym środowisko operacyjne danego procesu, ograniczając jego dostęp do rzeczywistych plików, informacji o rzeczywistym systemie czy komunikacji z innymi procesami uruchomionymi poza sandboxem. Celem sandboxu jest ograniczenia ewentualnych szkód wynikłych z nieprawidłowego / złośliwego działania uruchomionego programu, oraz ograniczenie szkód powstałych w wyniku udanego przełamania zabezpieczeń usług oferowanych przez dany system komputerowy.

Powyższe technologie są często wykorzystywane w analizie kodu złośliwego, zarówno automatycznej jak i ręcznej.

4 Przeciwdziałanie analizie wstecznej

Wraz z popularyzacją techniki analizy wstecznej, zaczęły powstawać mechanizmy mające utrudnić bądź uniemożliwić analizę wsteczną. Badaniami nad powyższymi mechanizmami zajęły się głównie dwie grupy, których produkty były najczęściej reversowane:

- autorzy oprogramowania komercyjnego – których oprogramowanie było reversowane celem obejścia mechanizmów anty-pirackich (software cracking)
- autorzy wirusów komputerowych oraz innego kodu złośliwego – których radosną twórczością zajęli się specjaliści z firm antywirusowych

Chciałbym w tym rozdziale zapoznać czytającego z najpopularniejszymi metodami utrudniania analizy wstecznej oprogramowania, dokonywanej przy użyciu narzędzi wymienionych w poprzednim rozdziale, oraz metod obejścia/likwidacji utrudnienia.

4.1 Anty-deasemblacja

Celem mechanizmów utrudniających deasemblacje jest wprowadzenie w błąd deasemblera, tak aby w wynik deasemblacji był sfałszowany lub niekompletny, tak aby prawdziwy kod został zatajony przed czytającym.

Zdecydowanie najprostszym sposobem na oszukanie standardowych deasemblerów jest wykonanie skoku, którego adresem docelowym jest środek innej, przeanalizowanej wcześniej instrukcji, lub samego skoku[8]. Przykładem może być poniższy kod maszynowy:

```
EB FF C8 BB 05 00 00 00
```

Zostanie zdeasemblowany (np. przez ndisasm) do postaci:

```
00000000  EBFF              jmp short 0x1
00000002  C8BB0500          enter 0x5bb,0x0
00000006  0000              add [eax],al
```

Deasembler analizował bajt po bajcie, w związku z czym po przetłumaczeniu pierwszej instrukcji EB FF na jmp short 0x1, zajmującej dwa bajty, rozpoczął analizę następnej dwa bajty dalej, czyli od bajtu C8 na offsecie 0x02. Natomiast z punktu widzenia procesora wykonującego powyższy kod, po zdekodowaniu pierwszej instrukcji zostanie wykonany skok na offset 0x01, czyli następna instrukcja będzie dekodowana od offsetu 0x01, a nie, jak założył deasembler 0x02. Kod który się wykona jest następujący:

```
00000000  EBFF              jmp short 0x1
00000001  FFC8              dec eax
00000003  BB05000000        mov ebx,0x5
```

Należy zauważyć że bajt na offsecie 0x01, czyli FF, jest za równo argumentem instrukcji skoku JMP, jak i początkiem instrukcji dekrementacji DEC.

Powyższy sposób działa za równo na deasemblery nie śledzące skoków, jak i na śledzące które zaznaczają bajty już poddane analizie wcześniej.

W przypadku wystąpienia powyższego triku w kodzie analizowanym przez ndisasm, uzyskanie prawidłowego wyniku jest niemożliwe bez zmian kodu ndisasma. Natomiast w przypadku interaktywnych deasemblerów, wystarczy cofnąć oznaczenie pierwszej instrukcji (instrukcji skoku) jako kodu, i uruchomić analizę od bajtu FF.

Analogiczne podejście zakłada ukrycie adresów skoków przed deasemblerem poprzez skorzystanie z pośredniego adresowania skoków. Taki skok może celować przykładowo w środek innej instrukcji (wg. założeń deasemblera o umiejscowieniu początku instrukcji). Przykładem może być poniższy kod:

```
mov eax, ADRES
jmp eax
lub
push ADRES
ret
```

Deasemblery nowej generacji z wbudowanym szczątkowym CPU potrafią sobie poradzić z powyższym zabezpieczeniem, po za tym bardzo pomocne są debuggery.

Innym podejściem do problemu jest użycie kodu samo modyfikującego się (SMC, ang. Self Modifying Code). Kod samo modyfikujący się, jak sama nazwa wskazuje, jest kodem który w trakcie wykonywania programu zmienia jego treść. SMC używa jest na dwa sposoby. Pierwszym sposobem jest użycie pojedynczych instrukcji, które modyfikują kolejną instrukcje, lub kilka kolejnych instrukcji[10]. Przykładem może być bardzo prosty kod:

```
B8 08 00 00 00 C6 00 6A 16 50
```

Który zostanie przetłumaczony przez standardowy deasembler na:

```
00000000  B808000000        mov eax,0x8
00000005  C6006A            mov byte [eax],0x6a
00000008  16                push ss
00000009  50                push eax
```

Natomiast gdyby powyższy kod został uruchomiony na procesorze, instrukcja znajdująca się pod adresem 00000005 zmodyfikuje obie kolejne instrukcje, wstawiając pod adres 00000008 bajt 6A, zastępując znajdujący się tam bajt 16. Prawdziwy wykonany kod wyglądał by następująco:

```
00000000  B808000000        mov eax,0x8
00000005  C6006A            mov byte [eax],0x6a
00000008  6A50              push byte +0x50
```

Druga metoda zakłada posunięcie się o krok dalej, i zaszyfrowanie całego kodu programu (oraz/lub danych), a następnie odszyfrowanie go na początku wykonania. Formą „szyfru" może być w tym miejscu również kompresja, której dodatkową zaletą jest zmniejszenie objętości kodu. Metoda ta jest obecnie bardzo popularna i szeroko rozpowszechniona, powstało również wiele programów automatyzujących kompresję i szyfrowanie programów wykonywalnych, jak np. UPX czy Yoda's Protector. Określane są one mianem packerów bądź protektorów.

Uzyskanie prawidłowego wyniku deasemblacji w przypadku powyższych metod możliwe jest na kilka sposobów. Pierwszy zakłada uruchomienie programu (na rzeczywistej maszynie lub emulatorze) i, po wykonaniu całkowitym programu, lub wcześniej, zapisanie (zrzucenie, ang. dump) kodu z pamięci z powrotem na dysk. Sposób ten jest bardzo skuteczny, jednak, w przypadku uruchomienia kodu na prawdziwym komputerze, dość niebezpieczny, jeśli analizowany jest kod niewiadomego pochodzenia, lub o złośliwym działaniu. Pewnym problemem może być tutaj również wyznaczenie miejsca w którym należy zrzucić kod, szczególnie jeśli program modyfikuje dane miejsce kilka-kilkanaście razy, lub np. zaciera kod za sobą nadpisując go losowymi danymi. Warto wspomnieć w tym miejscu o programach automatyzujących zrzucanie pamięci, takich jak procdump, LordPE, czy pluginy do PEiD oraz OllyDbg (OllyDump). Warto dodać również iż powstały pewne metody które zapobiegają prawidłowemu stworzeniu zrzutu przez te programy, takie jak czyszczenie nagłówków PE w pamięci (z których korzysta LordPE), wczytanie drugi raz modułu programu w przestrzeń tego samego procesu i wykonanie w nowym miejscu, czy przeniesienie kodu programu z przestrzeni obrazu na stos lub na heap którego programy zazwyczaj nie zrzucają.

Drugi sposób zakłada modyfikowanie kodu w miarę deasemblacji przez osobę reversującą, czy to ręcznie, czy też przez wcześniejszą analizę algo-

rytmu i napisanie (skorzystanie z gotowego) tzw. unpackera / deprotectora. Metoda ta jest bezpieczniejsza i odrobinę skuteczniejsza od poprzedniej, jednak jest o wiele bardziej czasochłonna (sekundy przeciw godzinom).

Kolejnym sposobem jest stworzenie wirtualnej maszyny oraz przetłumaczenie całości bądź części kodu programu na kod wirtualnej maszyny. Wtedy analizujący musi skupić się na analizie pojedynczych instrukcji VM, oraz mechanizmów dekodowania instrukcji, a następnie zdekodować program i dopiero rozpocząć docelową analizę. Jest to obecnie najskuteczniejsza metoda zabezpieczenia oprogramowania, szczególnie w wydaniu zakładającym wykorzystanie wielu zagłębionych maszyn wirtualnych. Analiza tak zabezpieczonego kodu jest trudna, i póki co nie udało mi się znaleźć informacji o sposobie który byłby w miarę szybki oraz skuteczny. W przyszłości rozwiązania mogą oprzeć się o metody heurystyczne, sztuczną inteligencję, bądź zaawansowane emulatory natywnego procesora.

Ostatnia metoda którą opisze polega na wykorzystaniu mechanizmu obsługi wyjątków do stworzenia systemu obsługującego teoretycznie nieprawidłowe instrukcje, które jednak, dzięki temu systemowi, będą prawidłowe. Mimo iż deasembler nie będzie potrafił rozpoznać instrukcji, faktycznie zostanie ona wykonana prawidłowo, mimo iż tak na prawdę za jej wykonanie nie będzie odpowiedzialny CPU, a procedura obsługująca wyjątek. Przykładowy mechanizm działania wyglądał by następująco:

1. Przy uruchomieniu, program umieszcza adres swojej procedury na stosie SEH

2. Dalsze wykonanie przebiega normalnie, aż do natrafienia nieprawidłowej instrukcji. Wtedy procesor generuje wyjątek #UD, czyli Invalid Opcode.

3. System operacyjny w kontekście procesu wywołuje procedurę znajdującą się na stosie SEH, dając jej do dyspozycji cały kontekst wątku.

4. Procedura dekoduje opcode który spowodował błąd, oraz go wykonuje na kontekscie procesu, przenosząc EIP wątku na kolejną instrukcję.

5. System powraca do głównego wątku kontynuując działanie.

Tak zbudowany mechanizm wymaga ręcznej analizy w celu poznania faktycznego działania opcodeów.

Istnieją również inne metody anty-deasemblacji, jednak w większości są one wariacją którejś z powyższych metod.

4.2 Anty-debugging

Mechanizmy anty-debugger mają za cel wykryć czy program jest debugowany, celem ewentualnego wczesnego zakończenia programu, przekierowania ścieżki wykonania na nieprawdziwy tor, lub unieszkodliwienie debuggera. Mechanizmy anty-debug można podzielić na dwa rodzaje. Pierwszy skierowany jest przeciwko mechanizmom wspomagającym debug udostępnianym przez procesor oraz system, a drugi przeciwko konkretnym debuggerom.

Najprostszą metodą zaliczającą się do pierwszej grupy jest jest odpytanie systemu. Moduł kernel32 udostępnia w tym celu funkcję IsDebuggerPresent, zwracającą TRUE jeśli debugger jest obecny. Najprostszą metodą obejścia tej metody jest zmodyfikowanie funkcji tak aby zawsze zwracała odpowiedź negatywną.[10]

Kolejne metody opierają się na detekcji kolejnych metod ustawiania breakpointów:[8]

- Software breakpoint (on execution) – Proces stara się wykryć czy opcode INT3 (0xCC), lub inny opcode powodujący wyjątek, został gdzieś umieszczony. Metoda ta realizowana jest albo przez sprawdzanie okre-

ślonych miejsc (głównie początków funkcji), lub sprawdzenie sumy kontrolnej całego kodu (jeśli instrukcja INT3 zastąpi jakiś element kodu, suma kontrolna nie będzie się zgadzać).

- Hardware breakpoint – Proces pobiera za pomocą funkcji GetThreadContext, lub wywołania SEH, wartości rejestrów debuggera, po czym sprawdza Dr7 czy któryś breakpoint sprzętowy jest aktywny.
- NX bit / XD bit breakpoint, Software Memory Breakpoint – Proces odpytuje system, przy pomocy funkcji VirtualQuery, o prawa dostępu do swojej pamięci celem sprawdzenia czy debugger nie zmienił praw dostępu do pamięci.
- Step – Proces sprawdza czy flaga TF w rejestrze EFLAGS jest ustawiona. Alternatywną metodą jest pomiar czasu, np. przy pomocy instrukcji rtdsc, który mija między kilkoma instrukcjami. Czas większy niż kilka cykli procesora może wskazywać na uruchomiony tryb krokowy.

Deaktywacja każdej z powyższych metod opiera się przede wszystkim na jej zlokalizowaniu. Wówczas można ją usunąć z kodu, lub zmodyfikować funkcje systemowe, tak aby mechanizmy przestały funkcjonować (np. sprawić aby GetThreadContext nie zwracało prawdziwych informacji).

Standardowymi metodami używanymi do wykrycia konkretnych programów debugujących jest próba znalezienia znanej nazwy procesu w liście procesów (np. gdb.exe lub ollydbg.exe), klasy okna debuggera (np. OLLYDBG), mutexów używanych przez debugger, lub sprawdzenie czy rodzicem procesu nie jest przypadkiem jakiś znany debugger. Obejście tych metod opiera się na zlokalizowaniu ich i deaktywowaniu w kodzie, lub odpowiedniej modyfikacji debuggera aby przestał być on wykrywany, np. poprzez zmianę nazwy jego pliku wykonywalnego, lub podmiana nazwy klasy okna w pliku wykonywalnym debuggera.

Przykładem metody dedykowanej przeciwko OllyDbg było wykorzystanie błędu typu format-bug, który umożliwiał spowodowanie błędu w debuggerze poprzez odpowiednie wywołanie funkcji OutputDebugString. Warto zaznaczyć że w momencie pisania tego rozdziału błąd nadal nie został załatany.

Niektóre debuggery, jak np. gdb, modyfikują środowisko uruchomienia procesu. Przykładowo prosty program wypisujący linie poleceń (puts(argv[0])) uruchomiony normalnie wypisze np.:

```
>test.exe
test.exe
```

Natomiast ten sam program uruchomiony z gdb wypisze:

```
>gdb -q test.exe
(gdb) r
Starting program: D:\code\gynvael\SkySong\rnd\gdb_win/test.exe
D:\code\gynvael\SkySong\rnd\gdb_win/test.exe
```

Niestety ten błąd, w przypadku gdb, jest przenoszony z platformy na platformę, w związku z czym jest skuteczny nie tylko w przypadku platformy Microsoft®Windows®, ale również w przypadku np. platform Linuxowych.

Obejście metod dedykowanych jest zazwyczaj czasochłonne, ponieważ wymaga poprawienia programu debuggera, po uprzednim zlokalizowaniu błędu.

4.3 Metody przeciwdziałania monitorowaniu

Najprostszymi metodami ogólnego przeznaczenia, są próby wykrycia czy w systemie jest uruchomiony proces o znanej nazwie należącej do programu monitorującego (np. filemon.exe), lub wykrycia okna o znanej klasie lub nazwie (np. File Monitor). Metody te są oczywiście proste do obejścia.

Pozostałe metody skupiają się na próbie wykrycia poszczególnych mechanizmów monitorowania.

Przykładowo metodą na wykrycie czy funkcje API są monitorowane, jest sprawdzenie po ich kodzie czy nie zostały przekierowane (skok na początku funkcji), lub czy adresy funkcji w IAT obrazu procesu zgadzają się z adresami w EAT danego modułu. Najlepszą metodą obejścia tych metod jest ich deaktywacja w kodzie.

Sytuacja komplikuje się w przypadku gdy część monitorująca znajduje się w ring 0, szczególnie jeśli program nie ma własnych sterowników i nie ma dostępu do ring 0. Przykładowa metoda wykrycia podmiany syscalla mogła by polegać na pomiarze minimalnego czasu realizacji danego syscalla, jednak ta metoda mogła by być nieskuteczna na systemach wieloprocesorowych lub przy dużym obciążeniu systemu.

4.4 Sposoby przeciwko emulatorom, wirtualizerom oraz sandboxom

Odwiecznym „świętym Gralem”[11] dla autorów mechanizmów anty-VM, anty-sandbox oraz anty-emu jest przedostanie się ze środowiska chronionego do środowiska rzeczywistego (środowiska gospodarza). Może to zostać osiągnięte jedynie w wyniku błędu w implementacji, bądź architekturze systemu chroniącego, ale jak historia szeroko pojętego hackingu pokazuje, takie rzeczy już się zdarzały. Udane przedostanie się do środowiska rzeczywistego jest oczywiście bardzo niebezpieczne, i często prowadzi do pełnego przejęcia kontroli nad systemem gospodarza. Należy zaznaczyć jednak iż takie luki występują rzadko i są bardzo szybko łatane, niemniej jednak istnieją.

Inne podejście zakłada, podobnie jak w przypadku debuggerów, wykrycie czy proces jest w środowisku chronionym, oraz sfałszowanie przebiegu w przypadku pozytywnego wyniku. Większość metod jest dedykowane przeciwko

jednemu specyficznemu programowi (wirtualizerowi, emulatorowi, itd), niektóre jednak są pozwalają na detekcje kilku programów danego typu. Działanie metod opiera się w większości na znalezieniu artefaktów pozostawionych w pamięci przez program wirtualizujący/emulujący, sprawdzeniu (stałej w większości wypadków) listy sprzętu podłączonego do wirtualnego komputera, lub wykorzystanie obserwowalnych nieprawidłowości/zaburzeń względem prawdziwego systemu.

Przykładem metody pozwalającej wykryć większość (wszystkie?) wirtualizery jest Red Pill stworzony przez Joanne Rutkowską[16]. Metoda ta opiera się o wywołanie instrukcji SIDT która pobiera adres systemowej tablicy IDTR. Wg. badań pani Rutkowskiej, adresy zwracane w przypadku wirtualizacji są dużo wyższe niż na rzeczywistym systemie.

Metodą dedykowaną przeciwko wirtualizerowi Microsoft®VirtualPC jest na przykład poprzedzenie dowolnego opcode kilkunastoma prawidłowymi prefiksami REP lub REPNE (dekoder procesora zezwala użycie więcej niż jednego prefiksu z danej grupy, w tej sytuacji obowiązuje ostatni występujący), tak aby instrukcja przekroczyła wielkością 15 bajtów, co spowoduje wyjątek #UD (Invalid Opcode) na rzeczywistym procesorze z rodziny Intel x86, jednak Microsoft®VirtualPC nie generuje tego wyjątku. Metoda ta okazała się również być skuteczna przeciwko wirtualizerowi QEMU. Należy również wspomnieć o metodzie PurplePill, opracowanej przez hackera o pseudonimie j00ru, opartej na użyciu wielokrotnych prefiksów przeciążenia segmentów, pozwalającej, oprócz detekcji VirtualPC i QEMU, również na detekcję emulatora bochs.

Bardzo często zdarza się iż wirtualizer/emulator podaje nieprawdziwe/błędne dane o typie procesora działającego w wirtualnej maszynie. Przykładem może być ponownie Microsoft®VirtualPC który jako nazwę procesora podaje

„ConnectixCPU”[12].

Temat detekcji maszyn wirtualnych jest bardzo szeroki, i obecnie nie istnieje (mimo zapewnień niektórych producentów) emulator lub wirtualizer niemożliwy do wykrycia znanymi metodami.

4.5 Inne metody utrudnienia analizy

Analiza może zostać utrudniona również przez zastosowanie niestandardowego zapisu formatu pliku wykonywalnego, zgodnego z dokumentacją oraz obsługiwanego przez system operacyjny, ale nie obsługiwanego przez debugger, deasembler czy inne programy wspomagające analizę.

Przykładem może być plik wykonywalny wygenerowany przez crinkler (http://www.crinkler.net/), kompresujący linker stworzony na potrzeby demosceny. Jego głównym celem jest stworzenie jak najmniejszych, jak najbardziej zwartych plików wykonywalnych, często nieprzekraczających wielkością 4KB. W tym celu, oprócz kompresji kodu i danych, stosuje on triki mające na celu zmniejszenie wielkości nagłówka, takie jak umieszczenie nazw modułów DLL jako nazwy sekcji, pominięcie podprogramu MS-DOS®, czy niestandardowe wykorzystanie wolnego miejsca w nagłówkach. Próba podejrzenia budowy pliku wykonywalnego stworzonego przez crinklera np. w Fileinfo (plug-in do Total Commandera), czy PEView powoduje błąd krytyczny w obu programach.

Innym utrudnieniem często jest zaśmiecenie kodu, poprzez wstawienie dużej ilości skoków lub instrukcji zbędnych. Pewną modyfikacją tej metody jest również zapis standardowych instrukcji przy pomocy mniej standardowych odpowiedników. Przykładem może być instrukcja „MOV EAX, 1” która może zostać zapisana jako:

```
LEA EAX, [1]
```

lub

```
XOR EAX, EAX
INC EAX
```

lub

```
PUSH 1
POP EAX
```

Itp.

Jako ciekawostkę można dodać pewien sposób oddziałujący na ludzką psychikę, oparty na umieszczeniu w kodzie prośby o nie reversowanie lub nie crackowanie danego produktu. Innym podejściem do tematu jest umieszczenie w kodzie groźby o konsekwencjach złamania licencji produktu poprzez jego reversowanie, ale takie aplikacje działają na software crackerów jak płachta na byka.

4.6 Pakery, protektory

Z uwagi na mnogość metod utrudniających analizę zaistniała potrzeba stworzenia narzędzi automatyzujących proces zabezpieczania aplikacji. Działanie aplikacji zabezpieczających, określanych mianem protektorów, opiera się na konwersji wejściowego, niezabezpieczonego pliku wykonywalnego, zaszyfrowanie go (często z losowym kluczem), opcjonalną kompresje, oraz dodanie kodu który odszyfruje oryginalny kod podczas uruchomienia programu. Kod deszyfrujący jest zazwyczaj dobrze (wg. autorów) zabezpieczony przez analizą wsteczną. Protektory są używane głównie przez producentów komercyjnego oprogramowania, oraz przez autorów kodu złośliwego, którzy dzięki protektorom, oprócz utrudnienia analizy wstecznej, zmieniają również „wygląd" kodu wirusa/robaka, przez co nie zostanie on znaleziony przez skanery antywirusowe, które przykładowo w bazie mają sygnaturę jedynie jego niezabezpieczonej wersji. W związku z ostatnim, powstała potrzeba

tworzenia deprotektorów które będą używane do konwersji wersji zabezpieczonej na niezabezpieczoną, dzięki czemu możliwe będzie sprawdzenie sygnatury kodu złośliwego. Deprotektory są również używane przez osoby zajmujące się software crackingiem, do odbezpieczania kodu zabezpieczonego, więc koło się zamyka. Przykładem protektórów mogą być ASProtect, Yoda's Protector, PolyEnE, oraz inne.

Warto wspomnieć również, że większość protektorów tworzy polimorficzny bądź metamorficzny kod deszyfrujący, czyli zabezpieczenie dwóch kopii tego samego pliku da w wyniku dwa różne (pod względem zawartości binarnej) pliki wykonywalne. Pakery są natomiast uproszczoną wersją protektorów, których głównym zadaniem jest jedynie zmniejszenie wielkości pliku wykonywalnego poprzez kompresję, a nie zabezpieczanie go. Przykładem packera może być UPX.

Pakery i protektory są potwierdzeniem przysłowia „każdy miecz ma dwa końce", z uwagi na fakt ich użycia za równo do chronienia oprogramowania komercyjnego, jak i złośliwego.

5 SkySong

5.1 Opis systemu

SkySong jest próbą zaprojektowania oraz zaimplementowania przenośnego, elastycznego, systemu wspomagającego analizę wsteczną, opartego na całkowitej emulacji przestrzeni użytkownika.

5.1.1 Cel oraz założenia

Głównym celem stworzenia systemu SkySong była potrzeba wypełnienia luki w narzędziach umożliwiających przeprowadzenie analizy jednego procesu użytkownika z poziomu emulowanego procesora i środowiska.

Emulacja, mimo iż wolniejsza od wirtualizacji, a tym bardziej od rzeczywistego procesora, daje stanowczo największe możliwości tworzenia mechanizmów ułatwiających analizę wsteczną, ponieważ ingerencja, oraz monitorowanie procesu analizowanego odbywa się na poziomie wewnętrznych mechanizmów procesora, a nie na poziomie np. systemu operacyjnego, jak to ma miejsce w przypadku debuggerów działających na rzeczywistym procesorze. Dzięki takiemu podejściu obejście znacznej większości mechanizmów mających na celu utrudnienie bądź uniemożliwienie analizy wstecznej jest trywialnie proste. Emulacja, przy zastosowaniu dobrego mechanizmu rejestrowania, umożliwia również dwukierunkowość wykonania, czyli możliwość cofania się w kodzie do dowolnego momentu wykonania.

Podsumowując, za wyborem emulacji, jako bazy systemu, przemawiało:

- Monitorowanie na poziomie wewnętrznych mechanizmów procesora oraz pamięci, co daje na przykład możliwość stworzenia mechanizmu niewykrywalnych breakpointów.
- Możliwość ingerencji w wewnętrzne mechanizmy procesora.
- Niewrażliwość na metody wykrywające debuggery oraz wirtualizację.

- Pełna kontrola nad środowiskiem wykonania.

Wadami emulacji, które jednak w tym wypadku nie uniemożliwiają jej wykorzystania, ale o których należy wspomnieć, i które były rozpatrywane, są:

- Duża różnica (spowolnienie) w prędkości działania względem prawdziwego systemu.
- Wielkość implementacji (możliwość popełnienia wielu błędów które w późniejszej fazie życia projektu, umożliwiły by wykrycie emulacji).

SkySong, wykorzystując w tym celu zalety emulacji, miał spełniać następujące założenia:

- Emulować procesor x86 pracujący w ring 3 trybu chronionego.
- Umożliwiać dwukierunkowe poruszanie się po kodzie (step next, step prev).
- Umożliwiać cofnięcie wykonania do dowolnego miejsca.
- Emulować środowisko Win32.
- Umożliwiać załadowanie pliku PE.
- Umożliwiać pracę na całym programie, jak i pojedynczej, wyjętej z kontekstu, funkcji.
- Umożliwiać dowolne skonfigurowanie środowiska działania programu.
- Być niewykrywalnym przez istniejące metody.
- Umożliwiać budowanie narzędzi opartych o SkySong, poprzez dostarczenie wygodnego API.

Należy zauważyć że w przypadku zastosowania mechanizmu rejestrowania wykonania, który umożliwiał by przejście, w dowolnym momencie, do dowolnego momentu wykonania programu, następuje odejście od konwencji breakpointów. Breakpointy używane są aby spowodować zatrzymanie programu,

celem analizy jego stanu w danym momencie wykonania. Jednak jeśli system udostępnia możliwość podgląd stanu w dowolnym momencie wykonania program, w miejsce mechanizmu breakpointów, potrzebny jest mechanizm wyszukujący momenty wykonania które spełniają dany warunek. Jest to podejście podobne do mechanizmu breakpointów, jednak dużo bardziej elastyczne i oferujące większe możliwości, oraz znaczny wzrost szybkości wyszukiwania interesujących momentów w kodzie.

5.1.2 Przypadki użycia

Przykładowe zastosowania funkcjonalności systemu opartego o emulator z mechanizmem rejestrowania przebiegu mogą być następujące:

- Stworzenie automatycznego systemu zrzucającego obraz procesu do pliku wykonywalnego, łącznie z odtworzeniem IAT. Taki system mógłby zostać użyty na przykład do automatycznego rozpakowywania kodu złośliwego zabezpieczonego przed deasemblacją za pomocą szyfrowania części bądź całości kodu (np. za pomocą protektora lub pakera). Przewaga systemu opartego o emulator w tym wypadku wynikałaby z problemu odbudowy IAT. Na systemie emulowanym można śledzić w pamięci w łatwy sposób funkcje importowane oraz miejsce zapisu adresów do funkcji importowanych.

- Stworzenie automatycznego systemu monitorującego proces w celu wykrycia wystąpienia błędów typu buffer overflow. Emulator mógł by zostać zaprogramowany aby śledzić próby modyfikacji adresu powrotu lub adresu starej ramki stosu, oraz wskazywać miejsce i warunki środowiskowe w momencie wystąpienia modyfikacji. System ten mógł by zostać rozszerzony do automatycznego systemu testującego, np. stosującego fuzzing (technika testowania przez podawanie losowo zmodyfikowanych danych).

- Stworzenie interaktywnego debuggera. Widocznymi korzyściami z proponowanego podejścia jest możliwość tworzenia niewykrywalnych break-

pointów, cofanie się do dowolnego momentu wykonania bez potrzeby restartowania procesu, oraz bezpieczeństwo przy analizie kodu złośliwego. Dodatkową zaletą debuggera mógł by być system odpowiadający na pytanie „jak powstała dana wartość", który korzystając z rejestru wykonania mógł by (analizując wstecz) stwierdzić które konkretnie instrukcje w kodzie, oraz jakie wartości, przyczyniły się do uzyskania takiej a nie innej wartości w danej zmiennej.

- Stworzenie automatycznego systemu wykonującego analizę behawioralną kodu złośliwego. Emulacja pojedynczego procesu jest szybsza niż wirtualizacja całego systemu, a jednocześnie zapewnia bezpieczeństwo na równie dobrym poziomie. Umożliwia ona również dokładne monitorowanie odwołań do systemu, łącznie z odwołaniami do bibliotek DLL, bez ingerencji w proces monitorowany.

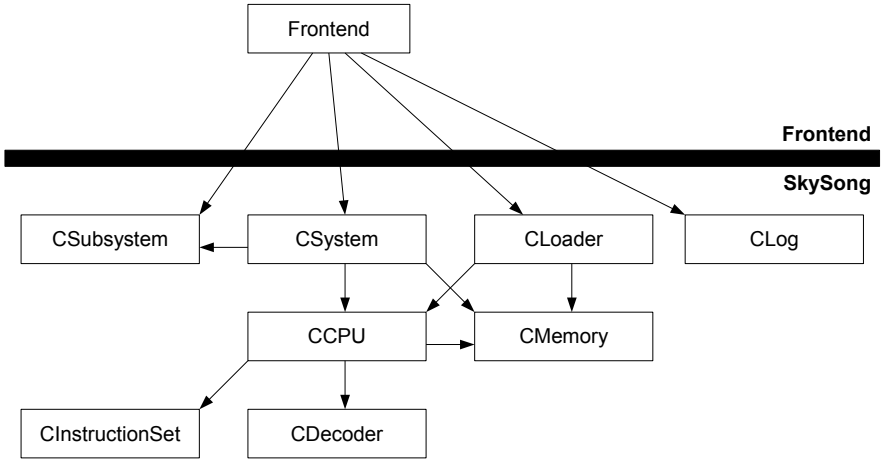
- Inne.

5.2 Architektura

Budowa SkySong opiera się o zestaw modułów, między którymi zachodzą pewne interakcje. Poszczególne moduły udostępniają zestaw funkcji, umożliwiających sterowanie oraz interakcje z modułem, ale jednocześnie zachowują autonomię uniemożliwiając ingerowanie w wewnętrzne mechanizmy modułów. Moduły w przypadku SkySong, z uwagi na wybór języka obiektowego, równoznaczne są klasom. W systemie SkySong wyróżnić można następujące moduły:

- CSystem – Moduł odpowiedzialny za koordynacj działań pozostałych modułów. Z logicznego punktu widzenia, CSystem reprezentuje system komputerowy, który zarządza jednym lub więcej CPU, pamięcią oraz subsystemem.

- CMemory – Moduł odpowiedzialny za obsługę pamięci, obecnie symuluje pamięć wirtualną na poziomie stronicowania. Jego głównym celem



Rysunek 2: Architektura SkySong

jest dostarczenie mechanizmów alokacji pamięci, dealokacji, odczytu pamięci, zapisu do pamięci oraz kontroli praw pamięci.

- CCPU – Moduł emulatora procesora, odpowiedzialny za dekodowanie oraz wykonanie instrukcji. Moduł ten emuluje procesor x86 działający w trybie chronionym ring 3.

- CDecoder – Moduł ściśle związany z modułem CCPU, służący do dekodowania instrukcji z postaci binarnej, do postaci zrozumiałej dla emulatora.

- CInstructionSet – Klasa zawierająca implementacji poszczególnych opkodów procesora, ściśle związana z CDecoder oraz CCPU.

- CLog – Moduł rejestrujący poszczególne mikrooperacje wykonywane na pamięci oraz rejestrach CPU przez opkody, subsystem oraz użytkownika (tj. zewnętrzne ingerencje w system), takie jak odczyt, zapis lub zmiana (czyli najpierw odczyta a potem zapis).

- CSubsystem – Grupa modułów odpowiedzialnych za emulacje poszczególnych subsystemów, np. subsystemu testowego lub subsystemu Win32. Gdy emulowany kod wywoła przerwanie, wyjątek lub dojdzie do zaznaczonego miejsca w pamięci CSystem przekazuje działanie odpowiedniemu modułowi implementującemu CSubsystem.

- CLoader – Grupa modułów odpowiedzialnych za przeniesienie pliku wykonywalnego do pamięci, oraz inicjację środowiska.

Z punktu widzenia programisty tworzącego oprogramowanie oparte o system SkySong (frontend), najważniejszym modułem jest CSystem, którego instancja koordynuje działanie rejestratora, procesora, pamięci, oraz subsystemu. Za równo instancje pamięci CMemory, jedną lub więcej instancji procesora CCPU, oraz opcjonalnie instancje subsystemu CSubsystem oraz instancje rejestratora CLog należy „podpiąć" pod główną instancje CSystem. Przykładowy cykl życia programu emulującego wykonanie procesu opartego o subsystem testowy wygląda następująco:

1. Tworzona jest instancja CSystem (dalej zwana Sys).

2. Tworzona jest instancja CMemory (dalej zwana Mem).

3. Tworzona jest instancja CCPU (dalej zwana Cpu).

4. Tworzona jest instancja CLoaderBin, odpowiedzialna za inicjację systemu na potrzeby programu wczytanego z beznagłówkowego pliku zawierającego kod programu (dalej zwana LoadBin).

5. Tworzona jest instancja CSubTest, odpowiedzialna za obsługę subsystemu testowego (dalej zwana Subsys).

6. Do systemu Sys dodawany jest procesor Cpu, pamięć Mem oraz subsystem Subsys.

7. LoadBin konfigurowany jest do współpracy z systemem Sys.

8. Ustalane jest miejsce w pamięci w które ma zostać załadowany program.

9. LoadBin ładuje program, oraz ustawia środowisko wykonania.

10. W systemie Sys procesor o identyfikatorze 0 jest przygotowywany do uruchomienia.

11. Aż do zakończenia programu, bądź wystąpienia nieobsługiwanego wyjątku, Sys realizuje program za pośrednictwem procesora Cpu.

Przykładowa implementacja powyższego w języku C++ wygląda następująco:

```
#include ,,SkySong.h''

int
main(void)
{
  CSystem    Sys;
  CCPU       Cpu;
  CMemory    Mem;
  CLoaderBin LoadBin;
  CSubTest   Subsys;

  Sys.AddCPU(&Cpu);
  Sys.SetMemory(&Mem);
  Sys.SetSubsystem(&Subsys);

  LoadBin.SetSystem(&Sys);
  LoadBin.SetLoadOffset(0x00400000);
  if(!LoadBin.Load("test.bin")) return 1;

  Sys.SetCurrentCPUId(0);
```

```
  while(1)
  {
    Sys.RunCurrentCPU(100);
    if(Cpu.IsExceptionIssued() || Cpu.IsInterruptIssued())
      return 2;

    if(Subsys.IsProcessTerminated())
      break;
  }

  return 0;
}
```

Frontend systemu SkySong może odpytać dowolną instancje o stan procesu zależny od danej instancji. Przykładowo instancja CCPU może zostać odpytana o stan wyjątków, przerwań, oraz wartości poszczególnych rejestrów procesora, a instancja CSubsystem o używane przez emulowany program odwołania systemowe.

Poniżej znajduje się szczegółowy opis poszczególnych modułów systemu SkySong.

5.2.1 Moduł CSystem

Moduł CSystem jest odpowiedzialny za koordynację działania wszystkich pozostałych modułów. Zawiera on wewnętrzną listę procesorów, oraz odnośniki do modułu pamięci, opcjonalnego modułu rejestrowania oraz opcjonalnego modułu subsystemu.

Implementacja modułu zawiera się w następujących plikach:

- SkySong/system/CSystem.h

- SkySong/system/CSystem.cpp

Moduł CSystem udostępnia następujący interfejs:

- SetSubsystem – Ustaw instancje subsystem.
- GetSubsystem - Pobierz obecną instancje subsystem.
- SetMemory – Ustaw instancje modułu pamięci.
- GetMemory – Pobierz obecną instancje moduły pamięci.
- AddCPU – Dodaj nową instancje procesora.
- GetCPUCount – Pobierz ilość procesorów.
- GetCPU – Pobierz instancje procesora wg. numeru.
- GetCurrentCPUId – Pobierz numer procesora który obecnie wykonuje kod.
- GetCurrentCPU – Pobierz instancje procesora który obecnie wykonuje kod.
- SetCurrentCPUId – Zmień procesor który obecnie wykonuje kod.
- SetLog – Ustaw instancje modułu rejestrowania.
- GetLog – Pobierz obecną instancje modułu rejestrowania.
- RunCurrentCPU – Wykonaj określona ilość instrukcji na obecnie wybranym procesorze.

CSystem umożliwia „podpięcie” kilku jednostek wykonywujących (procesorów) do jednego systemu, dzięki czemu możliwe jest bardzo proste symulowanie mechanizmu wątków, bez implementacji emulacji mechanizmu podmiany kontekstu wykonywania.

Frontend oparty o SkySong powinien stworzyć przynajmniej jedną instancje CSystem do prawidłowego działania. Stworzenie większej ilości instancji CSystem może zostać użyte do stworzenia symulacji sieci komputerowej, lub systemu komputerowego z większą niż jeden liczbą procesów.

5.2.2 Moduł CLog

Instancja modułu CLog rejestruje wszystkie otrzymywane od innych modułów, przypisując je do konkretnego momentu w czasie. W rejestrze zapisywane są informacje o trzech rodzajach akcji wykonywanych na danych:

- odczyt
- zapis
- zmiana – najpierw odczyt, następnie zapis

Od modułu CCPU rejestr otrzymuje informacje o akcjach wykonywanych na rozmaitych rejestrach procesora. Od modułu pamięci rejestr otrzymuje informacje o akcjach wykonywanych na komórkach pamięci. Rejestr CSystem jest odpowiedzialny za informowanie rejestru o tym kto wykonuje akcje (użytkownik, cpu (prefetch), subsystem, kod programu, loader), na jakim CPU, oraz w jakim momencie czasu.

Moduł CLog zaimplementowany jest w następujących plikach:

- SkySong/system/log/CLog.h
- SkySong/system/log/CLog.cpp

Rejestr formalnie jest podzielony na ramki czasu (ang. Time Frame). Każda ramka zawiera informacje nagłówkową, oraz zestaw akcji wykonanych w danej ramce czasu. Informacja nagłówkowa zawiera następujące informacje:

- Numer procesora

- Numer ramki czasu (liczba naturalna)
- Adres instrukcji wykonywanej przez procesor

Każda akcja która odbyła się w danej ramce czasu opisywana jest następującymi informacjami:

- Kto wykonał akcje (użytkownik, cpu, subsystem, kod programu, loader)
- Jaka to była akcja (zapis, odczyt, zmiana)
- Szczegółowa informacja o obiekcie którego dotyczy akcja (np. rejestr EAX) wraz z wielkością obiektu
- Jaka wartość była w komórce przed zapisem/zmianą
- Jaka wartość znajdzie się w komórce po zapisie/zmianie (pole pomijane w wypadku odczytu)

Przykładowo rozważmy krótki program:

```
00001000 mov eax, 10
00001007 mov edx, 10
0000100D add eax, edx
00001010 add [0x2000], eax
```

Wygeneruje następującą zawartość rejestru:

```
-- RAMKA: Moment=0, EIP=0x00001000, CPU=0
   ZAPIS : Kto=CPU, Co=EAX, Przed=0, Po=10
-- RAMKA: Moment=1, EIP=0x00001007, CPU=0
   ZAPIS : Kto=CPU, Co=EDX, Przed=0, Po=10
-- RAMKA: Moment=2, EIP=0x0000100D, CPU=0
   ODCZYT: Kto=CPU, Co=EDX, Przed=10
   ODCZYT: Kto=CPU, Co=EAX, Przed=10
   ZAPIS : Kto=CPU, Co=EAX, Przed=10, Po=20
-- RAMKA: Moment=3, EIP=0x00001010, CPU=0
```

```
   ODCZYT: Kto=CPU, Co=EAX, Przed=20
   ODCZYT: Kto=CPU, Co=MEM(0x2000), Przed=0
   ZAPIS : Kto=CPU, Co=MEM(0x2000), Przed=0, Po=20
```

Mechanizm defragmentacji, działający na zakończonych ramkach, szuka wystąpienia odczytu oraz zapisu do tego samego miejsca w danej klatce. W przypadku znalezienia oba elementy zostają połączone w „zmianę". Powyższy przykład po zastosowaniu mechanizmu defragmentacji wyglądał by następująco:

```
-- RAMKA: Moment=0, EIP=0x00001000, CPU=0
   ZAPIS : Kto=CPU, Co=EAX, Przed=0, Po=10
-- RAMKA: Moment=1, EIP=0x00001007, CPU=0
   ZAPIS : Kto=CPU, Co=EDX, Przed=0, Po=10
-- RAMKA: Moment=2, EIP=0x0000100D, CPU=0
   ODCZYT: Kto=CPU, Co=EDX, Przed=10
   ZMIANA: Kto=CPU, Co=EAX, Przed=10, Po=20
-- RAMKA: Moment=3, EIP=0x00001010, CPU=0
   ODCZYT: Kto=CPU, Co=EAX, Przed=20
   ZMIANA: Kto=CPU, Co=MEM(0x2000), Przed=0, Po=20
```

Należy zauważyć iż pojedyncza ramka czasu zajmuje średnio 56 bajtów pamięci. Przy długich programach wykonujących powyżej kilku milionów operacji wielkość rejestru może przekroczyć granicę 100MB, a w przypadku ogromnych długo działających programów rejestr może zająć nawet kilka GB pamięci.

Rejestr dodatkowo umożliwia cofnięcie stanu procesu do określonego momentu. Realizowane jest to krok po kroku, względem ramek czasu. Każda poprzednia ramka czasu jest nanoszona (pola „Przed") na stan pamięci oraz CPU, przez co odzyskiwany jest dokładnie taki stan jaki był w danym momencie wykonania programu. Należy zauważyć iż czas cofania stanu CPU i pamięci w takim przypadku jest liniowy, co dla dużego kodu może zająć do kilkunastu sekund. Pewnym rozwiązaniem jest zastosowanie ramek kluczowych, które będą występować co określoną ilość ramek, i będą zawierać

całą zawartość pamięci oraz stan wszystkich rejestrów CPU. Umożliwiło by to cofanie do najbliższej danemu momentowi ramki kluczowej, a następnie cofanie stanu procesora ramka po ramce. Wadą tego rozwiązania była by ilość zużywanej pamięci przez ramki kluczowe.

5.2.3 Moduł CMemory

Moduł CMemory jest odpowiedzialny za emulacje mechanizmu pamięci RAM. Został on stworzony przede wszystkim z myślą o architekturze x86, w związku z czym podział pamięci opiera się na stronach o wielkości 4MB lub 4KB. Z modułem pamięci związane są przede wszystkim dwa terminy, alokacja oraz dealokacja. Alokacja jest to zażądanie od modułu pamięci jednej (lub więcej) wolnej strony pamięci, o określonych prawach dostępu, znajdujących się pod określonym adresem. W przypadku alokacji większej ilości stron, kolejne strony mają adresy większe o wielkość strony od adresu poprzedniej strony. Nowo zaalokowana strona wypełniona jest zerami. Dealokacja odnosi się do zwolnienia wcześniej zaalokowanej strony. Wszelkie dane przechowywane na stronie są w takim wypadku tracone.

Adres strony musi być wyrównany do wielkości strony, czyli np. w wypadku stron o wielkości 4KB (4096B, 0x1000B) musi być podzielny przez 4096 (0x1000). Każda strona jest opisana dodatkowo prawami „do odczytu" (read), „do zapisu" (write) oraz „do wykonania" (execute). Kontrola praw odczytu oraz zapisu odbywa się na poziomie modułu CMem, ale kontrola prawa „do wykonania" musi zostać zrealizowana w module wykonującym (CCPU).

Implementacja modułu CMemory zawiera się w następujących plikach:

- SkySong/system/memory/CMemory.h
- SkySong/system/memory/CMemory.cpp

Moduł CMemory udostępnia następujący interfejs:

- Alloc – Alokuje jedną lub więcej stron o określonych prawach, na określonych adresach.
- Free – Dealokuje jedną lub więcej stron o podanych adresach.
- IsAllocated – Testuje czy strona o podanym adresie jest zaalokowana.
- SetFlags – Ustawia prawa dla jednej lub więcej kolejnych stron.
- GetFlags – Pobiera prawa dla strony o podanym adresie.
- GetPageSize – Pobiera wielkość strony o określonym adresie.
- SetBytes – Ustawia dane fragmentu pamięci.
- GetBytes – Pobiera dane fragmentu pamięci.
- SetBYTE, SetWORD, SetDWORD – Ustawia bajt, słowo (dwa bajty) lub podwójne słowo (cztery bajty) danych w podanym fragmencie pamięci.
- GetBYTE, GetWORD, GetDWORD – Pobiera bajt, słowo lub podwójne słowo danych z określonego fragmentu pamięci.
- GetLastError – Zwraca informacje o ostatnim błędzie.

Wewnętrzna budowa CMemory opiera się o hash-listę przechowującą informacje o zaalokowanych stronach pamięci oraz adresy danych stron. Wszelkie operacje na pamięci są zgłaszane instancji modułu CLog, celem zarejestrowania zmian dokonywanych w pamięci.

5.2.4 Moduł CCPU

Jednostką wykonywującą kod emulowanego programu jest instancja modułu CCPU. Moduł CCPU odpowiedzialny jest za rozkodowanie pojedynczej instrukcji kodu maszynowego, oraz jej wykonanie. Obecnie moduł skupia się na instrukcjach procesora x86 z rozszerzeniami FPU oraz MMX^{TM}, jednak trzeba zaznaczyć iż w chwili pisania tej pracy moduł CCPU jest wysoce

niekompletny. Moduł CCPU zawiera w sobie kilka dodatkowych modułów, takich jak CDecoder który jest odpowiedzialny za dekodowanie instrukcji z postaci binarnej do postaci zrozumiałej przez CCPU (sprowadza się to do wypełnienia odpowiedniej struktury informacjami o instrukcji oraz jej argumentach), oraz CInstructionSet który zawiera zestaw funkcji emulujących poszczególne instrukcje.

Moduł CDecoder zaimplementowany jest w następujących plikach:

- SkySong/system/cpu/CDecoder.h
- SkySong/system/cpu/CDecoder.cpp
- SkySong/system/cpu/CDecoderData.cpp
- SkySong/system/cpu/COpcodeInfo.h

Plik CDecoderData.cpp jest generowany automatycznie za pomocą skryptów, i zawiera tabele oraz mapy umożliwiające szybkie dekodowanie instrukcji. W skład pliku wchodzą następujące struktury:

- Tablica OpcodeList – zawierająca kompletną listę opkodów wraz z ich reprezentacją mnemoniczną, informacją o budowie oraz o przyjmowanych argumentach.
- Tablica GroupList – zawierająca informacje o grupach opkodów (tj opkodach które posiadają pole ModR/M, a w nim w bitach Reg (3-5) zamiast informacji o rejestrze, jest informacja o numerze opkodu w grupie opkodów).
- Tablica OpcodeArray – będąca jedynie kontenerem danych o opkodach oraz grupach opkodów dla mapy OpcodeMap.
- Mapa OpcodeMap – mapa wyszukująca po opkodzie zapisanym w podwójnym słowie strukturę opisującą opkod bądź grupę opkodów.

Aby przetłumaczyć instrukcję z kodu maszynowego na formę zrozumiałą dla CCPU, należy zdekodować wstępnie instrukcje, wydobyć z niej opkod, zapisać go w postaci podwójnego słowa, a następnie użyć OpcodeMap aby znaleźć grupę do której należy opkod bądź od razu samą strukturę opisującą opkod, jeśli opkod nie należy do żadnej grupy. Jeżeli opkod należy do grupy, należy zdekodować ModR/M, oraz w tablicy grup wyszukać informacje o opkodzie, posługując się indeksem uzyskanym z części Reg pola ModR/M. Informacje o opkodzie można użyć do całkowitego zdekodowania instrukcji, oraz do wypełnienia struktury zawierającej informacje o konkretnej instrukcji. Struktura ta posłuży jako źródło informacji dla funkcji wykonującej daną instrukcję.

Moduł CInstructionSet jest zaimplementowany w następujących plikach:

- SkySong/system/cpu/CInstruction.h
- SkySong/system/cpu/CInstruction.cpp
- SkySong/system/cpu/InstructionSet/*

Jest to obecnie największy pod względem objętościowym moduł. Moduł zawiera przede wszystkim szereg statycznych funkcji implementujących pojedyncze instrukcje procesora. Instrukcje pogrupowane względem nazwy są przechowywane w katalogu InstructionSet (obecnie jest to 309 instrukcji, co przekłada się na 309 plików źródłowych).

Głównymi plikami zawierającymi implementacje modułu CCPU są:

- SkySong/system/cpu/CCPU.h
- SkySong/system/cpu/CCPU.cpp

Instancja głównego modułu CCPU zawiera przede wszystkim rejestry procesora, koprocesora i rozszerzeń, oraz symulowaną tablicę GDT. CCPU udostępnia dwa oddzielne interfejsy. Pierwszy jest zależny od typu procesora, i

służy do operowania na rejestrach oraz tablicy GDT. Drugi jest wspólny dla każdego typu, i służy do uruchamiania jednostki dekodującej oraz wykonywującej. W drugim interfejsie znajdują się następujące funkcje:

- PrefetchInstruction – Pobiera instrukcje z pamięci i ją dekoduje.
- ExecuteInstruction – Wykonuje uprzednio pobraną i zdekodowaną instrukcje.

Dodatkowo CCPU udostępnia interfejs do operacji na przerwaniach oraz wyjątkach (jest on rozdzielony).

Każda instancja CSystem powinna zawierać przynajmniej jedną instancje CCPU.

5.2.5 Moduły subsystemów

Subsystem jest częścią emulatora odpowiedzialną za emulowanie systemowego API. Subsystem opiera się o funkcje zwrotne (callback), które są wywoływane w momencie gdy kod emulowany wywoła przerwanie, sysenter, syscall, wystąpi wyjątek, lub nastąpi próba wykonania kodu w miejscu wcześniej oznaczonym. Dzięki temu subsystem ma szansę na ingerencje w stan rejestrów wykonywanego procesu, jego pamięć lub jego system komputerowy (w celu np. dodania nowego procesora).

Obecnie od subsystemu wymagana jest implementacja następującego interfejsu:

- InitSubsystem – Uruchamiana w momencie stworzenia nowego procesu emulowanego.
- CloseSubsystem – Uruchamiana w momencie zakończenia działania procesu emulowanego.
- HandleException – Wywoływana w momencie wystąpienia wyjątku w procesie emulowanym.

- HandleInterrupt – Wywoływana w momencie wywołania przerwania w procesie emulowanym.
- HandleSysenter – Wywoływana w momencie wywołania instrukcji SYSENTER przez proces emulowany.

Przykładowym subsystemem jest subsystem testowy SubTest, który implementuje jedynie funkcje HandleInterrupt, dodając obsługę przerwania INT 99h, w którym umieszczona przykładowa funkcja systemowa umożliwiająca wypisywanie na konsolę tekstu zapisanego jako ASCIIZ (ang. ASCII Zero-terminated).

Subsystemy mieszczą się obecnie w katalogu SkySong/subsystem.

5.2.6 Moduły loaderów

Celem Loadera jest skonfigurowanie środowiska procesu emulowanego do uruchomienia, oraz wczytanie kodu oraz danych programu z nośnika (pliku). Od Loadera wymagana jest implementacja interfejsu reprezentowanego przez pojedynczą funkcję Load, która odpowiedzialna jest za konfigurację środowiska oraz wczytanie kodu z pliku. Loadery mieszczą się w katalogu SkySong/loader.

5.2.7 Dodatkowe programy

W katalogu _tools mieszczą się dodatkowe programy oraz pliki napisane na potrzebę projektu SkySong. Są to: oplist.txt – lista opkodów procesora wraz z rozszerzeniami FPU oraz MMX™. transform*.cpp – programy wczytujące oplist.txt oraz generujące szablony dla InstructionSet modułu CCPU, oraz tabele dekodera zawarte w pliku CDecoderData.cpp.

5.3 Wady obecnej wersji

Jak każdy produkt, SkySong ma również, oprócz zalet, wady, które głównie wynikają z jego niekompletności. Głównymi wadami w obecnej wersji są:

1. Niekompletność systemu. Brakuje sprawdzonej implementacji wszystkich instrukcji procesora, czy obsługi rozszerzeń SSE, SSE2, SSE3, SSSE3, Enhanced 3DNow!™, i 3DNow!™Professional. Subsystem Win32 jest również wysoce niekompletny.

2. Łatwość detekcji emulacji przez kod emulowany (związane jest to z poprzednim punktem).

3. Brak mechanizmu defragmentacji w module CLog, który umiał by łączyć kilka momentów wykonania w jeden. Taki mechanizm mógł by zostać użyty przykładowo do zmniejszania miejsca wymaganego do zapisania rejestru przebiegu wykonania pętli.

4. Architektura, mimo iż zaprojektowana elastycznie, powinna zostać jeszcze bardziej uelastyczniona, szczególnie aby umożliwić dopisanie emulacji innych rodzajów procesorów, z innych architektur, takich jak ARM™, MIPS®, Sparc®, SH3, czy innych.

5. Brak funkcji umożliwiającej usuwane instancji procesora z systemu, oraz opcji łączenia licznika instrukcji dla kilku instancji procesora.

6. SkySong jako system nie potrafi, oraz prawdopodobnie nigdy nie będzie potrafił, emulować systemu na poziomie ring0.

5.4 Plan rozwoju

Plan dalszego rozwoju systemu SkySong, oraz aplikacji opartych o SkySong, jest następujący:

1. Udostępnienie systemu SkySong na jednej z licencji OpenSource, umieszczenie repozytorium w ogólnodostępnym miejscu, oraz zaproszenie programistów, oraz reverse engineerów, do udziału w projekcie.

2. Drobna przebudowa architektury mająca na celu udostępnienie mechanizmów które umożliwią podpięcie emulatorów innych architektur, takich jak ARM™, Sparc™, x64, itp.

3. Zaimplementowanie oraz bardzo dokładne przetestowanie wszystkich brakujących instrukcji procesora x86 wraz ze wszystkimi dostępnymi rozszerzeniami (FPU, MMX™, SSE, SSE2, SSE3, SSSE3, Enhanced 3DNow!™, 3DNow!™Professional) oraz rozszerzeniami które zostaną udostępnione w niedalekiej przyszłości (SSE4).

4. Zaimplementowanie subsystemu Win32 wraz z bardzo dokładnymi testami kompatybilności. Subsystem Win32 docelowo powinien umożliwiać emulacje Win32 na kilku poziomach: poziomie sysenter, na którym emulowane są tylko syscalle; poziomie głównych systemowych bibliotek DLL (Kernel32.dll, User32.dll, Gdi32.dll, Advapi32.dll); oraz umożliwiać emulację wybranej funkcjonalności, a przekazywanie do systemu innych wywołań (np. operacji na plikach czy na rejestrze), wraz z umożliwieniem wirtualizacji tych ostatnich.

5. Stworzenie frontendu „SkySong Generic Unpacker" do automatyzacji rozpakowywania plików PE zabezpieczonych protektorami bądź pakerami.

6. Stworzenie frontendu „SkySong Debugger" umożliwiającego interaktywną pracę ze SkySongiem.

7. Testowanie oraz ulepszanie SkySong.

Powyższy plan przewidziany jest na kilka lat.

6 Zakończenie

Analiza wsteczna jest działem informatyki który się stale rozwija, powstają nowe narzędzia automatyzujące proces analizy, ułatwiające „rozpakowywanie" zaszyfrowanego kodu, czy monitorujące zachowania procesu. Wraz z rozwojem analizy wstecznej, rozwijają się również metody mające uchronić kod przed specjalistami od RCE. Nowe zabezpieczenia oparte są o rozwiązania sprzętowe (tokeny, dongle), maszyny wirtualne, sterowniki oraz wykorzystują rozmaite kruczki systemowe i sprzętowe. Należy jednak zaznaczyć iż do tej pory nie udało się znaleźć skutecznego mechanizmu anty-RE. Z jednej strony to źle, szczególnie dla korporacji które przez piratów, a wziązku z tym i software crackerów, tracą potencjalne zyski, ale z drugiej strony, dzięki temu wirusy komputerowe są szybko rozbrajane i deaktywowane. Póki co RE oraz anty-RE utrzymują się wzajemnie w stanie równowagi. Przyszłość pokaże czy szalach przechyli się na którąkolwiek ze stron.

6.1 Podziękowania

Od autora: Chciał bym w tym miejscu podziękować kilku osobom, niektórym bardzo bliskim, innym znanym mi tylko z nicka na IRC'u, za konstruktywne rozmowy oraz czasami „wskazanie" najlepszej drogi (w kolejności losowej):

- Moim rodzicom, bratu
- Mojemu promotorowi, dr Wiesławowi Cupale
- Team Vexillium (za wspólne projekty)
- fr3m3n'owi, j00ru, ReWolf'owi, 'ronin'owi, Nekrataal'owi oraz Unavowed'owi
- Przemysławowi Godkowi oraz Adamowi Stojanowskiemu

- Polskiej scenie RCE

Literatura

[1] Gil „Arkon" Dabah, *80x86 Machine Code*, http://www.ragestorm.net/distorm/vol1.html, **2007**

[2] Intel®Corporation, *I64 And IA32 Software Developer Manual, Volume 1*, Intel®Corporation, **2006**

[3] Intel®Corporation, *I64 And IA32 Software Developer Manual, Volume 2a*, Intel®Corporation, **2006**

[4] Intel®Corporation, *I64 And IA32 Software Developer Manual, Volume 2b*, Intel®Corporation, **2006**

[5] Intel®Corporation, *I64 And IA32 Software Developer Manual, Volume 3a*, Intel®Corporation, **2006**

[6] Intel®Corporation, *I64 And IA32 Software Developer Manual, Volume 3b*, Intel®Corporation, **2006**

[7] Elliot Chikofsky, *Przedmowa do Reversing: Secrets of Reverse Engineering*, Wiley Publishing, Inc., **2005**

[8] Eldad Eilam, *Reversing: Secrets of Reverse Engineering*, Wiley Publishing, Inc., **2005**

[9] Mark K. Russinovich, David A. Solomon, *Microsoft® Windows® Internals, Fourth Edition: Microsoft Windows Server™ 2003, Windows XP, and Windows 2000*, Microsoft Press, **2004**

[10] Kris Kaspersky, *Hacker Disassembling Uncovered*, A-LIST Publishing, **2003**

[11] Tom Liston, Ed Skoudis, *On the Cutting Edge: Thwarting Virtual Machine Detection*, http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf, **2007**

[12] Peter Ferrie, *Attacks on Virtual Machine Emulators*, http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf, **2006**

[13] Cristina Cifuentes, *Reverse Compilation Techniques*, Queensland University of Technology, **1994**

[14] Greg Hoglund, James Butler, *Rootkits: Subverting the Windows Kernel*, Addison Wesley Professional, **2005**

[15] Microsoft, *Microsoft Portable Executable and Common Object File Format Specification*, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx, **2006**

[16] Joanna Rutkowska, *Red Pill... or how to detect VMM using (almost) one CPU instruction*, http://invisiblethings.org/papers/redpill.html, **2004**

[17] Wikipedia, *Unreal mode*, http://en.wikipedia.org/wiki/Unreal_mode, **2007**

[18] Wikipedia, *Model-specific register*, http://en.wikipedia.org/wiki/Model-specific_register, **2007**

[19] Wikipedia, *SSE4*, http://en.wikipedia.org/wiki/SSE4, **2007**

[20] Wikipedia, *Microsoft® Windows®*, http://pl.wikipedia.org/wiki/Windows, **2007**

[21] Wikipedia, *Reverse engineering*, http://pl.wikipedia.org/wiki/Reverse_engineering, **2007**

[22] WIKIPEDIA, *Clean room design*, http://en.wikipedia.org/wiki/Cleanroom_design, **2007**

[23] WIKIPEDIA, *Emulator*, http://pl.wikipedia.org/wiki/Emulator, **2007**

[24] MSDN®, *Debugging a Running Process*, http://msdn2.microsoft.com/en-us/library/ms679301.aspx, **2007**

[25] MSDN®, *Debugging and Error Handling*, http://msdn2.microsoft.com/en-us/library/ms679300.aspx, **2007**