UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

MONITOROVANIE A RIADENIE APLIKÁCIÍ V SYSTÉME MS WINDOWS

2011

LUKÁŠ JUSKO

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Monitorovanie a riadenie aplikácií v systéme MS Windows

Bakalárska práca

Študijný program: Informatika
Študijný odbor: Informatika, 9.2.1.
Školiace pracovisko: Katedra informatiky
Školiteľ: Mgr. Peter Košinár

BRATISLAVA 2011

Lukáš Jusko
6ead7e17-7e27-4eb8-a75b-926f3317de6b

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Lukáš Jusko
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Monitorovanie a riadenie aplikácií v systéme MS Windows

Cieľ: Preskúmať možnosti monitorovania aktivít aplikácií v MS Windows a ich centralizovaného riadenia.

Vedúci: Mgr. Peter Košinár
Katedra: FMFI.KI - Katedra informatiky

Dátum zadania: 27.10.2010

Dátum schválenia: 27.10.2010

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
Vedúci

ČESTNÉ VYHLÁSENIE

Vyhlasujem, že som bakalársku prácu vypracoval samostatne a uviedol som všetku použitú literatúru.

……

vlastnoručný podpis študenta

POĎAKOVANIE

Chcel by som touto cestou vyjadriť veľké poďakovanie Mgr. Petrovi Košinárovi za jeho ochotu, ústretovosť, cenné pripomienky a rady pri písaní tejto bakalárskej práce, ktorá by bez jeho podpory vznikala len veľmi ťažko.

ABSTRAKT

Cieľom tejto bakalárskej práce je preskúmať možné riešenia prístupu ku kontrole a riadeniu aplikácie v operačnom systéme Windows.

V úvodných častiach sú vysvetlené základné poznatky, ktoré sú nevyhnutné pri skúmaní možných prístupov ku kontrole a riadeniu aplikácii v MS Windows. Cieľom nie je hľadať metódy na detekciu škodlivých aplikácií, ale nájsť riešenie na monitorovanie korektnej aplikácie, sledovanie jej činností. Nakoniec je zostrojená aplikácia, ktorá implementuje daný prístup a sleduje zvolené aplikácie.

Kľúčové slová: MS Windows, Aplikácia, Monitorovanie, Riadenie

ABSTRACT

The main aim of this bachelor's thesis is to explore possible approaches to application control and management in Windows operating system.

In the introductory sections there is explained knowledge necessary to examine possible approaches to the application control and management in MS Windows. The aim is not to look for methods to detect malicious applications, but to find a solution to manage correct application and monitor its activities. Finally, one of the approaches has been implemented in a form of a stand-alone application, allowing monitoring of other applications and their behavior.

Key words: MS Windows, Application, Monitoring, Management

OBSAH

Úvod ... 8

1 OS Windows a aplikácia ... 9
1. 1 OS Windows ... 9
1. 1. 1 Architektúra OS Windows ... 9
1. 1. 2 Windows API ... 10
1. 2 Aplikácia ... 11
2 PE formát ... 12
2. 1 Štruktúra PE formátu ... 12
2. 2 Import/Export oblasti ... 15
2. 2. 1 Import tabuľka ... 16
2. 2. 2 Export tabuľka ... 18
3 Vytvorenie procesu ... 19
3. 1 Funkcia CreateProcess ... 19
3. 2 Zavedenie DLL do pamäte ... 22
4 Analýza prístupu ku kontrole aplikácie ... 23
4. 1 Prehľad volania systémových funkcií ... 23
4. 2 Hlavné prístupy ku kontrole ... 24
4. 2. 1 Sledovanie aktivít v jadre systému ... 24
4. 2. 2 Úprava priamo na disku ... 25
4. 2. 3 Podvrhnutie falošnej knižnice ... 26
4. 3 Zhrnutie prístupov ... 27
5 Implementácia prístupu ... 28
5. 1 Implementačné detaily ... 28
5. 1. 1 Spustenie procesu do suspendovaného stavu ... 28
5. 1. 2 Spracovanie import tabuľky ... 29
5. 1. 3 Podhodenie falošnej knižnice ... 31
5. 1. 4 Zostrojenie falošnej knižnice ... 32
5. 2 Riadenie aplikácie ... 33

Záver ... 36
Literatúra ... 37
Prílohy ... 38

Úvod

Táto bakalárska práca má pomôcť programátorom a vývojárom pri ladení aplikácií v MS Windows. Aplikácia veľa vecí nevykonáva samotne, ale využíva služby operačného systému. Tieto volané služby väčšinou prijímajú parametre a vracajú návratovú hodnotu, alebo len vykonajú nejakú činnosť. Pri vývoji často krát dochádza k nežiaducemu správaniu aplikácie. Jednou z príčin je často krát nesprávne volanie systémovej služby, ktoré zapríčiní, že sa nevykoná, to čo sa od služby očakáva.

Prvá kapitola popisuje základnú architektúru operačného systému Windows, ako aj API rozhranie, ktoré umožňuje aplikáciám prístup k systémovým službám alebo operáciám. Taktiež sa popisuje aplikácia, ktorá je hlavným predmetom tejto práce.

V druhej kapitole je popis štruktúry PE formátu. PE formát majú EXE, DLL či SYS. Pre nás sú zaujímavé najmä EXE a DLL súbory, ktorých hlavné časti budú popísané v tejto kapitole, keďže bez nich je akýkoľvek ďalší postup hľadania prístupu ku kontrole aplikácií nemožný.

V tretej kapitole je popísaná funkcia CreateProcess(), ktorá je volaná pri vzniku nového procesu systému Windows. Vytvorenie procesu pozostáva z niekoľkých fáz, ktoré sú z nášho pohľadu pre nás veľmi dôležité. Takisto je popísané zavedenie DLL knižnice do pamäte, ktoré ma pre nás tiež veľký význam.

Štvrtá kapitola využíva poznatky z predošlých troch kapitol a na základe týchto poznatkov môžeme zanalyzovať možné prístupy ku kontrole a riadeniu aplikácie. Na základe analýzy výhod a nevýhod jednotlivých prístupov nájdeme pre nás najlepšiu variantu prístupu.

Piata kapitola popisuje proces implementácie zvoleného prístupu. Pri implementácii sa už naráža na špecifiká operačného systému a tie budú v tejto kapitole aj zaznamenané.

1 OS Windows a aplikácia

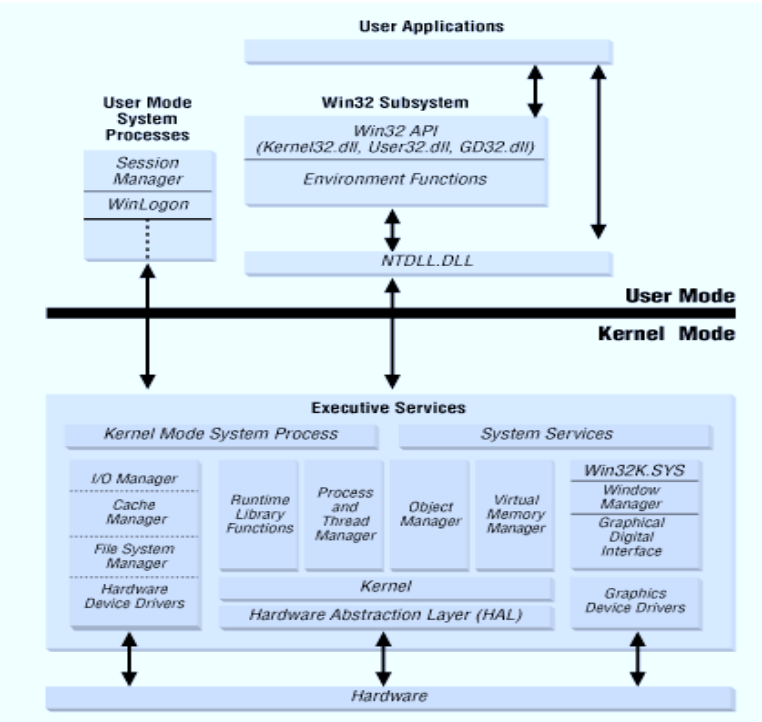
Táto kapitola sa zameriava na architektúru operačného systému Windows a API rozhranie, prostredníctvom ktorého môžu aplikácie využívať systémové služby a volania. Taktiež sa vysvetľuje čo je to vlastne aplikácia a ako môže využívať API rozhranie systému Windows. Presnejšie sa budeme zameriavať na 32-bitovú verziu operačného systému Windows.

1. 1 OS Windows

Microsoft Windows predstavuje sériu niekoľkých rodín operačných systémov od spoločnosti Microsoft. Pre nás bude zaujímavá predovšetkým rodina Windows NT, do ktorej patrí aj operačný systém Microsoft Windows XP, Microsoft Windows Vista, či najnovší operačný systém Microsoft Windows 7.

1. 1. 1 Architektúra OS Windows

Nasledovný obrázok popisuje základný prehľad architektúry OS Windows:



Obrázok bol prevzatý zo stránky [5].

Ako možno vidieť na obrázku OS Windows beží v dvoch režimoch:

- v užívateľskom režime beží aplikačný kód, tento režim je neprivilegovaný, má k dispozícií obmedzenú sadu rozhraní a obmedzený prístup k systémovým dátam
- v režime jadra beží privilegovaný kód (napr. systémové služby), ktorý má priamy prístup k systémovým dátam a hardwaru

Z obrázka je pre nás zaujímavý predovšetkým užívateľský režim, keďže práve v ňom bežia naše programy.

Systémové procesy hostia služby Windows. Napríklad plánovač úloh, Spooler, WinLogon (prihlásenie užívateľa do systému) .

Užívateľské aplikácie bežia v aplikačnom prostredí, v prípade potreby vykonať systémovú operáciu, majú k dispozícii Win32 subsystém alebo môžu priamo využiť knižnicu Ntdll.

Režim jadra, ako je možné vidieť z obrázka, ponúka systémové služby a systémové procesy v režime jadra. Okrem týchto služieb sa tu nachádzajú správca vlákien, správca V/V operácií, správca virtuálnej pamäte, správca súborov, správca medzipamäte. Na najnižšej úrovni sú ovládače na grafiku a hardware a abstraktná vrstva operujúca priamo nad hardwarom.

1. 1. 2 Windows API

Win32 subsystém, ktorý sa nachádza v užívateľskom režime medzi užívateľskou aplikáciou a knižnicou Ntdll obsahuje DLL knižnice, ktoré tvoria zdokumentované API (Application Programming Interface) rozhranie. Toto rozhranie umožňuje aplikáciám využívať systémové služby. DLL knižnica je binárna knižnica obsahujúca funkcie, ktoré exportuje, čiže ponúka aplikačnému prostrediu. Viac o DLL knižniciach sa nachádza v nasledujúcich kapitolách.

Zdokumentované API znamená, že k verejným funkciám, ktoré dané DLL knižnice exportujú, existuje popis vstupných parametrov, výstupných hodnôt podľa behu funkcie. Nezdokumentovaná časť knižníc nie je priamo určená aplikáciám, pretože Microsoft ich mení a neudáva, ako sa majú funkcie správne zavolať a aké majú byť

návratové hodnoty. Tie by podľa správneho systému mali volať len funkcie Windows DLL, ktoré majú poznať ich rozhranie a správanie.

Tento prístup z pohľadu vývoja pripomína návrhový vzor Fasáda, zdokumentovaná časť API tvorí fasádu a nezdokumentované API veľkú časť skrytú za touto fasádou. Takže často krát funkcia len zavolá ďalšiu funkciu a až tá zavolá systémovú službu. Podrobnejšie je o tom napísané v kapitole 5

1. 2 Aplikácia

Aplikácia je súbor programov, ktoré vykonávajú preddefinovanú postupnosť príkazov. Proces je jeden bežiaci program. Pre jednoduchosť budeme predpokladať, že aplikácia má práve jeden proces. Aplikácia by ako samotná aplikácia bez podpory operačného systému nič nezmohla. Preto je odkázaná na funkcie Windows API, ktoré jej umožňujú prístup k grafickému rozhraniu, prístup k prostriedkom, k Windows registrom, súborovému systému atď.

Tieto knižnice nie sú k aplikácii prilinkované na pevno. Ak vyjde novšia verzia DLL knižnice tak aplikácia s tým nemá žiaden problém, ak sa samozrejme nezmení rozhranie a správanie funkcií. To samozrejme je aj jedna z hlavných podstát DLL knižníc, aby pri zmene knižnice nebolo potrebne nanovo celú aplikáciu skompilovať.

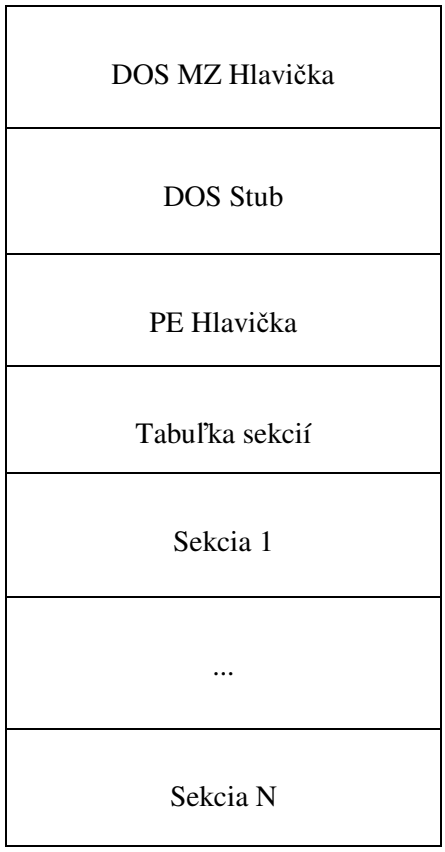
Takýmto spôsobom sa teda aplikácie môžu realizovať a využívať naplno celé rozhranie ponúkane systémom. To aj ukazuje potrebu práve sledovať túto komunikáciu medzi aplikáciu a DLL knižnicou, ktorú aplikácia využíva k volaniu konkrétnych funkcií. Na to, aby sme mohli sa do tejto komunikácie dostať a tak povediac ju uniesť, potrebujeme poznať štruktúru EXE súborov a DLL funkcií, ktoré túto komunikáciu tvoria.

2 PE formát

PE je skratka pre Portable Executable. Je to natívny formát súborov pre Win32. Tento formát majú EXE, DLL a SYS súbory. Jeho špecifikácia je odvodená od Unix Coff(common object file format). Význam časti portable (prenesiteľný) je, že tento formát je prenesiteľný naprieč celou platformou win32. Význam časti executable je, že tento formát je spustiteľný. EXE je priamo odvodený od tohto slova executable, ale pri DLL to až tak intuitívne nie je, že je to spustiteľný súbor. DLL knižnica je pri použití namapovaná do pamäte a tam dochádza k spusteniu jej kódu. Prienik do štruktúry PE formátu nám umožní veľmi cenný pohľad do štruktúry systému Windows.

2. 1 Štruktúra PE formátu

Nasledujúci obrázok popisuje hlavnú štruktúru PE formátu, ktorá sa skladá z jednotlivých oblastí, ktoré si detailne rozoberieme.



DOS MZ Hlavička je prvá oblasť, ktorú si môžeme predstaviť ako obyčajnú štruktúru obsahujúcu niekoľko položiek. Hneď prvá z nich je dvojslovo MZ, ktoré označuje, že sa jedná o DOS spustiteľný súbor. To je kvôli spätnej kompatibilite so staršími verziami Windowsu, ktoré PE formát nepoznajú. Pre nás dôležitou položkou je tá posledná, ktorá obsahuje 4-bajtový ofset do PE hlavičky.

DOS stub je druhá oblasť, ktorá sa tu nachádza len kvôli spätnej kompatibilite. Ak staršia verzia systému nepozná PE formát (napr. DOS), tak ofset do PE hlavičky bude pre ňu neznámy, tak ho odignoruje a dostane sa až sem do tejto oblasti. Ak sa jedná o verziu, ktorá PE formát rozoznáva, tak využije ofset do PE hlavičky, prejde na jej adresu a tak sa nikdy nedostane sem. Tu sa len užívateľovi vypíše správa „Tento program vyžaduje Windows“.

PE hlavička je ďalšia oblasť, do ktorej sa dostane už len tá verzia systému, ktorá PE formát spozná, čiže vie, že posledná položka v DOS MZ štruktúre je ofset do tejto oblasti. PE hlavička je taktiež štruktúra, oveľa zložitejšia ako tá predošlá a pre nás aj veľmi podstatná, keďže obsahuje dôležité údaje pre systémový spúšťač, ktorý zavádza daný formát do pamäte, a preto sa do nej pozrieme hlbšie. Štruktúra PE hlavičky sa inak nazýva IMAGE_NT_HEADERS a obsahuje nasledovné položky:

- Signature, je to podpis obsahujúci znaky „PE“ nasledovaný dvoma nulami
- IMAGE_FILE_HEADER, malá štruktúra obsahujúca niekoľko údajov:
    - Machine, CPU platforma, pre ktorú je súbor určený
    - NumberOfSections, počet sekcií, ktoré sa nachádzajú za tabuľkou sekcií
    - TimeDateStamp, dátum a čas vytvorenia súboru
    - PointerToSymbolTable, adresa do tabuľky symbolov, využíva sa pri ladení programu
    - NumberOfSymbols, počet symbolov, taktiež na účeli ladenia
    - SizeOfOptionalHeader, veľkosť štruktúry, ktorá nasleduje hneď po štruktúre IMAGE_FILE_HEADER, táto hodnota musí byť platná
    - Characteristics, informácia o súbore, či je to napr. EXE alebo DLL
- IMAGE_OPTIONAL_HEADER32, štruktúra, ktorá obsahuje kľúčové údaje:
    - AddressOfEntryPoint, RVA (relatívna virtuálna adresa) prvej inštrukcie, ktorá sa vykoná, keď bude súbor pripravený na spustenie

- ImageBase, je preferovaná adresa zavedenia PE súboru do pamäte, spúšťač sa pokúsi na túto adresu vo virtuálnom adresnom priestore zaviesť PE súbor iba ak je to miesto voľné, inak bude musieť zaviesť súbor na inú adresu, čo pre nás znamená, že na túto hodnotu sa nemožno spoliehať
- SectionAlignment, granularita zarovnania sekcií v pamäti, ak je napríklad táto hodnota nastavená na 1000h, tak každá sekcia bude namapovaná do adresného priestoru na adresu, ktorá je násobkom tejto hodnoty, časť priestoru medzi koncom sekcie a začiatkom ďalšej bude nevyužitý
- FileAlignment, je to isté ako SectionAlignment, ale jedná sa o zarovnanie sekcií v súbore na disku, kde je PE súbor uložený a následne je z týchto sekcií namapovaný do pamäte
- MajorSubsystemVersion/MinorSubsystemVersion, verzie Win32 subsystému
- SizeOfImage, celková veľkosť PE obrazu v pamäti, je to súčet všetkých hlavičiek a sekcií zarovnaných na najbližší násobok hodnoty SectionAlignment
- SizeOfHeaders, veľkosť všetkých hlavičiek plus veľkosť tabuľky sekcií
- Subsystem, informácia o tom, pre ktorý NT subsystém je PE súbor určený, najčastejšie sú to dve možné hodnoty: Windows GUI a Windows CUI (konzola)
- DataDirectory, je pole štruktúr IMAGE_DATA_DIRECTORY, ktorých je v poli presne 16, dve štruktúry sú pre nás kľúčové, pretože obsahujú ofsety do tabuliek importov a exportov

Tabuľka sekcií je ďalšia oblasť, ktorá nasleduje hneď po PE hlavičke. Tabuľka sekcií je pole štruktúr IMAGE_SECTION_HEADER a ich počet je rovný hodnote NumberOfSections, ktorú nám poskytuje štruktúra IMAGE_FILE_HEADER, pomocou ktorej vieme koľko štruktúr sa tu nachádza. Každá jedna štruktúra popisuje práve jednu zo sekcií, ktoré v rovnakom poradí nasledujú za tabuľkou sekcií. Štruktúra obsahuje nasledujúce údaje:

- Name1, jedná sa o ASCII reťazec s veľkosťou 8 bajtov, ktorý reprezentuje názov sekcie, tento názov je len informatívny a neplní žiadnu inú funkciu
- VirtualAddress, RVA sekcie, keď spúšťač mapuje sekciu do pamäte, zavedie ju na adresu, ktorá je určená súčtom tejto hodnoty a adresy, na ktorú bol PE súbor zavedený
- SizeOfRawData, veľkosť dát v sekcii zarovnaná na najbližší násobok hodnoty FileAlignment, túto hodnotu využije spúšťač, aby vedel koľko bajtov ma namapovať do pamäte zo súboru
- PointerToRawData, ofset začiatku sekcie v súbore, túto hodnotu spúšťač využije, aby vedel na ktorej adrese v súbore má hľadať sekciu
- Characteristics, sú to príznaky, ktoré informujú spúšťač o tom či sekcia obsahuje spustiteľný kód, inicializované dáta, neinicializované dáta, či sa môže do sekcie zapisovať, čítať zo sekcie, alebo spustiť kód

Sekcia je posledná oblasť v PE súbore, ale môže ich byť až niekoľko, presný počet udáva NumberOfSections. Sekcia je blok dát so spoločnými atribútmi ako kód, dáta, čítať, zapisovať, atď. Zoskupenie dát do sekcie sa vykonáva na základe spoločných atribútov a nie na logickom základe. Nezáleži na tom, ako sú dáta/kód použité, v prípade keď majú rovnaké atribúty, môžu byť spolu v jednej sekcii. To uľahčuje prácu spúšťaču, ktorý mapuje sekcie do pamäte a zároveň dáva bloku pamäte obsadenej danou sekciou uvedené atribúty.

2. 2 Import/Export oblasti

Importovaná funkcia je funkcia, ktorá nie je vo volajúcom module, ale je volaná modulom. Tieto funkcie sú uložené v DLL knižniciach. Avšak na to aby modul mohol danú funkciu použiť, musí definovať akú funkciu chce volať a z akej knižnice. Bez toho je spúšťač bezmocný a nedokáže funkciu lokalizovať. Miesto ktoré obsahuje definície funkcií, ktoré sa importujú, sa nazýva Import Tabuľka. Tak ako ju má súbor EXE, tak isto ju môže mať aj DLL knižnica, ktorá tiež na niektoré činnosti potrebuje využiť funkcie iných DLL knižníc.

Presne naopak je to s exportovanou funkciou. Tá je implementovaná v DLL knižnici a je deklarovaná na export, čiže okolité moduly si ju môžu importovať. Miesto v DLL knižnici, ktoré obsahuje zoznam exportovaných funkcií sa nazýva Export Tabuľka.

2. 2. 1 Import tabuľka

Na to aby sme sa dopátrali, kde v PE súbore sa nachádza Import tabuľka, musíme ešte urobiť prieskum štruktúry IMAGE_OPTIONAL_HEADER32, ktorá obsahuje aj pole štruktúr IMAGE_DATA_DIRECTORY. Táto štruktúra obsahuje dva prvky:

- VirtualAddress, RVA pre danú štruktúru, ktorá je reprezentovaná štruktúrou IMAGE_DATA_DIRECTORY
- iSize, udáva veľkosť dátovej štruktúry v bajtoch

Aby sme sa dostali k Import tabuľke, potrebujeme preskúmať hodnoty v poli na pozícii 2, kde sa nachádza Import Table. Takto sa vieme dostať na adresu prvej štruktúry IMAGE_IMPORT_DESCRIPTOR, ktorá popisuje jednotlivé importy.

Štruktúra IMAGE_IMPORT_DESCRIPTOR obsahuje:

- Characteristics / OriginalFirstThunk je to union, čiže oba názvy ukazujú na tie isté dáta, táto hodnota obsahuje RVA na pole štruktúr IMAGE_THUNK_DATA, táto štruktúra má dva prvky a ukazuje na štruktúru IMAGE_IMPORT_BY_NAME a túto štruktúru si predstavíme neskôr
- TimeDateStamp, táto hodnota nás veľmi nezaujíma, väčšinou je vynulovaná
- ForwarderChain, táto hodnota nás tiež veľmi nezaujíma, väčšinou je tiež vynulovaná
- Name1, obsahuje RVA názvu DLL knižnice, tento názov je kódovaný v ASCII, čiže za názvom nasleduje nulový bajt
- FirstThunk, podobne ako OriginalFirstThunk ukazuje na pole štruktúr IMAGE_THUNK_DATA (samozrejme na iné pole ako ukazuje OriginalFirstThunk)

Je dôležité vysvetliť, prečo OriginalFirstThunk a FirstThunk ukazujú na IMAGE_THUNK_DATA a tá ukazuje na štruktúru IMAGE_IMPORT_BY_NAME, ktorá obsahuje :

- Hint, dvojslovo, ktoré slúži na rýchlejšie vyhľadávanie funkcie v export tabuľke danej DLL knižnice
- Name1, ASCII názov funkcie, ukončený nulou

Dôvod je ten, že DLL knižnica neexportuje všetky funkcie rovnako. Prevažne väčšinu exportuje podľa názvu, v tom prípade IMAGE_THUK_DATA ukazuje na IMAGE_IMPORT_BY_NAME. Niektoré funkcie však nie sú exportované podľa názvu, ale sú ordinálne podľa čísla. V tom prípade štruktúra IMAGE_THUK_DATA neobsahuje ukazovateľ na štruktúru, ale obsahuje ordinálne číslo funkcie v DLL knižnici.

Aby však spúšťač vedel, či daná hodnota je ofset na štruktúru alebo ordinálne číslo funkcie, tak táto 4-bajtová hodnota má MSB (najvýznamnejší ľavý bit) nastavený na 1. To znamená, že táto 4-bajtová hodnota je potom jednoznačne identifikovateľná, či je ofset, alebo obsahuje ordinálne číslo, keďže ofset s MSB nastaveným na 1 je príliž veľký na to, aby mohol odkazovať na adresu.

Veľmi dôležité je pochopiť ako to je teda s OriginalFirstThunk a FirstThunk. Pri čítaní štruktúr IMAGE_IMPORT_DESCRIPTOR sú dva možné scenáre.

1. OriginalFirstThunk je nenulový a ukazuje na pole štruktúr IMAGE_THUNK_DATA, ktoré buď obsahujú ordinálne číslo funkcie, alebo ukazateľ na IMAGE_IMPORT_BY_NAME, zatiaľ čo FirstThunk najprv tiež ukazuje na pole štruktúr IMAGE_THUNK_DATA, no neskôr však spúšťač tieto štruktúry nahradí reálnymi adresami funkcií. Takto si spúšťač udržiava zoznam názvov funkcií ako a ich reálne adresy.
2. OriginalFirstThunk je nulový, to znamená že FirstThunk ukazuje ako v predošlom prípade na pole štruktúr IMAGE_THUNK_DATA. Potom ako sa spúšťač dostane k týmto údajom, prepíše IMAGE_THUNK_DATA na adresu príslušnej funkcie. Takto si už neudržiava názvy funkcií, ale iba ich adresy.

2. 2. 2 Export tabuľka

Keď dochádza k exportovaniu funkcie pre EXE alebo DLL, tak môže dvoma rozličnými spôsobmi. Prvý spôsob je, že funkcia je exportovaná podľa názvu, takže ak program, alebo knižnica chce danú knižnicu importovať, musí vedieť jej presný názov. Druhý spôsob, je že funkcia je exportovaná podľa čísla, čiže program, ktorý danú funkciu importuje podľa ordinálu, vie jej 16-bitovú hodnotu a tá je v rámci export tabuľky jedinečná.

Hlavnou štruktúrou v export tabuľke je IMAGE_EXPORT_DIRECTORY, ktorá obsahuje tieto členy:

- nName, aktuálny názov modulu, keďže meno súboru sa môže zmeniť
- NumberOfFunctions, celkový počet funkcií/symbolov, ktoré sú exportované v tomto moduli
- NumberOfNames, počet funkcií/symbolov, ktoré sú exportované v tomto module podľa mena
- AddressOfFunctions, tento údaj ukazuje na RVA poľa, ktoré obsahuje RVA všetkých funkcií/symbolov, ktoré sú v moduli exportované
- AddressOfNames, RVA na pole, ktoré udržiava RVA všetkých mien funkcií exportovaných v tomto module
- AddressOfNamesOrdinals, RVA na 16-bitové pole, ktoré udržiava ordinálne čísla spojené s názvami funkcií, ktoré sa nachádzajú v poli AddressOfNames

Export tabuľka je určená predovšetkým pre spúšťač, ktorý jednotlivým importovaným funkciám prideľuje adresy na základe export tabuľky modulu, v ktorom sa funkcia nachádza. Pre nás je štruktúra dôležitá z dôvodu, že našou snahou neskôr bude oklamať spúšťač a podhodiť mu niektoré nepravé adresy.

Samozrejme táto časť neposkytuje kompletný pohľad na štruktúru, vyberali sme len časti, ktoré sú relevantné pre naše použitie. Doplňujúce informácie možno nájsť na stránke [3].

3 Vytvorenie procesu

K vytvoreniu procesu prichádza, keď aplikácia zavolá jednu z funkcií vytvárania procesov, ako sú napr. CreateProcess, CreateProcessAsUser, CreateProcessWithTokenW alebo CreateProcessWithLogonW. Vytvorenie procesu systému Windows pozostáva z niekoľkých fáz, ktoré sa odohrávajú v troch súčastiach operačného systému: v klientskej knižnici Kernel32.dll Windows, vo výkonnej časti Windows a v procese subsystému Windows (Csrss). Architektúra Windows podporuje viac subsystémov prostredia a tak je vytvorenie výkonného objektu procesu Windows oddelené od činností súvisiacimi s vytváraním procesu Windows.

Nasledujúce časti čerpali z knihy [1], ktorá opisuje jadro systému Windows

3. 1 Funkcia CreateProcess

Ešte pred otvorením spusteného vykonateľného obrazu uskutoční funkcia CreateProcess niekoľko krokov. Vo funkcii CreateProcess je trieda priorít nového procesu špecifikovaná vo forme nezávislých bitov v parametri CreationFlags. Preto je možné v jednom volaní funkcie CreateProcess špecifikovať viac než jednu prioritu. Systém Windows rieši otázku tried priorít priraďovaných danému procesu prostou voľbou množiny. Ak nie je novému procesu špecifikovaná žiadna trieda priorít, trieda priorít sa štandardne nastaví na normálnu. Funkcia CreateProcess má 6 hlavných fáz, ktoré si podrobne prejdeme.

Fáza 1: Otvorenie obrazu, ktorý sa má vykonať. Cieľom tejto fázy, ako názov naznačuje, je nájsť príslušný obraz Windows, ktorý spustí daný vykonateľný súbor zadaný volajúcim a vytvorí objekt sekcie pre jeho neskoršie mapovanie do adresného priestoru nového procesu. Ak je zadaným vykonateľným súborom súbor .exe pre Windows, použije sa priamo. Keď sa však nejedná o súbor .exe pre Windows, tak funkcia CreateProcess určitými krokmi hľadá podporný obraz Windows nevyhnutný k jeho spusteniu. Nás ale zaujíma súbor .exe, takže ďalšie kroky funkcie CreateProcess v tejto fáze nás už nezaujímajú.

Fáza 2: Tvorba objektu procesu výkonnej časti. V tomto okamžiku má CreateProcess otvorený platný spustiteľný súbor Windows a vytvorený objekt sekcie pre jeho mapovanie do adresného priestoru nového procesu. Následne vytvorí objekt behu aplikácie v systéme Windows, v ktorom daný obraz pobeží, zavolaním internej systémovej funkcie NtCreateProcess. Vytvorenie objektu behu (čo sa uskutočňuje vytvorením vlákna) zahrňuje nasledovné podfázy:

- príprava bloku EPROCESS, ktorý zahrňuje mnoho atribútov súvisiacich s daným procesom a radu ďalších súvisiacich dátových štruktúr, na ktoré ukazuje
- vytvorenie počiatočného adresného priestoru procesu, ktorý sa skladá z adresára stránok, stránky hyperpriestoru a zoznamu pracovnej množiny
- inicializácia bloku procesu jadra (KPROCESS), ktorý obsahuje ukazateľ na zoznam vlákien jadra; blok procesov jadra, taktiež ukazuje na adresár tabuliek stránok procesov, celkový čas vykonávania vlákien procesu, východziu základnú prioritu plánovania procesov
- dokončenie nastavenia adresného priestoru procesu, čo zahrňuje inicializáciu zoznamu pracovnej množiny a popisovačov virtuálneho adresného priestoru a mapovanie obrazu do tohto adresného priestoru
- nastavenie bloku PEB (Process Environment Block), ktorý na rozdiel od EPROCESS a iných štruktúr sa ako jediný nenachádza v systémovom priestore, ale v adresnom priestore procesu, ako hovorí aj samotný názov bloku, pretože obsahuje informácie menené kódom bežiacim v užívateľskom režime, napr. je tu uložená základná adresa obrazu, na ktorú je spustený proces namapovaný
- dokončenie nastavenia objektu behu, napr. funkcia CreateProcess vloží blok nového procesu na koniec zoznamu aktívnych procesov systému Windows, nastaví čas vytvorenia procesu a volajúcemu sa vráti manipulátor tohto nového procesu.

Fáza 3: Vytvorenie počiatočného vlákna a jeho zásobníka a kontextu. V tomto okamihu je objekt behu programu v systéme už plne nastavený. Nemá však žiadne vlákno, takže nemôže nič robiť. Skôr ako môže byť vytvorené príslušné vlákno, potrebuje zásobník a kontext pre svoj beh, takže tieto veci sa nastavia v tejto fáze. Veľkosť zásobníka počiatočného vlákna sa preberá z obrazu – žiadnym iným spôsobom

sa nedá zadať iná veľkosť. Po tomto už môže byť vytvorené počiatočné vlákno, čo sa uskutoční volaním funkcie NtCreateThread. Vlákno však nebude zatiaľ nič robiť, vytvorí sa v pozastavenom stave a k jeho spusteniu nedôjde, pokiaľ nebude celý proces plne inicializovaný.

Fáza 4: Upozornenie subsystému Windows na nový proces. V tomto okamihu sú už vytvorené všetky potrebné objekty procesu a vlákna výkonnej časti. Kernel32.dll následne odošle subsystému Windows správu, aby sa mohol pripraviť na nový proces a vlákno.

Fáza 5: Spustenie vykonávania počiatočného vlákna. V tejto fáze už je určené prostredie procesu, sú alokované prostriedky využívané jeho vláknami, proces má vlákno a subsystém Windows vie o novom procese. Pokial volajúci nezadal príznak CREATE_SUSPENDED, začne sa vykonávať počiatočné vlákno, aby sa mohlo rozbehnúť a zaistiť zbytok práce inicializácie procesu, ktorý sa deje v kontexte nového procesu (fáza 6). Pokial volajúci zavolal CreateProcess s príznakom CREATE_PROCESS tak v tejto fáze zostane proces pozastavený a pokračovať sa bude až keď volajúci zavolá funkciu ResumeThread.

Fáza 6: Vykonanie inicializácie procesu v kontexte nového procesu. Nové vlákno začne svoj život vykonaním spúšťacej rutiny KiThreadStartup, ktorá beží v režime jadra. KiThreadStartup zníži svoju úroveň a zavolá systémovú počiatočnú rutinu vlákien PspUserThreadStartup. Následne sa na port ladenia procesu odošle správa o vytvorení procesu, aby subsystém mohol príslušnému procesu debuggeru podať ladiacu udalosť spustenia procesu (funkcia CreateProcess bola zavolaná s príznakom CREATE_PROCESS_DEBUG_INFO). PspUserThreadStartup potom čaká na to, kým subsystém Windows neobdrží odpoveď od debuggeru (prostredníctvom funkcie ContinueDebugEvent). Akonáhle subsystém Windows odpovie, znovu sa spustia všetky vlákna. Po skončení funkcie PspUserThreadStartup sa zavolá rutina LdrInitializeThuk. Táto rutina inicializuje správcu haldy, tabuľky NLS, pole úložiska lokálneho vlákna (TLS) a štruktúry kritickej sekcie. Potom zavedie všetky požadované knižnice DLL a zavolá vstupné body knižníc DLL funkčným kódom DLL_PROCESS_ATTACH. Nakoniec začne vykonávanie obrazu v užívateľskom režime.

3. 2 Zavedenie DLL do pamäte

DLL knižnica je k spustiteľnému súboru prilinkovaná dynamicky, čiže za behu programu. Z toho je aj to označenie Dynamic Load Library, navyše jedna knižnica môže byť simultánne využívaná viacerými programami, čo zdokonaľuje multitasking.
K dynamickému linkovaniu môže dôjsť dvojako. Rozlišujeme implicitnú väzbu a explicitnú väzbu.

Implicitná väzba sa vykonáva hneď po spustení spustiteľného súboru. Ako sme mohli vidieť v predchádzajúcej podkapitole, dochádza k tomuto linkovaniu vo fáze 6 funkcie CreateProcess. V import tabuľke sa nachádzajú názvy jednotlivých knižníc, ktoré spúšťač použije, aby našiel danú knižnicu buď v priečinku knižníc systému alebo v priečinku, kde sa nachádza spustiteľný súbor. Následne alokuje v adresnom priestore miesto pre DLL knižnicu, kde ju namapuje a odošle jej správu DLL_PROCESS_ATTACH, čo znamená, že bola pripojená k procesu. Väčšinou túto správu knižnice využijú na inicializáciu prostriedkov alebo nastavenia. DLL knižnica implicitne nalinkovaná sa v pamäti nachádza až do ukončenia procesu, ku ktorému bola nalinkovaná. Vtedy jej príde správa DLL_PROCESS_DETACH a knižnica tak môže za sebou upratať a bude odobratá z pamäte.

Pri Explicitnej väzbe je knižnica do pamäte namapovaná explicitne, čiže keď si o ňu požiada program. Takéto linkovanie sa uskutoční volaním funkcie LoadLibrary, ktorá berie ako argument názov knižnice. Po tom, čo je knižnica namapovaná do pamäte, obdrží správu DLL_PROCESS_ATTACH podobne ako knižnica linkovaná implicitne. Rozdiel je pri volaní funkcií z explicitne linkovanej knižnice. Vtedy program najprv potrebuje mať handle na knižnicu, ktorú získa prostredníctvom funkcie GetModuleHandle a názov funkcie, ktorú chce zavolať. Tieto dva údaje, handle a názov funkcie, poskytne ako vstupné parametre funkcii GetProcAddress a tá mu vráti adresu funkcie v danej knižnici, ak sa tam nachádza. Narozdiel od implicitného linkovania, je knižnica uvoľnená buď systémom, keď program skončí alebo programom, keď zavolá funkciu FreeLibrary s názvom knižnice.

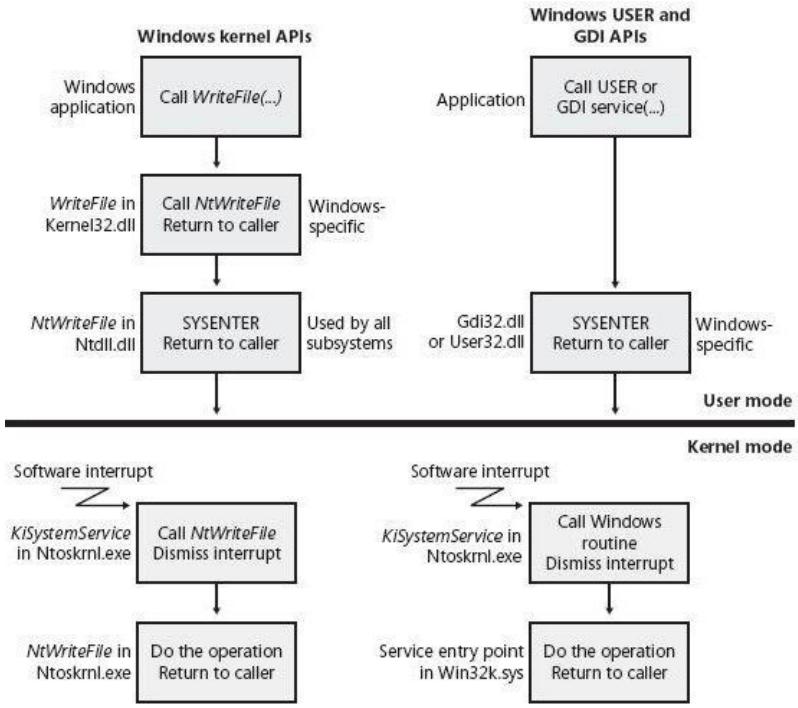
Ďalšie informácie možno nájsť v knihe [2].

4 Analýza prístupu ku kontrole aplikácie

V predchádzajúcich kapitolách sme prešli cez architektúru Windows, štruktúru PE formátu ž k vytvoreniu procesu, ktorý pozostáva z niekoľkých fáz. Tieto poznatky nám umožňujú analyzovať možné prístupy ku kontrole aplikácie v systéme Windows. Prístup ku kontrole aplikácie si môžeme predstaviť ako sledovanie volaní systémových funkcií Windows API, ktoré sú volané aplikáciou. Funkcia často prijíma vstupné parametre a vracia návratovú hodnotu, teda našim cieľom bude, aby sme vedeli tieto parametre zachytiť.

4. 1 Prehľad volania systémových funkcií

Nasledujúci obrázok zobrazuje priebeh volania API funkcií.



Obrázok bol prevzatý zo stránky [5].

Keď aplikácia zavolá funkciu WriteFile, ktorá sa nachádza v Kernel32.dll, tak tá nevykoná samotnú operáciu, ale zavolá funkciu NtWriteFile, ktorá sa nachádza v knižnici Ntdll.dll. Táto funkcia tiež operáciu nevykoná. Nastaví hodnoty registrov a vyvolá interrupt. Týmto interruptom sa snaží zavolať systémovú službu, ktorá beží

v jadre systému. Samozrejme hneď ako vyvolá interrupt, tak sa táto služba hneď nevykoná. Medzi užívateľským režimom a režimom jadra sa nachádza brána (gate), ktorá rozhoduje o tom či povolí volajúcemu zavolať systémovú službu.

Výnimkou sú funkcie USER a GDI Windows, pretože tieto funkcie majú inštrukcie odosielania systémových služieb implementované priamo v User32.dll a Gdi32.dll. V tomto prípade teda knižnica Ntdll.dll do celého procesu vôbec nevstupuje.

Knižnica Ntdll.dll predstavuje nezdokumentovanú časť systému Windows. Táto funkcia slúži predovšetkým na volanie systémových služieb. Jej funkcie vedia aké hodnoty sú potrebné pre interrupt a ako ho zavolať správne. Tieto nastavenia sa menia, preto tento prístup neovplyvní aplikácie, ktoré používajú zdokumentované API.

Niektoré aplikácie, nemusia volať funkciu WriteFile, ale zavolajú priamo funkciu NtWriteFile, alebo dokonca vyvolajú priamo interrupt. Takýto prístup je dosť pre aplikáciu nebezpečný, pretože nezdokumentovaná časť nie je stála, mení sa a môže spôsobiť, že pri aktualizovaní knižníc systému už nebude správne, alebo vôbec fungovať. To je aj význam knižníc ako Kernel32.dll, Advapi32.dll atď., že ich funkcie sa výrazne nemenia a umožňujú, aby po zmene implementácie volania systémových služieb, mohli aplikácie aj naďalej fungovať správne.

4. 2 Hlavné prístupy ku kontrole

Teraz sa naskytá hlavná otázka, kde sa pripojiť na komunikáciu a sledovať jednotlivé volania. Nasledovné podkapitoly, sa na tieto prístupy sledovania komunikácie bližšie pozerajú a analyzujú ich výhody, nevýhody.

4. 2. 1 Sledovanie aktivít v jadre systému

Tento spôsob sledovania aktivít spočíva v tom, že budeme sledovať komunikáciu medzi užívateľským režimom a režimom jadra. Takže budeme pri bráne a sledovať volanie systémových služieb. Jedna z možností, ako sa dostať medzi užívateľský režim a režim jadra, je modifikácia adresy, ktorá obsluhuje príslušné prepínacie prerušenie.

Tento spôsob má veľkú výhodu, že nám neunikne žiadne volanie systémovej služby. Takto vieme o všetkých operáciách, ktoré môžeme monitorovať. Vieme podľa hodnôt registrov určiť s akými parametrami sa služba volá a tiež aj zachytiť po jej skončení, s akým úspechom, resp. akú návratovú hodnotu vrátila. Tento prístup je navyše aj generický, umožňuje sledovať každú aplikáciu, nie iba jednu konkrétnu.

Nevýhodou tohto spôsobu sledovanie je ten, že aplikácia nepotrebuje na všetko systémové služby. Systémová služba stojí čas, preto veci, ktoré sa dajú urobiť v užívateľskom režime sa nepresúvajú do režimu jadra. Pri sledovaní systémových služieb nám uniká veľa volaní API funkcií, ktoré sa vykonajú v užívateľskom režime a my ich nezachytíme. Napríklad kopírovanie reťazca z jedného pamäťového miesta na iné si nevyžaduje systémovú službu, to by potom systém Windows nezvládol, keby mal každej aplikácií takéto operácie uskutočňovať v režime jadra.

4. 2. 2 Úprava priamo na disku

Iný prístup ku kontrole predstavuje úprava spustiteľného súboru priamo na disku. Táto technika spočíva v tom, že každý súbor je pred spustením na disku, či už EXE alebo DLL, a tak je možné ich pred spustením upraviť. Úprava by prebiehala v tom zmysle, že sa z import tabuľky cieľového EXE súboru, načítajú názvy funkcií a upraví sa kód funkcie, tak aby sa najprv vykonal náš kód a potom by sme prenechali akciu samotnej funkcii Tak by sa aplikácia spustila a jednotlivé volania by sme už mali zmapované a tak mohli sledovať volania API funkcií.

Výhoda tejto techniky je, že vieme zachytiť všetky volania API funkcií, ktoré chceme. Je jedno či aplikácia volá funkciu Kernel32.dll alebo Ntdll.dll, vieme si upraviť import tabuľku a začiatok funkcie tak , aby sa najprv vykonal náš kód.

Nevýhoda tejto techniky je, že to nie je generické. Pre každú aplikáciu potrebujeme osobitný prístup, čo výrazne komplikuje sledovanie rôznych aplikácii. Navyše sú tieto zmeny ľahko detekovateľné, takže aplikácia môže pri spustení úmyselne zlyhať kvôli zisteným zásahom do jej kódu. Taktiež môže nastať problém, ak je súbor na disku digitálne podpísaný, vtedy môžeme pri najlepšom podpis odstrániť, ale súbor stratí svoju dôveryhodnosť.

4. 2. 3 Podvrhnutie falošnej knižnice

Tretím možným prístupom je podvrhnutie falošnej knižnice. Tento prístup znamená, že pri štarte aplikácie nedostane aplikácia priamo knižnicu, ktorú potrebuje, ale dodáme jej našu „falošnú“, ktorá bude posúvať volania funkcií „originálnej“ knižnici. Týmto spôsobom sa vieme dostať medzi aplikáciu a DLL knižnicu inou knižnicou, ktorá v očiach aplikácie bude vystupovať ako „originálna“ knižnica a v očiach „originálnej knižnice“ bude vystupovať, ako aplikácia, ktorá ju využíva. Takýmto spôsobom získame úplný prehľad o komunikácii medzi aplikáciou a knižnicou.

Podvrhnutie falošnej knižnice vyžaduje vedieť akým spôsobom sa aplikácia spúšťa, v akej fáze sú pripojené DLL knižnice. ako ich Windows vyhľadáva a pridáva do adresného priestoru. Všetky tieto veci sú popísané v kapitole č.3, kde sa rozoberá vytvorenie procesu. Na základe popisov funkcie CreateProcess vieme, že DLL knižnica sa pripája vo fáze č. 6 a v predošlej fáze č. 5 sa spracúva príznak CREATE_SUSPENDED.

To znamená, že ak vytvoríme proces s príznakom CREATE_SUSPENDED, vytvoríme proces v suspendovanom režime. To nám umožňuje sa dostať do import tabuľky ešte predtým, ako spúšťač načíta názvy knižníc. Keď tieto názvy upravíme, tak spúšťač si natiahne našu „falošnú“ knižnicu a zavolá jej funkcie, ktoré neurobia nič iné ako to, že zaznamenajú vstupné parametre, zavolajú „originálnu“ funkciu a po jej dokončení zaznamenajú výstup funkcie a prenechajú riadenie späť programu.

Tento prístup má veľkú výhodu, pretože je generický. Vieme ktorejkoľvek aplikácií túto knižnicu podhodiť a takto sledovať volania jej jednotlivých funkcií. Ďalšia výhoda je tá, že je pre aplikáciu veľmi ťažké zistiť, či naozaj používa knižnicu, ktorú importuje, pretože nedochádza k zmene kódu, funkcie korektne posúvajú parametre originálnej funkcii a následne vracajú výstupné parametre volajúcemu programu.

Nevýhoda tohto prístupu je, že nevieme čo sa deje uprostred samotnej aplikácie, keďže vieme zachytiť len jej komunikáciu s okolitým prostredím. Ďalšou nevýhodou je, že pre každú knižnicu, potrebujem vytvoriť novú falošnú knižnicu, čo je dosť pracné, ak si vezmeme, koľko funkcií je v jednej Windows API knižnici. Ale túto nevýhodu je možné eliminovať tým, že vieme zobrať existujúcu knižnicu a vytvoriť pomocou algoritmu novú, ktorá obsahuje definíciu každej exportovanej funkcie.

4. 3 Zhrnutie prístupov

	Generickosť	Mód	Detekcia	Pokrytie
V jadre	Áno	Kernel	Ťažká	Systémové
Úprava disku	Nie	Užívateľský	Ľahká	Vysoké
Podvrhnutie	Áno	Užívateľský	Ťažká	Vysoké

Uvedená tabuľka popisuje hlavné vlastnosti jednotlivých prístupov, ktoré sme opísali. Každý prístup má svoje výhody a nevýhody, ale z tabuľky je zrejmé, že ako najvhodnejší pre nás sa javí tretí prístup. Z tohto dôvodu sme sa rozhodli implementovať práve tento prístup.

5 Implementácia prístupu

Ako prístup ku kontrole aplikácie sme si zvolili prístup založený na podvrhnutí falošnej knižnice, ktorý je pre nás najvhodnejší. Spôsob akým sa to dá dosiahnuť, sme popísali veľmi abstraktne, a preto táto kapitola popisuje jednotlivé úskalia pri implementácii zvoleného prístupu.

5. 1 Implementačné detaily

Hlavné body implementácie prístupu:

- Načítanie štruktúry IMAGE_NT_HEADERS
- Spustenie procesu v suspendovanom stave
- Nájdenie adresy import tabuľky
- Načítanie import tabuľky
- Vytvorenie falošnej knižnice
- Podvrhnutie falošnej knižnice
    - Premenovaním funkcií
    - Skopírovaním export tabuľky
- Podvrhnutie explicitne linkovanej knižnice

V nasledujúcich podkapitolách sú uvedené detaily pri implementácii hlavných bodov. Niektoré body patria do jednej podkapitoly, keďže spolu úzko veľmi súvisia.

5. 1. 1 Spustenie procesu do suspendovaného stavu

Aplikácia, ktorá je prílohou tejto práce, bola napísaná v C++, kde sme mohli priamo využiť Windows.h, ktorý obsahuje zadefinované štruktúry vrátane tej hlavnej IMAGE_NT_HEADERS. Navyše je k dispozícii aj metóda GetImageNtHeader, ktorá berie ako parameter ofset začiatku PE hlavičky a vráti štruktúru IMAGE_NT_HEADERS. Okrem tejto funkcie sa dá hlavička načítať aj z disku, prečítaním veľkosti bajtov rovnej veľkosti štruktúry IMAGE_NT_HEADERS.

Spusteniu procesu sme sa venovali v kapitole č. 3, kde sme si ukázali jednotlivé fázy funkcie CreateProcess. Na základe týchto vedomostí vieme, že aby sme spustili proces v suspendovanom stave stačí zadať príznak CREATE_SUSPENDED. Funkcia CreateProcess pri úspešnom spustení procesu vráti true a do vstupného parametru, ktorý berie referenciou, nie hodnotou, vráti handle procesu.

Aplikačné prostredie Windows umožňuje všetkým procesom, aby mohli pristupovať k adresnému priestoru ostatných procesov, ktoré tiež bežia v ring 3, čo je najnižšie privilegovaná úroveň. Prístup znamená čítať z pamäte iného procesu a zapisovať do pamäte iného procesu. To je pre nás veľmi dôležité, pretože bez tohto prístupu by náš plán nemohol byť zrealizovaný. Windows API funkcie, ktoré umožňujú prístup k inému procesu obsahujú na konci názvu „Ex“, čo znamená „Extended“ (rozšírený). Napr. ak funkcia VirutalAlloc umožňuje procesu vytvoriť vo svojom adresnom priestore novú pamäťovú oblasť, tak funkcia VirtualAllocEx umožní procesu vytvoriť novú pamäťovú oblasť v inom procese.

5. 1. 2 Spracovanie import tabuľky

Aby sme dostali k import tabuľke, potrebujeme vedieť dve hodnoty: ofset import tabuľky a adresu, na ktorú bol súbor namapovaný do pamäte. Prvú hodnotu máme po načítaní štruktúry IMAGE_NT_HEADERS a druhú hodnotu nemáme zatiaľ k dispozícii. V štruktúre IMAGE_NT_HEADERS sa nachádza aj ImageBase, ktorý obsahuje hodnotu, na ktorú by proces mal byť namapovaný, ale nie vždy sa tak stane. Preto sa na túto hodnotu nemožno spoľahnúť. K tejto adrese sa dostaneme volaním funkcie GetThreadContext:

```
CONTEXT con;
con.ContextFlags = CONTEXT_FULL;
GetThreadContext(process_handle, &con);
```

Funkcia GetThreadContext ako parameter berie handle procesu a ukazateľ na štruktúru CONTEXT, do ktorej uloží špecifické dáta o procese. Preto aby sme dostali úplnú štruktúru CONTEXT je za potreby nastaviť ContextFlags na hodnotu CONTEXT_FULL. Po skončení funkcie GetThreadContext sa nachádzajú v štruktúre

dáta o procese potrebné pre procesor. Nás zaujíma jedna jediná hodnota a tou je hodnota registra EBX, ktorý obsahuje adresu PEB tabuľky suspendovaného procesu. V tejto tabuľke sa na ofsete 8 nachádza ImageBaseAddress, ktorý obsahuje adresu, na ktorú bol suspendovaný proces namapovaný.

```
DWORD image_offset = con.Ebx + 8;
ReadProcessMemory(process_handle, image_offset, &ImageBase, 4, NULL);
```

Takto zjednodušene vyzerá získanie ImageBaseAddress suspendovaného procesu. Teraz, keď už vieme základnú adresu procesu, vieme prečítať ľubovoľné pamäťové miesto procesu.

Keď už vieme adresu import tabuľky, nebráni nám nič v tom, aby sme ju prečítali. Na začiatku import tabuľky sa nachádza pole štruktúr IMAGE_IMPORT_DESCRIPTOR, aby sme vedeli kde toto pole končí, tak za poslednou štruktúrou sa nachádza prázdna štruktúra IMAGE_IMPORT_DESCRIPTOR, čiže vynulovaná. Na prvý pohľad sa zdá, že každá jedna štruktúra IMAGE_IMPORT_DESCRIPTOR, zodpovedá práve jednej DLL knižnici. Vo väčšine prípadov to tak je, ale nie je to zaručené.

Ak už máme načítané štruktúry, tak ich môžeme iteratívne prejsť a pre každú jednu štruktúru načítať jej názov. Toto načítavanie nie je zložité, stačí len čítať ASCII znaky, pokiaľ nenatrafíme na nulový znak. To či je OriginalFirstThunk nastavený na nulu, alebo obsahuje adresu nám môže byť úplne jedno. V tejto fáze, ktorej sa suspendovaný proces nachádza, nie sú ešte pridané adresy knižníc, takže FirstThunk ukazuje v podstate na rovnaké IMAGE_THUNK_DATA štruktúry. Táto štruktúra, ako sme spomínali v kapitole č. 2 obsahuje ofset na IMAGE_IMPORT_BY_NAME, ak je funkcia exportovaná názvom, alebo obsahuje ordinálne číslo funkcie, čiže funkcia je exportovaná číslom. Aby sme to zistili stačí nám urobiť jednoduché overenie či najľavší bit je nastavený na 1, ak áno jedná sa o ordinálne číslo, v opačnom prípade o ofset na IMAGE_IMPORT_BY_NAME štruktúru, ktorú načítame veľmi jednoducho, keďže obsahuje len dvojslovo a postupnosť ASCII znakov ukončenú nulou

5. 1. 3 Podhodenie falošnej knižnice

Po prečítaní import tabuľky už vieme, čo daný proces importuje a potrebuje pre svoj správny beh. Takmer každá aplikácia importuje knižnicu Kernel32.dll, ktorá je nevyhnutná pre beh aplikácie. Preto sme si túto knižnicu zvolili na podvrhnutie falošnej knižnice. Spúšťač postupuje pri čítaní import tabuľky presne ako my doposiaľ. Jediné čo o knižnici vie je jej názov, ktorý si prečíta ako my a vyhľadá takýto súbor na disku. V tomto prípade hľadá súbor s názvom Kernel32.dll a nájde ho úspešne v priečinku C:\Windows. Nič nám nebráni v tom, aby sme prepísali názov knižnice na Kernel32.fdl a tak prinútili spúšťač namiesto Kernel32.dll hľadať súbor Kernel32.fdl.

Je veľmi dôležité, aby dĺžka mena našej falošnej knižnice bola rovnaká ako tej pôvodnej, pretože nevieme, čo nasleduje priamo za názvom knižnice v pamäti. Takto sme podhodili spúšťaču našu falošnú knižnicu, ale to je len polovičný úspech. Po tom ako našiel našu knižnicu, začne skúmať export tabuľku našej falošnej knižnice. Existujú dve možné stratégie, ako dosiahnuť to, aby mapovanie názvov funkcií prebehlo úspešne. Ak neprebehne úspešne spúšťač oznámi chybu o tom, že nenašiel vstupný bod funkcie a proces tak neúspešne skončí.

Prvá stratégia spočíva v premenovaní funkcií. To znamená že ak IMAGE_IMPORT_BY_NAME obsahuje názov funkcie GetModuleHandleA, tak mi ho zmeníme na getModuleHandleA a tak zadeklarujeme v našej falošnej knižnici funkciu getModuleHandleA. Tento krok je veľmi dôležitý, pretože v našej funkcii getModuleHandleA chceme volať pôvodnú funkciu GetModuleHandleA a ak našej funkcii ponecháme rovnaký názov, tak spustíme nekonečnú rekurziu, čo si neželáme. Treba tiež zachovať dĺžku názvu knižnice z rovnakého dôvodu ako pri názve knižnice. Samozrejme musíme upraviť takto každú jednu funkciu, ktorá je importovaná, pretože táto stratégia vyžaduje, aby naša podhodená knižnica „falšovala“ každú jednu importovanú funkciu z pôvodnej knižnice.

Druhá stratégia spočíva tom, že skopírujeme export tabuľku pôvodnej knižnice. V suspendovanom procese vytvoríme nové pamäťové miesto pomocou funkcie VirtualAllocEx. Do tohto pamäťového miesta prekopírujeme export tabuľku, v našom príklade export tabuľku Kernel32.dll. V export tabuľke nahradíme adresy tých funkcií, ktoré naša falošná knižnica chce sledovať. Tam nastavíme adresu našej podhodenej funkcie z falošnej knižnice. Adresy funkcií, ktoré nás nezaujímajú musíme upraviť tak, aby ukazovali na správnu adresu. ImageBaseAddress knižnice Kernel32.dll vieme

získať veľmi jednoducho, keďže ju používa aj náš program, ktorý monitoruje aplikáciu. Takto len k RVA adresám pripočítame ImageBaseAddress a program bude nami neželané funkcie knižnice Kernel32 volať priamo mimo nášho dosahu.

Nie každá knižnica je linkovaná implicitne. Niektoré si program vyžiada za behu programu pomocou funkcie LoadLibrary, ktorá sa nachádza v knižnici Kernel32.dll. Ukázali sme ako sa dá podchytiť funkcia GetModuleHandleA, takže v prípade LoadLibrary, to tiež nebude problém. Naša falošná funkcia LoadLibrary, ktorú môžeme nazvať napríklad ToadLibrary, dostane názov knižnice, ktorú chce program explicitne nahrať do pamäte. Tu môžeme program taktiež veľmi ľahko oklamať a to tak, že funkcii LoadLibrary podhodíme názov falošnej knižnice. Potom keď sa bude program pýtať na adresy jednotlivých funkcií pomocou funkcie GetProcAddress, stačí, aby sme predtým programu podhodili našu funkciu getProcAddress a keď sa bude pýtať na funkciu, ktorú naša falošná funkcia definuje, tak podhodíme tú, v opačnom prípade pohodíme pôvodnú funkciu.

5. 1. 4 Zostrojenie falošnej knižnice

Ukázali sme ako teda vieme podhodiť falošnú knižnicu, ktorá je linkovaná implicitne alebo explicitne. Zostrojiť takúto knižnicu nie je nič zložité. Každá funkcia obsahuje DLLMain funkciu, ktorá spracúva správy, ktoré jej prídu od systému pri pripojení k procesu, k vláknu, alebo pri odpojení od procesu alebo vlákna.

Naša knižnica musí obsahovať deklaráciu funkcií, ktoré exportuje:

```
extern “C” __declspec(dllimport) FARPROC
getProcAddress(
        IN HMODULE hModule,
        IN LPCSTR lpProcName
);
```

Deklarácia funkcie getProcAddress vyžaduje slovo extern, čo znamená, že funkcia bude exportovaná. Označenie ”C” znamená, že ide o céčkovskú konvenciu pomenovania, čiže názov funkcie nebude dekorovaný znakmi, ktoré hovoria o tom, aké

prijíma funkcia parametre. Funkcie Windows API sú písané v C a tento jazyk neumožňuje preťažovanie funkcií, preto sú názvy jedinečné a musia také byť aj v našej falošnej knižnici.

Po deklarácii je potrebná definícia. Základnou úlohou našich funkcií je to, aby zapísali vstupné parametre, uložili ich na zásobník a zavolali pôvodnú funkciu. Po jej skončení zapíšu návratovú hodnotu, alebo hodnotu parametra, ktorý funkcia nastavuje a vrátia riadenie späť programu. Táto časť predstavuje monitorovanie aplikácie. Každá funkcia, ktorú chceme sledovať a program ju vyžaduje, bude zalogovaná a tak môže tento výpis veľmi pomôcť, napr. pri hľadaní chýb.

Dokumentáciu k API funkciám možno nájsť na stránke [4].

5. 2 Riadenie aplikácie

Doteraz sme sa venovali možnostiam monitorovania aplikácie. Vieme monitorovať len jej komunikáciu s Windows API, ktoré jej poskytuje funkcie na rôzne operácie, ktoré aplikácia sama nedokáže vykonať. Monitorovať samotný kód aplikácie však nevieme, to by bolo náročné a je to aj mimo hranice tejto práce. Monitorovať kód aplikácie si vyžaduje spustiť aplikáciu v ladiacom (debug) móde a analyzovať jednotlivé sady inštrukcií, čo je pomerne veľmi zložité. V konečnom dôsledku prínos takéhoto monitorovania nie je rovný námahe, ktorá je potrebná v realizáciu takéhoto monitorovania.

Riadeniu aplikácie sme sa zatiaľ nevenovali, ale vyplýva to z toho, že riadenie je závislé od monitorovania aplikácie. To znamená, že nemôžeme riadiť aplikáciu bez toho, aby sme ju prv monitorovali. Teraz, keď sa nám podarilo zrealizovať prístup na kontrolu aplikácie, môžeme uvažovať nad možnosťami jej riadenia. Ako sme spomenuli, vieme monitorovať len jej volania API funkcií, z toho vyplýva, že vieme riadiť len tieto volania. Takže ak nevieme monitorovať samotný kód aplikácie, tak nevieme ani riadiť tento kód.

Riadenie aplikácie sme teda obmedzili na riadenie volania API funkcií, ktoré aplikácii sprostredkúvajú systémové operácie. Možnosti, akými sa dajú riadiť tieto volania API funkcií sú nasledovné:

- Upraviť vstupné parametre
- Upraviť výstupné parametre
- Nezavolať pôvodnú API funkciu
- Zavolať inú API funkciu alebo funkcie

Upraviť vstupné parametre pre pôvodnú API funkciu je jedná z možností ako môžeme riadiť aplikáciu. Toto riadenie môže napríklad mať význam, ak chceme funkcii obmedziť niektoré príznaky, alebo chceme podhodiť vlastné parametre v závislosti od tých, ktoré aplikácia dala pôvodne, alebo úplne nezávisle od parametrov od aplikácie. Význam takéhoto riadenia závisí od API funkcie. Napríklad chceme pri volaní API funkcie aplikácii zakázať niektoré príznaky a tak jej obmedziť jej možnosti. Iný dôvod môže byť prepisovať reťazce, to už závisí od kontextu a treba byť opatrný a vedieť čo je pri prepísaní reťazca našou snahou. Napríklad pri falšovaní knižnice Kernel32.dll, ktorú premenujeme na Kernel32.fdl, potrebujeme často reťazec "Kernel32.fdl" nahradiť reťazcom "Kernel32.dll", keď potrebujeme, aby program pracoval s pôvodnou Kernel32.dll knižnicou.

Upraviť výstupné parametre je tiež jedná z možností, ale táto možnosť výrazne závisí od kontextu použitia. Funkcia ako výstupný parameter vracia informáciu o úspešnom behu, alebo vracia ukazateľ na štruktúru. K tejto možnosti treba pristupovať veľmi opatrne, pretože aplikácia môže zavolaním API funkcie niečo vykonať, ale úpravou návratovej hodnoty môže očakávať neúspech a to môže v konečnom dôsledku spôsobiť zlý postup aplikácie vedúci k závažnej chybe, ktorá ukončí beh aplikácie. Ale môže to zároveň poslúžiť na otestovanie behu aplikácie v neočakávaných prípadoch, čo môže byť veľmi nápomocné pri testovaní aplikácie.

Nezavolať pôvodnú API funkciu je jedna z možností obmedzenia prístupu aplikácie k jednotlivým prostriedkom. Namiesto volania originálnej funkcie, len zaznamenáme snahu o volanie funkcie a nevykonáme v podstate nič. Môžeme jej teda takto napríklad odoprieť volanie funkcie WriteFile, keď chceme aplikácii zakázať výstup do súboru a vrátime aplikácií len návratovú hodnotu o neúspešnosti zápisu do súboru.

Zavolať inú API funkciu alebo funkcie je tiež možnosť, ktorú máme k dispozícii. Sémantika využitia takejto možnosti veľmi závisí od kontextu. Využitie takejto možnosti môže byť cenné ak potrebujeme presmerovať volanie API funkcie na inú API

funkciu alebo funkcie Napríklad ak spúšťame aplikáciu na inej verzii operačného systému, môžeme niektoré API funkcie nahradiť inými, ktoré sú pre danú verziu operačného systému vhodnejšie.

Záver

V tejto bakalárskej práci sme v prvých kapitolách zhrnuli potrebné teoretické vedomosti potrebné k neskoršej analýze prístupov ku kontrole aplikácie a k implementácii zvoleného prístupu.

Zanalyzovali sme možné prístupy ku kontrole aplikácie, ktoré v našom prípade pripadali do úvahy. Na základe požiadaviek od jednotlivých prístupov na generickosť, detekciu, mód a pokrytie sme zvolili pre nás najvhodnejšiu stratégiu prístupu ku kontrole aplikácie, stratégiu podvrhnutia falošnej knižnice. Táto stratégia je založená na podvrhnutí knižnice, ktorá definuje rovnaké funkcie ako pôvodná aplikácia a tak vieme funkcie falošnej knižnice dosadiť namiesto tých pôvodných a takto presmerovať volanie API funkcií k nám.

Popísali sme implementačné detaily implementácie zvolenej stratégie, s ktorými sme sa stretli pri programovaní aplikácie. Implementačné detaily obsahujú postup od vytvorenia suspendovaného procesu, cez úpravu import tabuľky, až po vytvorenie falošnej knižnice, ktorú aplikácii podvrhneme namiesto pôvodnej knižnice.

Na záver sme poukázali na možnosti monitorovania aplikácie, keďže vieme monitorovať len komunikáciu aplikácie s Windows API. Monitorovať priamo kód aplikácie je nad rámec tejto práce, keďže si to vyžaduje spustiť proces v ladiacom móde a analyzovať sadu inštrukcií, čo je veľmi náročná úloha. Preto sme sa zamerali len na monitorovanie komunikácie aplikácie s Windows API.

Popísali sme možnosti riadenia komunikácie medzi aplikáciu a Windows API funkciou, ktoré nám umožňuje meniť vstupné parametre, výstupné parametre, zablokovať jednotlivé volania API funkcií alebo presmerovať volanie API funkcie na inú API funkciu alebo API funkcie.

Na základe poznatkov popísaných v tejto práci sme zostrojili aplikáciu, ktorá implementuje zvolený prístup a je prílohou k tejto práci.

Literatúra

Knihy

[1] RUSINOVICH, M. E. – SOLOMON, D. A. 2005. *Microsoft Windows Internals*. 4. vyd. Microsoft Press, ISBN 0-7356-1917-4.

[2] PIROGOR, V. 2005. *The Assembly programming Master Book*. A-LIST, ISBN 1931769362.

Internetové zdroje

[3] *Win32 Assembly Tutorials*.
Dostupné na internete: <http://win32assem bly.online.fr/tutorials>
dňa 23. 4. 2011.

[4] *MSDN Library*.
Dostupné na internete: <http://msdn.microsoft.com/en-us/library>
dňa 23.4.2011.

[5] *TechNet*.
Dostupné na internete: <http://technet.microsoft.com>
dňa 23.4.2011

Prílohy

Prílohou k tejto bakalárskej práci je aplikácia, ktorá bola naprogramovaná na základe poznatkov popísaných v tejto práci. Jej účelom je poukázať na správnosť zvolenej stratégie prístupu ku kontrole a riadeniu aplikácie.