SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1587

PROCESNA ZAŠTITNA STIJENA

Martin Grmek

Zagreb, Lipanj 2006.

Sažetak

U ovom radu je opisan sustav sigurnosne politike ponašanja procesa. Opisani su načini manipulacije aktivnim procesima. Sustav detekcije pokretanja novog procesa temelji se na funkciji `PsSetCreateProcessNotifyRoutine()` kojom se prijavljuje sveobuhvatna sustavska funkcija povratnog poziva. Unutar funkcije povratnog poziva obavlja se asinkrona komunikacija između jezgrinog upravljačkog programa i nadzorne aplikacije. Nadzorna aplikacija definira sustav sigurnosne politike za ponašanje procesa. Nakon dobivene informacije o novopokrenutom procesu, nadzorna aplikacija provjerava proces u sustavu sigurnosne politike, te postupa prema definiranom pravilu. U suprotnom, korisnik se informira o novom procesu. Daje se mogućnost korisniku da odluči što će učiniti sa procesom: da li će dopustiti da se nastavi izvođenje ili će ga uništiti. Nadzorna aplikacija prikazuje listu aktivnih procesa. Svaki aktivni proces moguće je uništiti, zaustaviti ili promijeniti mu prioritet. Nažalost, ova metoda za detekciju pokretanja procesa ne omogućuje nadzor nad pokretanjem sustavskih servisa i upravljačkih programa. Ovime se otvara prostor da „crvi“, „trojanski konji“ i neželjeni programi ipak budu pokrenuti bez znanja korisnika.

Abstract

This thesis deals with building a system process policy and manipulation with active processes. System for detecting process execution is based on kernel function `PsSetCreateProcessNotifyRoutine()` that enables registration system-wide callback function. Asynchronous process communication between kernel-mode driver and monitoring application is done inside callback function. Monitoring application defines system process policy. After notification about new process, monitoring application checks the system process policy and acts according to defined rule. Otherwise, monitoring application notifies user about new process and gives user a choice to terminate process or to continue running the process. Monitoring application contains a list of active processes. Process priority class can be changed for each process, also each process can be terminated or stopped. Unfortunately, this method for detecting process execution doesn't allow control of execution of system services and kernel drivers. This opens space for worms, trojan horses and malicious programs to be run without a knowledge of user.

Sadržaj

1. Uvod

Cilj ovog diplomskog rada je ostvariti zaštitu koja bi štitila operacijski sustav od pokretanja neželjenih aplikacija kao što su „crvi“, „trojanski konji“ i druge aplikacije sa skrivenim destruktivnim svojstvima. Zaštita radi na principu sigurnosne zaštitne stijene. Sigurnosna politika definira ponašanje pojedinih aplikacija, odnosno procesa. Drugim riječima, prilikom pokretanja aplikacije za koju nije definirana sigurnosna politika (kao i kod zaštitne stijene), korisniku bi se dalo na izbor što želi učiniti s trenutno pokrenutom aplikacijom: dopustiti ili zabraniti pokretanje. Svaka odluka, po želji korisnika, pamti se i sprema u dnevnik sustava sigurnosne politike.

Ovakav način zaštite sustava pouzdaniji je od bilo kojeg antivirusnog programa. Npr. ako je virus novi i nema ga u bazi virusa, antivirusni program ga nije u mogućnosti prepoznati sve dok proizvođač antivirusnog programa ne izda proširenje baze virusa. Do tog trenutka može proći i više dana, a za to vrijeme virus se može proširiti cijelim računalnim sustavom.

Ukoliko se neka aplikacija za koju je izgrađeno pravilo koje dopušta njezino izvršavanje, zarazi virusom, tada se promijeni veličina datoteke i podaci o trenutku zadnje izmjene izvršne datoteke. Ponovnim pokretanjem aplikacije sustav sigurnosne politike detektira novonastale promjene te obavještava korisnika da se aplikacija izmijenila od zadnjeg pokretanja. Sada korisnik može blokirati izvođenja takve zaražene aplikacije i spriječiti daljnje širenje virusa.

Sustav je moguće još dodatno pojednostaviti korištenjem dnevnika sigurnosne politike koji bi dopustili izvođenje nekih osnovnih programa i time olakšali korištenje „običnom“ korisniku.

2. Detekcija pokretanja aplikacija

Kako bi se mogla definirati politika ponašanja procesa, potrebno je u stvarnom vremenu detektirati kada se pokreće nova aplikacija, odnosno proces. Bitno je da se pokretanje novog procesa detektira što ranije, odnosno prije samog početka izvođenja procesa. Svaki novi proces mora biti zaustavljen odmah nakon stvaranja kako bi se izvršila provjera u sustavu sigurnosne politike. Ukoliko proces ima dozvolu za pokretanje, proces nastavlja s izvršavanjem; u suprotnom, proces završava.

Postoji više metoda detekcije pokretanja procesa. Spomenute su neke od metoda koje će u daljnjem tekstu biti detaljnije objašnjene:

1. Uporaba *DLL* (*Dynamic Link Library*) datoteke.
2. Uporaba jezgrene *API* (*Application Programming Interface*) funkcije `PsSetCreateProcessNotifyRoutine()`.
3. Modifikacija *SSDT* tablice (*System Service Dispatch Table*).

Praktični dio rada zasniva se na drugoj metodi, odnosno na korištenju jezgrene *API* funkcije `PsSetCreateProcessNotifyRoutine()`, koja se pokazala kao optimalno rješenje. Detekcija novih procesa vrlo je jednostavna korištenjem ove metode. Navedenom funkcijom prijavljuje se sveobuhvatna sustavska funkcija povratnog poziva, koji operacijski sustav poziva svaki puta kada ne pokreće novi proces ili se postojeći uništava.

2.1. Detekcija pokretanja procesa uporabom DLL datoteke

Ideja metode je da se svaki proces „prisili“ na učitavanje dobro poznate *DLL* datoteke prilikom pokretanja, koja bi detektirala pokretanja procesa.

Detekcija novog procesa unutar *DLL* datoteke riješena je na sljedeći način. U glavnoj `DllMain()` funkciji definira se odziv na poruku `DLL_PROCESS_ATTACH`. Prilikom učitavanja *DLL* datoteke u proces, poziva se glavne funkcije `DllMain()` s porukom `DLL_PROCESS_ATTACH`. Glavna funkcija *DLL* datoteke `DllMain()` izvršit će se u adresnom prostoru procesa koji je učitao *DLL* datoteku. Korištenjem *API* funkcije `GetCurrentProcessId()` dobiva se identifikacijski broj procesa iz kojeg se poziva funkcija. Budući se `DllMain()` funkcija izvršava u adresnom prostoru pozivajućeg procesa, poziv *API* funkcije `GetCurrentProcessId()` vratit će identifikacijski broj pozivajućeg procesa, odnosno procesa koji učitava *DLL* datoteku (novonastalog procesa). Kada se zna identifikacijski broj procesa korištenjem ostalih *API* funkcija, lako se dolazi do brojnih informacija vezanih uz proces.

Primjerice, vrlo važno za sustav sigurnosne politike je da se zna putanja do izvršne datoteke procesa. U tu svrhu koristi se *API* funkcija `GetModuleFileName()` koja za zadanu ručicu (*Handle*) vraća putanju datoteke na koju pokazuje ručica.

```
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD  reason_for_call,
                      LPVOID lpReserved)
{
  if(reason_for_call == DLL_PROCESS_ATTACH)
  {
    // dohvati ime procesa u kojem je učitan DLL
    char lib_name[MAX_PATH];
    GetModuleFileName(hModule, lib_name, MAX_PATH);
    DWORD pID = GetCurrentProcessId();

    // Javi aplikaciji da je proces kreiran
    ...
   }
   return TRUE;
}
```

Slika 2.1. Detekcija procesa `DllMain()` funkcijom

Nakon što je riješen problem detekcije koji proces učitava *DLL* datoteku, ostaje problem kako učitati *DLL* datoteku u svaki proces, tj. kako „prisiliti“ svaki proces da prilikom pokretanja učita unaprijed određenu *DLL* datoteku. To je moguće ostvariti na dva načina:

1. uporabom *registry*-a i
2. uporabom sveobuhvatnih sustavskih *Windows* zakački.

2.1.1. Detekcija pokretanja procesa uporabom *registry*-a

Kako bi se *DLL* datoteka učitala u procese koji koriste datoteku `USER32.DLL`, možemo jednostavno ubaciti naziv naše *DLL* datoteke u sljedeći *registry* ključ:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\Windows\AppInit_DLLs
```

Vrijednost ovog ključa sadrži naziv jedne *DLL* datoteke ili grupe *DLL* datoteka odvojenih ili zarezom ili razmakom. Prema *MSDN* (*Microsoft Developers Network*) dokumentaciji, sve *DLL* datoteke navedene kao vrijednost tog ključa, učitavaju se od svake *MS Windows* aplikacije pokrenute u trenutno prijavljenoj sjednici.

Zanimljivo je da je stvarno učitavanje navedenih *DLL* datoteka dio inicijalizacijskog postupka `USER32.DLL` datoteke. `USER32.DLL` u svojoj `DllMain()` funkciji čita vrijednost navedenog *registry* ključa i poziva funkciju `LoadLibrary()` za svaku od navedenih *DLL* datoteka.

Međutim, ovo se primjenjuje samo na aplikacije koje koriste `USER32.DLL`. Drugo ograničenje je da je ovaj mehanizam podržan samo u operacijskim sustavima *Windows NT*, 2000 i *XP*. Premda je ovo bezopasan način učitavanja *DLL* datoteke u procese, postoji nekolicina nedostataka [2]:

- U želji da se aktivira odnosno deaktivira proces učitavanja DLL datoteke, nužno se mora ponovno pokrenuti operacijski sustav.

- *DLL* datoteka koja služi za detekciju novih procesa, učitat će se jedino u procese koje koriste `USER32.DLL` datoteku. Konzolne aplikacije ne koriste `USER32.DLL` datoteku, tako da će informacija o njihovom pokretanju ostati nedostupna.
- Ne postoji nikakva kontrola nad učitavanjem *DLL* datoteke u procese. To znači da se ugrađuje u svaku *GUI* (*Graphics User Interface*) aplikaciju, bez obzira da li je to potrebno ili ne. Time se pojavljuje redundantni dodatak, koji može poprilično narasti ukoliko je trenutno aktivan velik broj aplikacija.
- Problem zaustavljanja procesa.

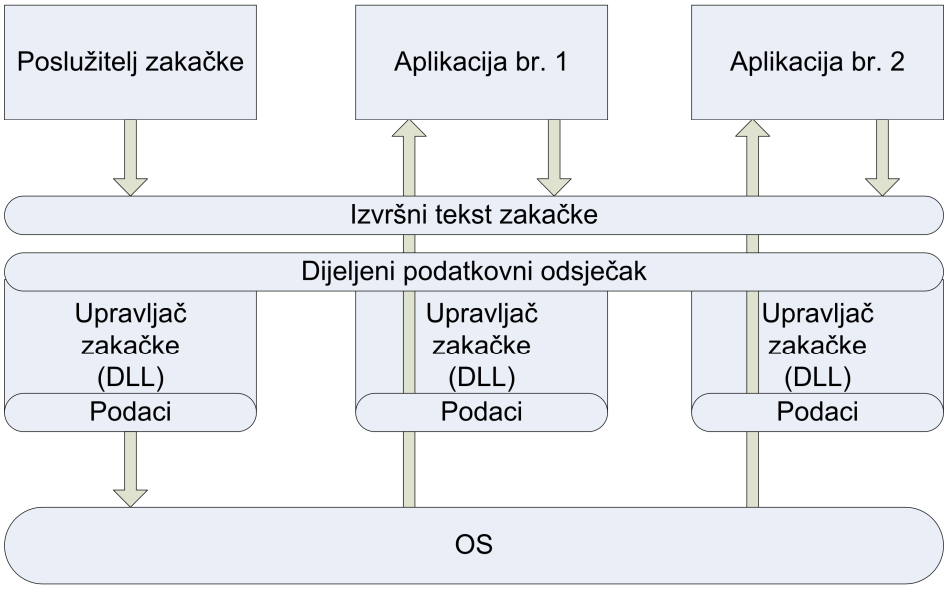
2.1.2. Uporaba sveobuhvatnih sustavskih *Windows* zakački

Jako popularan način učitavanja *DLL* datoteke u ciljani proces bazira se na *Windows* zakačkama (*Windows hooks*). Kako je pokazano u MSDN-u, zakačka je zamka u sustavu rada s porukama.

Aplikacija može instalirati filtar funkciju koja će pratiti promet poruka u sustavu i pojedinom procesu prije nego one stignu ciljnu proceduru.

Zakačka se normalno ostvaruje u *DLL* datoteci kako bi se udovoljili osnovni zahtjevi za sveobuhvatne sustavske zakačke (*system-wide hooks*). Osnovni koncept ove vrste zakački je da se procedura povratnog poziva zakačke izvrši u adresnom prostoru svakog zakačenog procesa u sustavu. Za instalaciju zakačke koristi se *API* funkcija `SetWindowsHookEx()` s odgovarajućim parametrima.

Jednom kada aplikacija instalira sveobuhvatnu sustavsku zakačku, operacijski sustav raspoređuje *DLL* datoteku u adresni prostor svakog procesa.



Slika 2.2. Prikaz rada *Windows* zakački

Zbog toga će globalne varijable u *DLL* datoteci biti u svakom procesu posebno i ne mogu se dijeliti između procesa koji su učitani unutar zakačene *DLL* datoteke. Sve

varijable koje sadrže dijeljene podatke moraju biti postavljene unutar segmenta dijeljenih podataka (*Shared data section).* Sljedeći dijagram pokazuje primjer prijavljivanja zakačke od strane procesa poslužitelja zakačke i učitavanje u adresni prostor aplikacija broj 1 i broj 2.

Sveobuhvatna sustavska zakačka prijavljuje se samo jednom nakon pokretanja *API* funkcije `SetWindowsHookEx()`. Ako nije došlo do pogreške, funkcija vraća ručicu na zakačku. Vraćena vrijednost potrebna je na kraju proizvoljne funkcije zakačke, kada se poziva *API* funkcija `CallNextHookEx()`. Nakon uspješnog poziva *API* funkcije `SetWindowsHookEx()`, operacijski sustav automatski učitava (ali ne nužno i odmah) *DLL* datoteku u sve procese koji udovoljavaju zahtjevima konkretnog filtara zakačke.

```
LRESULT CALLBACK CBTProc( int nCode,
                          WPARAM wParam,
                          LPARAM lParam)
{
  if ( (nCode==HCBT_ACTIVATE) || (nCode==HCBT_SYSCOMMAND) ||
       (nCode==HCBT_QS)       || (nCode==HCBT_CREATEWND))
  {
    //pokrenut je novi proces
    //izvrši obradu i javi aplikaciji
  }

  // Sve poruke moramo proslijediti dalje sa CallNextHookEx.
  return CallNextHookEx(sg_hGetMsgHook, nCode, wParam, lParam);
}
```

Slika 2.3. Primjer funkcije povratnog poziva prijavljene za sveobuhvatnu sustavsku zakačku

Ne postoji drugi način da se rastereti jednom zakačena *DLL* datoteka (učitana je u adresni prostor ciljnog procesa), već da proces koji je prijavio zakačku pozove *API* funkciju `UnhookWindowsHookEx()` ili da se aplikacija koja je bila zakačena ugasi.

Evo nekih prednosti ovog pristupa [2]:

- Mehanizam je podržan u operacijskim sustavima *MS Windows NT*, 2000 i *XP* i *MS Windows* 9x, a vjerojatno će biti podržan i u budućim verzijama.
- Za razliku od *registry* metode, ovaj mehanizam omogućuje učitavanje i rasterećivanje *DLL* datoteke po potrebi upotrebom funkcije `UnhookWindowsHookEx()`.

Iako je ovakav način učitavanja *DLL* datoteke zgodan, prate ga neki nedostaci [2]:

- *Windows* zakačke mogu značajno degradirati svojstva cijelog sustava, zato što povećavaju količinu procesiranja koji sustav mora obaviti za svaku poruku.
- Vrlo je teško uklanjati pogreške kod ovog mehanizma.
- Ova vrsta zakački utječe na procesiranje cijelog sustava i u nekim slučajevima (greške) mora se ponovno pokretati sustav kako bi se ispravila greška.
- Ova metoda neće raditi sa sustavskim servisima (*System Services*).
- Problem zaustavljanja procesa.

2.2. Uporaba funkcije PsSetCreateProcessNotifyRoutine

MS Windows NT, 2000 i *XP* pružaju *API* funkcije, poznate kao *Process Structure Routines* [3] dane na raspolaganje drugim aplikacijama (eksportirane) od strane `NTOSKRNL.DLL`. Jedna od tih funkcija je `PsSetCreateProcessNotifyRoutine()` i ona nudi mogućnost prijave sveobuhvatne sustavske funkcije povratnog poziva, koji operacijski sustav poziva svaki puta kada se pokreće novi proces ili se postojeći uništava.

Ostvarenje ove funkcije moguća je jedino u upravljačkom programu. Funkcija `PsSetCreateProcessNotifyRoutine()` dopušta prijavu i odjavu funkcije povratnog poziva. Prijava i odjava funkcije povratnog poziva najčešće se ostvaruje u upravljačkoj otpremnoj rutini (*Driver's dispatch routine*). To omogućuje dinamičko pokretanje i zaustavljanje sustava detekcije novog procesa korištenjem *IOCTL* poruke.

Korištenjem spomenute funkcije lako se ostvaruje metoda praćenja stvaranja i uništavanja procesa jednostavnim jezgrenim upravljačkim programom, koji će nadzornoj aplikaciji javljati stanja procesa.

```
NTSTATUS PFNameDriverIOControl(IN PDEVICE_OBJECT
DeviceObject,IN PIRP Irp)
{
   PIO_STACK_LOCATION stack;
   UCHAR *buff1, buff2;
   ULONG code;

   // dohvaćanje stoga
   stack = IoGetCurrentIrpStackLocation(Irp);
   // kazaljek na ulazno izlazne spremnike
   in_buffer = out_buffer = Irp->AssociatedIrp.SystemBuffer;
   // dohvaćanje IOCTL poruke
   code = stack->Parameters.DeviceIoControl.IoControlCode;

   switch(code)
   {
      // pokretanje sustava detekcije novog procesa
      case IOCTL_START_MONITORING:
         PsSetCreateProcessNotifyRoutine(PFProcessCallback,
                                         FALSE);
         break;
      // zaustavljanje sustava detekcije novog procesa
      case IOCTL_STOP_MONITORING:
         PsSetCreateProcessNotifyRoutine(PFProcessCallback,
                                         TRUE);
         break;
      ...
   }

  return STATUS_SUCCESS;
}
```

Slika 2.4. Upravljačka otpremna funkcija `PFNameDriverIOControl()`

Detekcija izvođenja procesa i informiranje nadzorne aplikacije o tim događajima obavlja se unutar funkcije povratnog poziva koja je prijavljena funkcijom `PsSetCreateProcessNotifyRoutine()`. Funkcija povratnog poziva ima tri parametra. Prvi parametar `hParentId` je identifikacijski broj procesa koji pokreće novi proces, a `hProcessId` je identifikacijski broj procesa koji se pokreće. Treći parametar je `bCreate` i on je istina ako se funkcija povratnog poziva za novonastali proces. Kada se funkcija povratnog poziva poziva za proces koji se uništava, treći parametar `bCreate` je laž.

```
VOID PFProcessCallback(IN HANDLE  hParentId,
                       IN HANDLE  hProcessId,
                       IN BOOLEAN bCreate)
{
  if(bCreate) {
    //novi process sa identifikacijskim broje hProcessId
    //je pokrenut, zaustavi proces,
    //signaliziraj aplikaciju da je pokrenut novi proces
  }
}
```

Slika 2.5. Funkcija povratnog poziva

Prednosti ove metode su:

- dinamički je moguće učitati i rasteretiti jezgreni upravljački program i
- omogućena je prijava i odjava funkcije povratnog poziva.

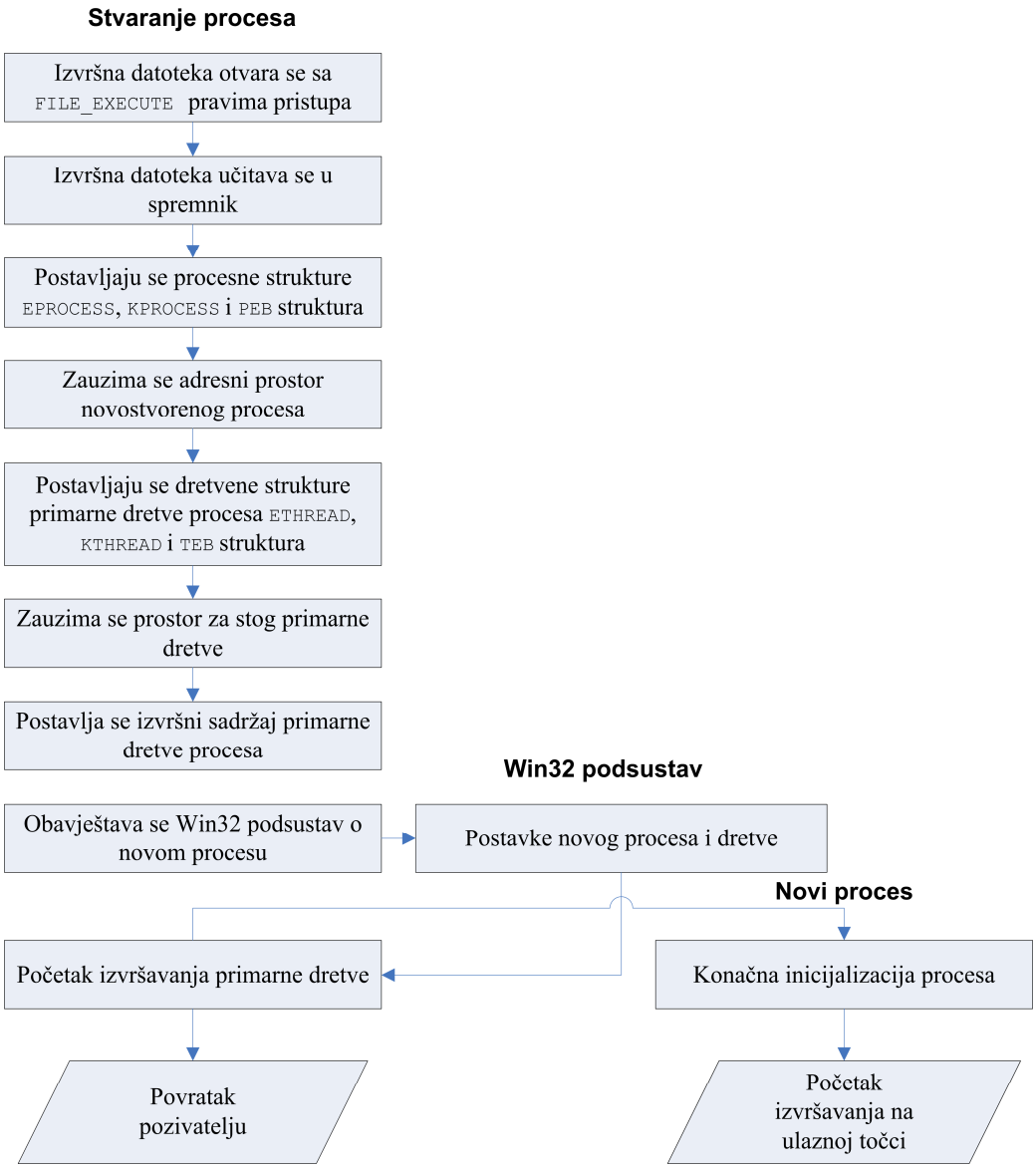
Nedostatak ove metode je isti kao i kod metode detekcije uporabom *DLL* datoteke, a to je problem zaustavljanja procesa.

2.3. Modifikacija SSDT tablice

Jedna od najboljih, ali i najsloženijih metoda za detekciju pokretanja procesa, temelji se na modifikaciji ponašanja sustava stavljanjem zakačke na temeljne jezgrine *API* funkcije.

Ova tehnika ostvaruje se modifikacijom zapisa unutar jezgrene sustavsko-servisne otpremne tablice (*System Service Dispatch Table*). *SSDT* tablica sadrži polje funkcijskih kazaljki koje pokazuju na temeljne *API* pozive. Ovakve modifikacije osiguravaju da zakačena funkcija (funkcija čije će se ponašanje izmijeniti) biva pozvana prije originalne temeljne *API* funkcije. Zakačena funkcija najčešće na kraju poziva originalnu temeljnu *API* funkciju i mijenja izlazne rezultate prije vraćanja rezultata u pozivajući program.

Ostaje odgovoriti na pitanje koju temeljnu *API* funkciju zakačiti? Premda `NtCreateProcess()` zvuči kao očit odgovor, nažalost to nije točno. Moguće je stvoriti proces bez poziva ove funkcije. Tako npr. `CreateProcess()` priprema strukture vezane uz proces bez poziva funkcije `NtCreateProcess()` [4]. Stoga je zakačka funkcije `NtCreateProcess()`neupotrebljiva.



Slika 2.6. Glavne faze stvaranja procesa *Windows NT* sustava

Na dijagramu Slika 2.6 prikazan je tijek stvaranja procesa. Operacijski sustav *MS Windows* temelji se na dretvama. Ne može postojati proces bez dretve. Proces je samo spremnik dretvi. Prilikom stvaranja procesa stvara se primarna dretva u kojoj se izvršava izvršni tekst programa.

Kako bi bilo koji od ovih koraka bio uspješan, svi prethodni koraci moraju završiti uspješno (ne mogu se postavljati procesne strukture bez da se izvršna datoteka učita u spremnik). Prema tome, ako se odluči prekinuti jedan od tih koraka, svi sljedeći koraci isto tako će se srušiti, a stvaranje procesa bit će prekinuto. Razumljivo je da iza svakog od prethodnih koraka stoji pozivanje određenih temeljnih *API* funkcija. S namjerom da pratimo i kontroliramo stvaranje procesa, sve što se mora napraviti je da se zakače one *API* funkcije koje su obvezatne u toku stvaranja novog procesa.

Bolji izbor je da se zakači funkcija `NtCreateSection()`. Kako bi pokrenuli neku aplikaciju, potrebno je učitati izvršnu datoteku u radni spremnik. Ovo je jedino

moguće ostvariti pozivom funkcijom `NtCreateSection()` sa predanim parametrima za preslikavanje izvršne datoteke u radni spremnik (`SEC_IMAGE` atribut) i zahtjevom zaštite stranice spremnika koja omogućuje izvršavanje.

Ukoliko se presretne ovakav poziv `NtCreateSection()` funkcije sigurno se radi o pokretanju novog procesa. U ovom trenutku može se donijeti odluka. U slučaju da ne želimo da se proces pokrene, napravi se da funkcija `NtCreateSection()` vrati `STATUS_ACCESS_DENIED`. Znači, kako bi se ostvarila potpuna kontrola nad stvaranjem procesa, sve što je potrebno je da se zakači temeljna *API* funkcija `NtCreateSection()` na sveobuhvatnoj sustavskoj bazi.

Prednost uporabe ove metode:

- Glavna prednost ove metode je detekcija procesa u stvarnom vremenu, odnosno već u prvim koracima pripreme za pokretanje procesa i ne može se dogoditi da obavijest o kreiranju procesa kasni.
- Ne javlja se problem zaustavljanja procesa, jer se proces provjerava u sustavu sigurnosne politike i prije nego što biva pokrenut. Time se ujedno i štede sredstva koja bi proces zauzeo prilikom svog stvaranja.

Nedostaci:

- Ovu metodu koriste i neki zlonamjerni programi (*Kernel Rootkits*) pa se može dogoditi da neki antivirusni programi ili neki drugi programi za zaštitu detektiraju ovakav program kao zlonamjerni program .

Da je ova metoda vrlo kvalitetna i učinkovita pokazuje da se ona koristi u nekim vrlo kvalitetnim programima za zaštitu od upada kao što su *Kerio Personal Firewall 4*, *DiamondCD Process Guard 2* i *Sebek 2.1.5*.

2.3.1. Modifikacija *SSDT* tablice u praksi [5]

U ovom dijelu prikazan je način zakačivanja funkcije `ZwWriteFile()`. U operacijskom sustavu *MS Windows*, aplikacije za zapisivanje podataka u datoteke koriste *API* funkciju `WriteFile()` eksportiranu od strane `KERNEL32.DLL`. Prilikom poziva funkcije `WriteFile()` poziva se temeljna *API* funkcija `ZwWriteFile()`. Funkcija `ZwWriteFile()` eksportirana je od strane `NTDLL.DLL`. Posao obavljen od strane `ZwWriteFile()` obavlja se u jezgrenom prostoru, tj. u prostoru jezgre operacijskog sustava. Zato ostvarenje `ZwWriteFile()` u `NTDLL.DLL` sadrži samo minimalni izvorni tekst programa koji se prenosi u prostor jezgre operacijskog sustava korištenjem prekida `0x112`.

```
// postavljanje servisnog broja u registar
1- MOV EAX,  0112
// postavljanje kazaljke parametara na stogu
2- LEA EDX, DWORD PTR SS:[ESP+4]
// poziv prekida
3- INT 2E
4- RETN 24
```

Slika 2.7. Funkcija `ZwWriteFile()` u strojnom jeziku

Broj[1] `0x112` u prvoj liniji je servisni broj (*Service number*) za funkciju `ZwWriteFile()` u operacijskom sustavu *Windows XP*. Koristi se kao indeks u jezgrinoj servisno-otpremnoj tablici (*SSDT*) za lociranje adrese funkcije koja sadrži stvarni izvršni tekst programa za zapisivanje podataka u datoteku. Adresa *SSDT*-a može se pronaći unutar servisne opisne tablice (*Service Descriptor Table – SDT*).

Referenciranje na *SDT* moguće je `KeServiceDescriptorTable` simbolom, koji je eksportiran od strane `NTOSKRNL.EXE`.

```
typedef struct ServiceDescriptorTable {
  SDE ServiceDescriptor[4];
} SDT;

typedef struct ServiceDescriptorEntry {
  PDWORD KiServiceTable;
  PDWORD CounterTableBase;
  DWORD  ServiceLimit;
  PBYTE  ArgumentTable;
} SDE;
```

Slika 2.8. Definicija strukture *SDT* simbola

Prvi član strukture, `SDT.ServiceDescriptor[0].KiServiceTable`, sadrži kazaljku na *SSDT* sustavskih funkcija ostvarenih u `NTOSKRNL.EXE`. *SSDT* tablica sadrži polje funkcijskih kazaljki koje pokazuju na temeljne *API* pozive. Broj `ServiceLimit` sadrži broj funkcijskih kazaljki u polju.

Vrijednost tipa `DWORD KiServiceTable[0x112]` je kazaljka na `NtWriteFile()` funkciju, koja sadrži stvarni izvršni tekst programa za zapisivanje u datoteku. Dakle, kako bi se promijenilo ponašanje *API* funkcije `WriteFile()`, treba napisati zamjensku funkciju, učitati je u prostor jezgre operacijskog sustava kao upravljački program i izmijeniti kazaljku `KiServiceTable[0xED]` tako da pokazuje na zamjensku funkciju. Zamjenska funkcija treba zadržati kopiju originalne funkcijske kazaljke (originalnu vrijednost `KiServiceTable[0xED]`), tako da se pozivanjem originalne funkcije može obavi planirani zadatak.

Kako bi se znalo gdje u *SSDT* tablicu treba upisati adresu zamjenske funkcije potrebno je poznavati servisni broj funkcije `ZwWriteFile()`.

```
DWORD * addr=(DWORD *)
   (1+(DWORD)GetProcAddress(GetModuleHandle("ntdll.dll"),
                                          "ZwWriteFile"));
```

Slika 2.9. Izračun servisnog broja funkcije `ZwWriteFile()`

Slika 2.9 prikazuje zanimljiv način kako doći do servisnog broja funkcije `ZwWriteFile()`. Sve *API* funkcije iz `ntdll.dll` datoteke započinju s linijom `MOV EAX, ServiceIndex`, koja se primjenjuje u svim verzijama operacijskih sustava *MS Windows NT*. Ta instrukcija je veličine 5 okteta. `MOV EAX` oznaka instrukcije je veličine jednog okteta, a preostala četiri okteta predstavljaju servisni broj. Prema

[1] Još se naziva „Čarobni broj“ (*Magic number*)

tome, kako bi se doznao servisni broj koji pripada određenoj temeljnoj *API* funkciji, potrebno je pročitati četiri okteta s adrese s odmakom od jednog okteta od početka funkcije.

Izmjenu *SSDT* tablice najlakše je obaviti iz jezgrenog upravljačkog programa. Upravljačkom programu šalje se *IOCTL* poruka s parametrom servisnog broja funkcije koja se želi zakačiti.

```
NTSTATUS DrvDispatch(IN PDEVICE_OBJECT device,IN PIRP Irp)
{
  UCHAR*buff=0; ULONG a,base;

  //dohvaćanje stoga
  PIO_STACK_LOCATION loc=IoGetCurrentIrpStackLocation(Irp);

  //ako je zaprimljena IOCTL poruka 1000
  if(loc->Parameters.DeviceIoControl.IoControlCode==1000)
  {
    buff=(UCHAR*)Irp->AssociatedIrp.SystemBuffer;

    // dohvaćanje indeksa funkcije koja se želi zakačiti
    memmove(&Index,buff,4);
    // adresa funkcije
    a=4*Index+(ULONG)KeServiceDescriptorTable->ServiceTable;

    // mapiranje dijela SSDT tablice gdje će se zapisativati
    // adresa nove funkcije
    base = (ULONG) MmMapIoSpace(
                   MmGetPhysicalAddress( (void*)a), 4, 0);
    a = (ULONG)&Proxy;

    _asm
    {
      // spremanje adrese orginalne funkcije u
      // globalnu varijablu RealCallee
      mov eax,base
      mov ebx,dword ptr[eax]
      mov RealCallee,ebx
      // spremanje adrese nove funkcije u
      // SSDT tablicu
      mov ebx,a
      mov dword ptr[eax],ebx
    }

    MmUnmapIoSpace(base, 4);

    memmove(&a, &buff[4], 4);
    output = (char*) MmMapIoSpace(
             MmGetPhysicalAddress( (void*)a), 256, 0);
  }

  Irp->IoStatus.Status=0;
  IoCompleteRequest(Irp,IO_NO_INCREMENT);
  return 0;
}
```

Slika 2.10. Otpremna rutina upravljačkog programa s primjerom modifikacije *SSDT* tablice

Prvo se mapira spremnik za razmjenu u adresni prostor jezgre funkcijom `MmMapIoSpace()`, a zatim se zapiše adresa zamjenske funkcije u *SSDT* tablicu, odnosno zapiše se nakon što se pohrani originalna adresa funkcije u globalnu varijablu `RealCallee`. Kako bi se izmijenio odgovarajući zapis *SSDT* tablice, potrebno je mapirati odredišnu adresu funkcijom `MmMapIoSpace()`. Zašto je sve ovo potrebno, kada već znamo adresu *SSDT* tablice? Problem je u tome što je moguće da se *SSDT* tablica nalazi u spremniku nad kojim imamo pristup samo za čitanje (*Read-only memory*). Stoga se mora provjeriti da li je trenutni pristup za pisanje stranice spremnika u kojoj se nalazi adresa na koju želimo zapisati, ako nije tako, mora se promijeniti pristup toj stranici spremnika. Ovo se obavlja jednostavnim mapiranjem odredišne adrese funkcijom `MmMapIoSpace()`.

Pored modifikacije *SSDT* tablice iz prostora jezgre operacijskog sustava pomoću jezgrenog upravljačkog programa, *SSDT* moguće je promijeniti i iz korisničkog prostora korištenjem metode direktnog zapisa u spremnik jezgre operacijskog sustava korištenjem `\device\physicalmemorij`. Više o ovoj metodi može se pronaći u dokumentu [5].

Popis svih „čarobnih brojeva“ odnosno sustavske tablice poziva (*System Call Table*) za razne operacijske sustave *MS Windows* nalazi se u [6].

2.3.2. Potraga za preusmjerenim temeljnim jezgrenim funkcijama

Temeljne *API* funkcije mogu biti zakačene na dva načina:

1. izmjenom adrese poziva funkcije u *SSDT* tablici ili
2. postavljanjem instrukcije bezuvjetnog skoka na početak funkcije.

Što se tiče prve metode, vrlo je lagano otkriti preusmjerenu temeljnu *API* funkciju. Nakon što se dozna adresa *SSDT* tablice, potrebno je doznati lokaciju spremnika gdje je smještena *DLL* datoteka `NTDLL.DLL`. Prolaženjem kroz svaki element *SSDT* tablice, gleda se da li je adresa temeljne funkcije izvan područja spremnika u kojem je učitana `NTDLL.DLL` datoteka. Funkcijom `GetModuleInformation()` dobiva se informacija na kojoj je loakciji spremnika smještena *DLL* datoteka. Područje spremnika na kojem je smještena `NTDLL.DLL` datoteka dobije se tako da se početnoj lokaciji spremnika na kojem je smještena pridoda veličina datoteke.

Kod druge metode, potrebno je pregledati sve elemente *SSDT* tablice i provjeriti da li se na adresi temeljne *API* funkcije nalazi instrukcija bezuvjetnog skoka. Ukoliko se nalazi instrukcija bezuvjetnog skoka na lokaciji spremnika gdje započinje temeljna *API* funkcija, to ukazuje da je ta funkcija zakačena, odnosno preusmjerena.

3. Asinkrona procesna komunikacija

Asinkrona komunikacija između jezgrenog upravljačkog programa i nadzorne aplikacije temelji se na funkcijama `KeInitializeApc()` i `KeInsertQueueApc()`. Navedene funkcije *Microsoft* nije dokumentirao, pa njihov prototip ne postoji u *DDK* (*Driver Development Kit*) [7] zaglavljima.

```
NTKERNELAPI
VOID
KeInitializeApc (
  IN PRKAPC Apc,
  IN PKTHREAD Thread,
  IN KAPC_ENVIRONMENT Environment,  // Albert Almeida u [9]
                                    // spominje vrijednosti
                                    // Original,
                                    // Attached ili Current
  IN PKKERNEL_ROUTINE KernelRoutine,
  IN PKRUNDOWN_ROUTINE RundownRoutine OPTIONAL,
  IN PKNORMAL_ROUTINE NormalRoutine OPTIONAL,
  IN KPROCESSOR_MODE ApcMode,
  IN PVOID NormalContext
);

BOOLEAN
KeInsertQueueApc(
   PKAPC Apc,
   PVOID SystemArgument1,
   PVOID SystemArgument2,
   UCHAR mode);  // Albert Almeida u [9]
                 // spominje ovaj parametar kao
                 // IN KPRIORITY Increment
                 // (Kernel, User, ili Maximum (0,1,2))
```

Slika 3.1. Prototip nedokumentiranih funkcija `KeInitializeApc()` i `KeInsertQueueApc()` za asinkronu komunikaciju između jezgrenog upravljačkog programa i nadzorne aplikacije

Zadnji parametar funkcije `KeInsertQueueApc()` moguće da je uvećanje prioriteta dretve koja vrši asinkronu komunikaciju [9]. Ovaj podatak ne može se provjeriti, pa je tijekom ostvarenja praktičnog rada vrijednost ovog parametra 0.

Kada se dretva opredijeli upravljačkom programu, izvorni tekst programa unutar funkcije povratnog poziva upravljačkog programa koja služi za informiranje o novim procesima poziva funkciju `QueueApcToClientThread()`, kako bi postavila postavke za asinkronu komunikaciju koja će pozivati funkciju `ApcCallback()` u nadzornoj aplikaciji.

`QueueApcToClientThread()` zauzima i inicijalizira strukturu `KAPC`, u spremnik koji nije podijeljen na stranice. `KAPC` struktura pokazuje na jezgrenu funkciju (čiji je jedini zadatak izbrisati *APC* objekt) i funkciju nadzorne aplikacije (čija je adresa predana u postupku prijavljivanja dretve nadzorne aplikacije). Može se dodati i funkcija koja će se izvesti po završetku asinkrone komunikacije, ali nije potrebno jer

će *APC* objekt obrisati operacijski sustav u slučaju da se uništi dretva s redom čekanja asinkrone komunikacije. Funkcijom `KeInsertQueueApc()` *APC* objekt stavlja se u red čekanja dretve nadzorne aplikacije.

```
BOOLEAN QueueApcToClientThread(PVOID SystemArgument1,
                               PVOID SystemArgument2)
{
  BOOLEAN retval = TRUE;
  struct _KAPC    *pApc;
  // zauzimanje prostora za APC objekt
  pApc = ExAllocatePool( NonPagedPool, sizeof(struct _KAPC));

  if(NULL == pApc) {
    ASSERT(pApc);
    retval = FALSE;
  }
  else {
    // inicijalizacija APC objekta
    KeInitializeApc(pApc, g_pKThreadClient,
           OriginalApcEnvironment,
           &KernelApcRoutine,      // jezgrena APC funckija
           NULL,                   // završna APC funkcija
           g_KMessageInfo.pFn,     // korisnička APC funkcija
           UserMode,               // Apc način
           (PVOID) NULL);

    //ubaci APC objekt u red čekanja dretve nadzorne aplikacije
    retval = KeInsertQueueApc(pApc, SystemArgument1,
                              SystemArgument2, 0);
  }

  return retval;
}
```

Slika 3.2. Inicijalizacija asinkrone komunikacije

Nakon inicijalizacije, *APC* objekt ubačen je u red čekanja, sa parametrima (`SystemArgument1` i `SystemArgument2`) koji se trebaju predati funkciji nadzorne aplikacije. Objekt `KTHREAD` (globalna varijabla `pKThreadClient` pokazuje na njega) postavljen je tijekom postupka prijave procesa u funkciji `DeviceControl()`.

```
VOID KernelApcRoutine( IN struct _KAPC *Apc,
         IN OUT PKNORMAL_ROUTINE *NormalRoutine,
         IN OUT PVOID *NormalContext,
         IN OUT PVOID *SystemArgument1,
         IN OUT PVOID *SystemArgument2 )
{
    ExFreePool(Apc);
}
```

Slika 3.3. Jezgrena funkcija asinkrone komunikacije

Može se primijetiti da se nigdje ne vodi računa o sinhronizaciji podacima niti o redu čekanja. U osnovi, radi se sa nanovo zauzetim objektom, te se prepušta operacijskom sustavu da se brine o detaljima.

APC objekti korisničkog načina rada dostavljaju se samo dretvi u pripravnom stanju. Pretplaćena dretva, čiji je KTHREAD objekt korišten za inicijalizaciju asinkrone komunikacije, čeka na događaj u korisničkom načinu rada čineći ga spremnim. Doduše, događaj ne mora biti signaliziran kako bi APC objekt bio isporučen. Kada operacijski sustav omogući dretvi izvršavanje, provjerava dretveni red čekanja za asinkronu komunikaciju (u stvarnosti postoje dva reda čekanja za asinkronu komunikaciju [9]), i ako u redu postoje objekti korisničkog načina rada asinkrone komunikacije koje treba dostaviti, a i dretva je spremna, operacijski sustav će za svaki PKAPC u redu čekanja pozvati korisničku funkciju, a zatim i jezgrenu funkciju (ako postoji), po *FIFO* (*First in first out*) principu.

```
function ApcThreadProc(that: TMonitor): Integer;
var
  threadexitcode: Cardinal;
  WaitResult: DWord;
begin
  threadexitcode := 0; Result := 0;

  // ako to nije dretva koja je izvršila pretplatu
  if false = that.KernelSubscribe then
      threadexitcode := 1
  else begin
    // priprema dretve da bude spremna tako da
    // korisnički apc stavljeni u red čekanja
    // u jezgri mogu biti dostavljeni ...
    that.fEvent := CreateEvent(nil,  // sigurnosni atributi
        False,  // ručno poništavanje
        False,  // početno stanje nesignaliziran
        nil);   // događaj bez naziva

    // provjeri havtaljku...
    if 0 = that.fEvent then begin
        that.fLastError := GetLastError();
        threadexitcode := 2;
    end
    else begin
      repeat
        WaitResult := WaitForSingleObjectEx(that.fEvent,
                    INFINITE, TRUE);
      until WaitResult = WAIT_OBJECT_0;
      // Funkcija će vratiti WAIT_IO_COMPLETION kada je
      // apc dostavljen. Funkcija vraća WAIT_OBJECT_0 kada
      // je događaj signaliziran, tako da se
      // funkcijom SetEvent može ubiti dretva.
    end;
  end;
  EndThread(threadexitcode);
end;
```

Slika 3.4. Prikaz prijavljivanja i čekanja na događaj unutar dretvene korisničke funkcije

Može se primijetiti da je jezgrena funkcija vjerojatno opcionalna, ali ne daje mogućnost da se oslobodi KAPC objekt. Jednom kada je *APC* objekt dostavljen, dretva nadzorne aplikacije nastavlja gdje je stala, nakon poziva funkcije

`WaitForSingleObjectEx()`, koja vraća `WAIT_IO_COMPLETION`, te se opet vraća u stanje čekanja.

Upravljački program komunicira stavljanjem APC objekata u red čekanje pretplaćene dretve. Asinkrona komunikacija uzrokuje izvršavanje funkcije `ApcCallback()`, koja će iz zaprimljenih parametara ustvrditi koji se proces pokreće.

4. Manipulacija procesima

Svaki proces u operacijskom sustavu *MS Windows* ima svoj identifikacijski broj *PID* (*Process Identifier*). Kada se proces kreira funkcijom `CreateProcess()`, funkcija vrati ručicu procesa te ručicu primarne pripadajuće dretve. Funkcija `CreateProcess()` vraća i identifikacijski broj procesa *PID* te identifikacijski broj dretve *TID* (*Thread Identifier*). Unutar procesa moguće je koristiti funkciju `GetCurrentProcessId()` kako bi se doznao identifikacijski broj vlastitog procesa. Identifikacijski broj procesa valjan je tijekom života procesa sve dok isti ne bude uništen. Uporabom funkcije `Process32First()` može se dobiti identifikacijski broj procesa roditelja.

Jednom kada se zna identifikacijski broj procesa, funkcijom `OpenProcess()` dolazi se do ručice procesa. Funkcija `OpenProcess()` omogućuje da se specificiraju prava pristupa ručice i da li je ručicu moguće naslijediti. Uporabom funkcije `GetCurrentProcess()` proces može doći do pseudo ručice koja pokazuje na vlastiti procesni objekt. Ova pseudo ručica važeća je jedino za pozivajući proces. Ovakvu ručicu nemoguće je naslijediti ili duplicirati za uporabu u drugim procesima. Kako bi se došlo do stvarne ručice, potrebno je koristiti funkciju `DuplicateHandle()`.

Za dobivanje više mogućnosti nad pojedinom ručicom poželjno je postaviti `SeDebugPrivilege` ovlasti na trenutni proces. `SeDebugPrivilege` ovlasti daju trenutnom procesu jednake ovlasti kao što ima *Debuger*.

```
function SetDebugPrivilege(sPrivilege: String): Boolean;
var
  hToken: THandle;
  TokenPriv,
  PrevTokenPriv: TOKEN_PRIVILEGES;
  ReturnLength: Cardinal;
begin
  Result := False;
  ReturnLength := 0;

  // dohvati oznaku (Token) procesa
  if OpenProcessToken(GetCurrentProcess(),
    TOKEN_ADJUST_PRIVILEGES or TOKEN_QUERY, hToken) then begin
    try
      // dohvati LUID (Locally Unique Identifier)
      if LookupPrivilegeValue(nil, PChar(sPrivilege),
        TokenPriv.Privileges[0].Luid) then begin
        // postavljanje jedne ovlasti
        TokenPriv.PrivilegeCount := 1;
        TokenPriv.Privileges[0].Attributes :=
                                      SE_PRIVILEGE_ENABLED;
        PrevTokenPriv := TokenPriv;

        // omogući ovlasti
        AdjustTokenPrivileges(hToken, False, TokenPriv,
          SizeOf(PrevTokenPriv), PrevTokenPriv, ReturnLength);
      end;
```

```
    finally
      // zatvori ručicu
      CloseHandle(hToken);
    end;
  end;
  // da li je sve uredu
  Result := GetLastError = ERROR_SUCCESS;
end;
```

Slika 4.1. Funkcija za postavljanje *Debug* ovlasti na trenutni proces

4.1. Zaustavljanje procesa

Zasad nijedan od operacijskih sustava *MS Windows* nema dokumentiranu *API* funkciju za zaustavljanje procesa, kao što postoji funkcija `SuspendThread()` za zaustavljanje dretvi.

Prolazeći kroz listu dretvi koje pripadaju određenom procesu, svaku dretvu zaustavljamo funkcijom `SuspendThread()`. Dohvaćanje liste dretvi željenog procesa moguće je korištenjem `CreateToolhelp32Snapshot()` funkcije [10]. Nažalost, ovakva metoda ima tri problema:

1. Pozivanje funkcije `SuspendThread()` za dretvu koja posjeduje sinkronizacijski objekt, muteks ili kritični odsječak, može dovesti do potpunog zastoja ukoliko neka dretva pokušava održati sinkronizaciju sa dretvom koja je zaustavljena.
2. Nisu svi programi predviđeni da budu zaustavljeni, pogotovo oni više dretveni, tako da se neki programi mogu ponašati neprirodno nakon ponovnog pokretanja.
3. Može se dogoditi da nakon dohvaćanja liste dretvi procesa, proces kreira novu dretvu koja nije u listi, i koja onda neće biti zaustavljena.

```
BOOL PauseProcess(DWORD dwOwnerPID)
{
  HANDLE        hThreadSnap = NULL;
  BOOL          bRet        = FALSE;
  THREADENTRY32 te32        = {0};

  // Napravimo snimak svih dretvi u sustavu
  hThreadSnap = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
  if (hThreadSnap == INVALID_HANDLE_VALUE) return (FALSE);

  te32.dwSize = sizeof(THREADENTRY32);
  // U snimku tražimo dretve koje pripadaju traženom procesu.
  if (Thread32First(hThreadSnap, &te32)) {
    do {
      // ako dretva pripada traženom procesu
      if (te32.th32OwnerProcessID == dwOwnerPID) {
        HANDLE hThread = OpenThread(THREAD_SUSPEND_RESUME,
                                    FALSE, te32.th32ThreadID);
        // zaustavi dretvu
        SuspendThread(hThread);
        CloseHandle(hThread);
      }
```

```
      }
      while (Thread32Next(hThreadSnap, &te32));
      bRet = TRUE;
    }
    else bRet = FALSE;

    // zatvori ručicu
    CloseHandle (hThreadSnap);
    return (bRet);
}
```

Slika 4.2. Funkcija `PauseProcess()` za zaustavljanje procesa zadanog identifikacijskim brojem

Analogno `PauseProcess()` funkciji može se definirati i `ResumeProcess()` funkcija gdje se umjesto `SuspendThread()` koristi funkcija `ResumeThread()`.

U operacijskom sustavu *MS Windows XP* postoje nedokumentirane funkcije `NtSuspendProcess()` i `NtResumeProcess()` eksportirane od strane `NTDLL.DLL` datoteke. Nažalost, ne zna se da li funkcija `NtSuspendProcess()` ima iste probleme kao prethodna metoda. Vrlo je vjerojatno da prva dva problema vrijede.

4.2. Uništavanje procesa

Uništavanje procesa ima sljedeće posljedice:

- preostale dretve u procesu označene su za uništavanje,
- svi zauzeti resursi su oslobođeni,
- svi jezgreni objekti su zatvoreni,
- izvršni tekst procesa briše se iz spremnika,
- postavlja izlazni kod procesa i
- signalizira se procesni objekt.

Kada se otvorene ručice prema jezgrenim objektima automatski zatvore kad se proces uništi, objekti i dalje postoje dok se ne zatvore sve ručice koje pokazuju na njih. Stoga će objekt i dalje biti važeći nakon što se uništi proces koji ga je koristio, ako postoji još neki proces koji ima otvorenu ručicu prema njemu.

Funkcija `TerminateProcess()` koristi se za bezuvjetan izlazak iz procesa. Funkcija `TerminateProcess()` započinje uništavanje i ne čeka da postupak završi već odmah vraća vrijednost. Ovako se zaustavlja izvršavanje svih dretvi u procesu i zahtijeva se otkazivanje svih I/O operacija u tijeku. Uništeni proces ne može završiti sve dok se sve I/O operacije u tijeku ne završe ili ne prekinu. Proces ne može spriječiti vlastito uništenje.

Ako se proces uništi funkcijom `ExitProcess()`, operacijski sustav poziva ulaznu funkciju svake priključene *DLL* datoteke sa vrijednošću koja pokazuje da se proces odvaja od *DLL* datoteke. *DLL* datoteke nisu obaviještene o odvajanju procesa kada se proces uništi funkcijom `TerminateProcess()`.

Ako je proces uništen funkcijom `TerminateProcess()`, sve dretve procesa uništavaju se trenutno bez mogućnosti za pokretanjem dodatnog izvršnog teksta

programa. To znači da dretva ne izvrši izvršni tekst programa koji se nalazi u dijelu koji izvršava nakon uništenja dretve, niti se ne obavijeste priključene *DLL* datoteke o odvajanju procesa.

```
procedure TerminateProcessByID(PID: DWord);
var
  hProc: THandle;
begin
  // dohvatimo ručicu procesa zadanog identifikacijskim brojem
  // s pristupnim pravima za uništavanje procesa
  hProc := OpenProcess(PROCESS_TERMINATE, False, PID);

  if hProc <> 0 then begin
    // uništavanje procesa
    TerminateProcess(hProc, 0);

    // zatvaranje ručice
    CloseHandle(hProc);
  end;
end;
```

Slika 4.3. Funkcija za uništavanje procesa zadanog identifikacijskim brojem

Kada operacijski sustav uništava proces, on ne uništava ni jedan proces koji je stvoren od procesa roditelja. Uništavanje procesa ne garantira informiranje `WH_CBT` porukom zakačene procedure.

4.3. Promjena prioriteta procesa

Funkcijom `SetPriorityClass()` postavlja se prioritetni razred za određeni proces. Ova vrijednost zajedno sa vrijednostima prioriteta pojedine dretve procesa određuje bazni prioritet svake dretve.

Svaka dretva ima bazni prioritet određen vrijednošću prioriteta dretve i prioritetne klase njezinog procesa. Operacijski sustav koristi bazni prioritet svih izvršnih dretvi kako bi se odredila dretva koja sljedeća dobiva djelić procesorskog vremena. Funkcijom `SetThreadPriority()` omogućuje se postavljanje baznog prioriteta dretve koji se relativno odnosi na prioritetni razred procesa.

Kao prioritetni razred procesa podrazumijeva se `NORMAL_PRIORITY_CLASS`. Uporabom `CreateProcess()` funkcije može se odrediti klasa prioriteta procesa djeteta kada se stvara. Ukoliko je klasa prioriteta pozivajućeg procesa `IDLE_PRIORITY_CLASS` ili `BELOW_NORMAL_PRIORITY_CLASS`, novi proces će naslijediti taj razred.

Procesi koji nadgledaju sustav, kao što su čuvari zaslona ili aplikacije koje periodično osvježavaju ekran, trebali bi koristiti prioritetni razred `IDLE_PRIORITY_CLASS`. Ovo sprječava uplitanje dretve ovog procesa, koji nema visoki razred prioriteta, s dretvama visokog prioriteta.

`HIGH_PRIORITY_CLASS` razred prioriteta treba koristiti oprezno. Ako se dretva pokreće s najvećim prioritetom duže vrijeme, preostale dretve u sustavu neće dobiti

procesorsko vrijeme. Ukoliko je prioritet nekoliko dretvi postavljen na visoku razinu istovremeno, dretve tada gube svoju efikasnost. Visoki razred prioriteta trebao bi biti rezerviran za dretve koje moraju odgovarati na vremenski kritične događaje. U slučaju da aplikacija provodi jedan zadatak koji zahtijeva visoki razred prioriteta dok su preostali zadaci normalnog prioriteta, korištenjem funkcije `SetPriorityClass()` može se privremeno podići prioritetni razred aplikacije. Nakon što se obavi vremenski kritičan zadatak, prioritetni razred može se vratiti na početno stanje. Druga strategija stvara proces visokog prioriteta čije su dretve blokirane većinu vremena, a po potrebi probudi dretve kada je potrebno obaviti kritični zadatak, odnosno, dretve visokog prioriteta koriste se samo u kratkim periodima, i samo kada je potrebno obaviti vremenski kritičan zadatak.

Gotovo nikad se ne bi trebao upotrebljavati `REALTIME_PRIORITY_CLASS` prioritetni razred zato što se prekidaju sustavske dretve koje upravljaju mišem, tipkovnicom i pozadinskim zapisivanjem na disk. Ovaj razred može biti pogodan za aplikacije koje komuniciraju direktno sa hardverom ili koje izvode kratke zadatke koji moraju imati minimalan broj prekida.

```
function SetProcessPriorityByID(PID: DWord; Priority: DWord):
Boolean;
var
  hProc: THandle;
begin
  Result := False;
  // dohvatimo ručicu procesa zadanog identifikacijskim brojem
  // sa scim pristupnim pravima izmjene procesa
  hProc := OpenProcess(PROCESS_ALL_ACCESS, False, PID);

  if hProc <> 0 then begin
    // postavljanje zadanog prioritetnog razreda
    Result := SetPriorityClass(hProc, Priority);

    // zatvaranje ručice
    CloseHandle(hProc);
  end;
end;
```

Slika 4.4. Funkcija za promjenu razreda prioriteta procesa zadanog identifikacijskim brojem

4.4. Čitanje spremnika procesa

Programi za ispravljanje pogrešaka otvaraju proces s pravima pristupa `PROCESS_VM_READ` i `PROCESS_VM_WRITE`. Korištenje ovih prava omogućuje programima za ispravljanje pogrešaka da čitaju i zapisuju virtualni spremnik procesa korištenjem funkcija `ReadProcessMemory()` i `WriteProcessMemory()`.

Funkcija `ReadProcessMemory()` kopira podatke sa specifičnog adresnog prostora određenog procesa u specificirani međuspremnik trenutnog procesa. Svaki proces koji ima pravo `PROCESS_VM_READ` nad ručicom procesa može pozvati ovu funkciju. Cijelo područje koje se želi pročitati mora biti dostupno, u suprotnom funkcija ne uspijeva.

Prije bilo kakvog čitanja ili pisanja funkcija `ReadProcessMemory()` i `WriteProcessMemory()` poželjno je postaviti `SeDebugPrivilege` ovlasti na trenutni proces, kako bi funkcije bolje radile.

4.4.1. Naredbeni redak i izvršna datoteka procesa

U adresnom prostoru svakog procesa nalazi se struktura tipa `RTL_USER_PROCESS_PARAMETERS`, koja sadrži dodatne informacije o procesu. Tako se iz ove strukture može doznati izvršna datoteka procesa (*ImagePathName*), te naredbeni redak (*CommandLine*).

```
typedef struct _RTL_USER_PROCESS_PARAMETERS {
  ULONG                   MaximumLength;
  ULONG                   Length;
  ULONG                   Flags;
  ULONG                   DebugFlags;
  PVOID                   ConsoleHandle;
  ULONG                   ConsoleFlags;
  HANDLE                  StdInputHandle;
  HANDLE                  StdOutputHandle;
  HANDLE                  StdErrorHandle;
  UNICODE_STRING          CurrentDirectoryPath;
  HANDLE                  CurrentDirectoryHandle;
  UNICODE_STRING          DllPath;

  // kazaljka na stazu izvršne datoteke procesa
  UNICODE_STRING          ImagePathName;

  // kazaljka na naredbeni redak procesa
  UNICODE_STRING          CommandLine;

  PVOID                   Environment;
  ULONG                   StartingPositionLeft;
  ULONG                   StartingPositionTop;
  ULONG                   Width;
  ULONG                   Height;
  ULONG                   CharWidth;
  ULONG                   CharHeight;
  ULONG                   ConsoleTextAttributes;
  ULONG                   WindowFlags;
  ULONG                   ShowWindowFlags;
  UNICODE_STRING          WindowTitle;
  UNICODE_STRING          DesktopName;
  UNICODE_STRING          ShellInfo;
  UNICODE_STRING          RuntimeData;
  RTL_DRIVE_LETTER_CURDIR DLCurrentDirectory[0x20];
} RTL_USER_PROCESS_PARAMETERS, *PRTL_USER_PROCESS_PARAMETERS;
```

Slika 4.5. `RTL_USER_PROCESS_PARAMETERS` struktura

Problem je u tome što ne znamo gdje se `RTL_USER_PROCESS_PARAMETERS` struktura nalazi. Adresa `RTL_USER_PROCESS_PARAMETERS` strukture može se pročitati iz *PEB* (*Process Environment Block*) bloka. Unutar *PEB* strukture postoji kazaljka na `RTL_USER_PROCESS_PARAMETERS` strukturu (*ProcessParameters*).

```
typedef struct _PEB {
  BOOLEAN                 InheritedAddressSpace;
  BOOLEAN                 ReadImageFileExecOptions;
  BOOLEAN                 BeingDebugged;
  BOOLEAN                 Spare;
  HANDLE                  Mutant;
  PVOID                   ImageBaseAddress;
  PPEB_LDR_DATA           LoaderData;

  // kazaljka na RTL_USER_PROCESS_PARAMETERS
  // strukturu u kojoj se nalaze podaci o
  // izvršnoj datoteci procesa i
  // narebenom retku
  PRTL_USER_PROCESS_PARAMETERS ProcessParameters;

  PVOID                   SubSystemData;
  PVOID                   ProcessHeap;
  PVOID                   FastPebLock;
  PPEBLOCKROUTINE         FastPebLockRoutine;
  PPEBLOCKROUTINE         FastPebUnlockRoutine;
  ULONG                   EnvironmentUpdateCount;
  PPVOID                  KernelCallbackTable;
  PVOID                   EventLogSection;
  PVOID                   EventLog;
  PPEB_FREE_BLOCK         FreeList;
  ULONG                   TlsExpansionCounter;
  PVOID                   TlsBitmap;
  ULONG                   TlsBitmapBits[0x2];
  PVOID                   ReadOnlySharedMemoryBase;
  PVOID                   ReadOnlySharedMemoryHeap;
  PPVOID                  ReadOnlyStaticServerData;
  PVOID                   AnsiCodePageData;
  PVOID                   OemCodePageData;
  PVOID                   UnicodeCaseTableData;
  ULONG                   NumberOfProcessors;
  ULONG                   NtGlobalFlag;
  BYTE                    Spare2[0x4];
  LARGE_INTEGER           CriticalSectionTimeout;
  ULONG                   HeapSegmentReserve;
  ULONG                   HeapSegmentCommit;
  ULONG                   HeapDeCommitTotalFreeThreshold;
  ULONG                   HeapDeCommitFreeBlockThreshold;
  ULONG                   NumberOfHeaps;
  ULONG                   MaximumNumberOfHeaps;
  PPVOID                  *ProcessHeaps;
  PVOID                   GdiSharedHandleTable;
  PVOID                   ProcessStarterHelper;
  PVOID                   GdiDCAttributeList;
  PVOID                   LoaderLock;
  ULONG                   OSMajorVersion;
  ULONG                   OSMinorVersion;
  ULONG                   OSBuildNumber;
  ULONG                   OSPlatformId;
  ULONG                   ImageSubSystem;
  ULONG                   ImageSubSystemMajorVersion;
  ULONG                   ImageSubSystemMinorVersion;
  ULONG                   GdiHandleBuffer[0x22];
  ULONG                   PostProcessInitRoutine;
```

```
  ULONG                   TlsExpansionBitmap;
  BYTE                    TlsExpansionBitmapBits[0x80];
  ULONG                   SessionId;
} PEB, *PPEB;
```

Slika 4.6. *PEB* struktura

Problem je doći do adrese *PEB* bloka. U operacijskim sustavima *MS Windows XP* i *MS Windows* 2000 dostupna je funkcija `NtQueryInformationProcess()`, koja omogućuje dohvaćanje informacija o specifičnom procesu, između ostalog i adresu PEB bloka. Kada se funkcija `NtQueryInformationProcess()` pozove sa parametrom `ProcessBasicInformation`, međuspremnik `ProcessInformation` koji pokazuje na strukturu tipa `PROCESS_BASIC_INFORMATION` treba biti dovoljno velik da prihvati cijelu strukturu. Unutar navedene strukture postoji kazaljka koja pokazuje na *PEB* blok (*PebBaseAddress*).

```
typedef struct _PROCESS_BASIC_INFORMATION {
    PVOID     Reserved1;

    // adresa PEB bloka
    PPEB      PebBaseAddress;

    PVOID     Reserved2[2];
    ULONG_PTR UniqueProcessId;
    PVOID     Reserved3;
} PROCESS_BASIC_INFORMATION;
```

Slika 4.7. `_PROCESS_BASIC_INFORMATION` struktura

Kada je poznata adresa *PEB* bloka odnosno `RTL_USER_PROCESS_PARAMETERS` strukture, poznat je i spremnik koji sadrži stazu izvršne datoteke procesa.

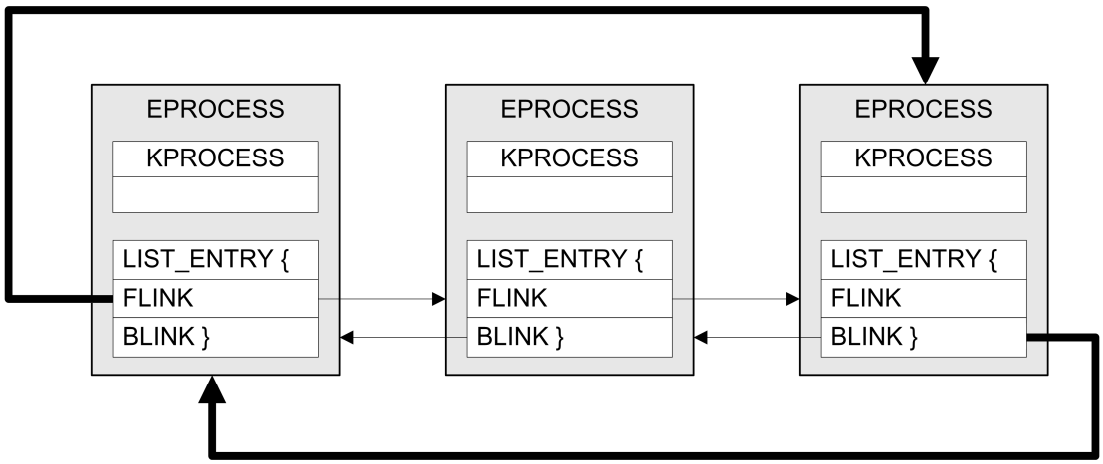
Vlastitim istraživanjem dijela spremnika procesa ustanovljeno je da kazaljke `ImagePathName i CommandLine` ustvari nisu kazaljke, već predstavljaju posmak u odnosu na početak `RTL_USER_PROCESS_PARAMETERS` strukture. Ovo je slučaj kada se proces pauzira odmah poslije svojeg kreiranja, kada postupak stvaranja procesa nije proveden do kraja (faze stvaranja procesa objašnjene su u poglavlju 2.3). Funkcija koja vraća naredbeni redak i izvršnu datoteku procesa zadanog identifikacijskim brojem prikazana je u Dodatku A.

4.5. Detekcija skrivenih procesa

Kako bi detektirali skrivene procese, treba znati kako je moguće sakriti procese. Proces se može sakriti na više načina:

1. Postavi se zakačka na temeljnu *API* funkciju `NtQuerySystemInformation()` koja s određenim parametrom vraća listu aktivnih procesa. Nakon što se pozove originalna temeljna *API* funkcija, u rezultatima se potraži proces koji se želi sakriti te se izmijene povratni rezultati. Primjer zamjenske funkcije prikazan je u [11].

2. Proces se izbriše iz sustava liste aktivnih procesa. Svaki proces ima `EPROCESS` blok koji ga predstavlja. Locira se `EPROCESS` blok procesa koji se želi sakriti. U svakom `EPROCESS` bloku je dvostruko povezana lista, koja pokazuje na sljedeći odnosno prethodni `EPROCESS` blok. Sada je potrebno promijeniti proces koji se nalazi prije i poslije u listi (Slika 4.8).



Slika 4.8. Premošćivanje `EPROCESS` bloka koji se želi sakriti

3. Izbriše se zapis o procesu koji se želi sakriti iz tablice `PspCidTable`. `PspCidTable` sadrži popis svih aktivnih procesa i dretvi. Struktura `PspCidTable` nije globalna varijabla, tj. nije dostupna ostalim procesima, pa je vrlo teško doći do njezine adrese. Postoje dva načina na koja je to moguće. Jedan od njih je skeniranje funkcije `PsLookupProcessByProcessId()` u potrazi za adresom tablice, a drugi je objasnio Edgar Barbosa u svom tekstu [12].

Ideja otkrivanja skrivenih procesa je da se pomoću raznih metoda doznaju liste aktivnih procesa te se onda dobivene liste usporede. Razlike su skriveni procesi.

Funkcija koja se koristi za dobivanje liste aktivnih procesa je `CreateToolhelp32Snapshot()`. Lista dobivena ovom funkcijom koristi se kao referentna lista.

```
BOOL GetProcessList(cList aList)
{
  HANDLE hPsSnap;
  HANDLE hProcess;
  PROCESSENTRY32 pe32;
  DWORD dwPriorityClass;

  // Napravimo snimak svih procesa u sustavu
  hPsSnap = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );

  if( hPsSnap == INVALID_HANDLE_VALUE )
    return( FALSE );
```

```
  pe32.dwSize = sizeof( PROCESSENTRY32 );

  // Dohvati informaciju o prvom procesu
  if( !Process32First( hPsSnap, &pe32 ) )
  {
    // zatvori ručicu
    CloseHandle( hPsSnap );
    return( FALSE );
  }

  // Proći kroz snimak procesa
  do {
    // ubaci identifikacijski broj u listu
    aList.Add( pe32.th32ProcessID );
  } while( Process32Next( hPsSnap, &pe32 ) );

  // zatvori ručicu
  CloseHandle( hPsSnap );
  return( TRUE );
}
```

Slika 4.9. Funkcija za dohvaćanje liste aktivnih procesa uporabom funkcije `CreateToolHelp32Snapshot()`

Svaka metoda skrivanja procesa, skriva se od jednog ili više načina dohvaćanja liste procesa.

Popis aktivnih procesa moguće je dobiti ako se prođe kroz listu koja se nalazi u `EPROCESS` bloku. Drugim riječima, prolazimo kroz dvostruko povezanu listu dok ne dođemo na početak ili kraj. Svi pronađeni identifikacijski brojevi procesa spremaju se u listu koja se na kraju usporedi sa referentnom listom.

U slučaju da je zakačena funkcija `NtQuerySystemInformation()` metodom izmjene *SSDT* tablice, može se direktno pozvati `NtQuerySystemInfomation()` funkcija sustavskim pozivom odnosno instrukcijom `INT 2Eh` (Slika 4.10). Naravno, moramo poznavati servisni broj funkcije. Popisi svih servisnih brojeva mogu se naći na [6].

```
NTSTATUS UtilsZwRoutine(ULONG ZwIndex,...)
{
  NTSTATUS  status;

  _asm {
    // postavljanje servisnog broja
    MOV  EAX, [ZwIndex]
    // stog pokazuje na drugi parametar funkcije UtilsZwRoutine
    LEA  EDX, [EBP+0xC]
    // poziv prekida
    INT  0x2Eh
    // spremanje povratne vrijednosti
    MOV  [status], EAX
  }

  return status;
}
```

Slika 4.10. Funkcija uz pomoć koje se pozivaju temeljne *API* funkcije

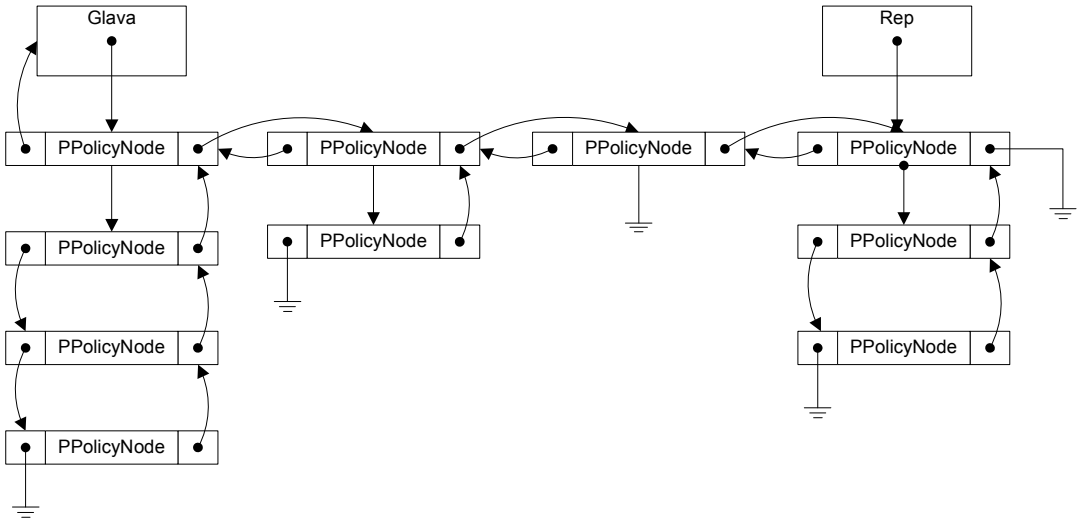
Jedna od zanimljivih metoda detekcije skrivenih procesa je metoda koju koristi program *F-secure Blacklight*. Pomoću funkcije `OpenProcess()` otvara sve procese s identifikacijskim brojem od `0x0` do `0x4E1C`. Ukoliko je funkcija uspjela za pojedini identifikacijski broj procesa, taj identifikacijski broj sprema se u listu. Nakon što prođe sve identifikacijske brojeve, usporedi dobivenu listu procesa s listom procesa koju vrati funkcija `CreateToolhelp32Snapshot()`.

5. Sustav sigurnosne politike

Zadatak sustava sigurnosne politike je kontrola izvođenja procesa te provjera autentičnosti svakog pokrenutog procesa za kojeg je definirano pravilo u sustavu. Moglo bi se definirati više načina detekcije autentičnosti, npr.:

1. jednostavna – koja provjerava veličinu, vrijeme kreiranja, vrijeme modificiranja izvršne datoteke procesa, ili
2. Detaljna – koja pored jednostavne provjere izračunava sažetak izvršne datoteke procesa pomoću md5 algoritma.

Sustav se sastoji od dvostruko povezane liste. Svaki čvor ima i kazaljku na listu djece. Ovakvom organizacijom dobivaju se horizontalna i vertikalna lista. Roditelji su smješteni u horizontalnoj listi, dok su djeca smještena u vertikalnoj listi.



Slika 5.1. Organizacija listi pravila u sustavu sigurnosne politike

Prilikom pokretanja procesa postoji informacija koji se proces pokreće i koji ga proces pokreće (odnos dijete-roditelj), te je ovakva organizacija najprikladnija.

Proces može pokrenuti novi proces koji se tada naziva proces dijete. Proces koji pokreće novi proces naziva se proces roditelj. Izvršna datoteka nekog procesa može se javiti više puta u sustavu. Moguće je čak da se pojavi kao roditelj i kao dijete. Ovakav sustav omogućuje da se isti dijete proces od jednog roditelja izvrši dok od drugog roditelja ne.

Sustav sigurnosne politike ima ostvarenu funkciju `Optimize()` koja optimizira horizontalnu i vertikalnu listu. Svaki čvor posjeduje brojač `UsageCounter` koji se uvećava svaki puta kada se primijeni pravilo iz dotičnog čvora. Funkcija `Optimize()` prema brojačima `UsageCounter` sortira liste prema padajućim vrijednostima, čime se znatno ubrzava pretraživanje listi. Programi koji se češće pokreću bit će na počecima listi.

Sustav definira funkciju `CheckRule()` koja za zadane izvršne datoteke procesa roditelja i procesa dijete provjerava stanje u sustavu. Funkcija traži pojavljivanje izvršne datoteke procesa roditelja u listi. Pronađe li se čvor za izvršnu datoteku roditelja, i ako proces roditelj ima dozvolu `prAllowChild` za izvršavanje djece, tada se u listi djece traži pojava izvršne datoteke procesa djeteta, i ako se i taj čvor pronađe, tada se postupa prema pravilima definiranim u tom čvoru. U suprotnom, ako roditelj nema dozvolu `prAllowChild` za izvršavanje djece, tada se lista djece i ne pretražuje, već funkcija vraća vrijednost da roditelj nema dozvolu pokretanja djece, odnosno vraća vrijednost `prrTerminate` koja ukazuje da se proces ne smije pokrenuti.

Prilikom provjere pravila u čvoru provjerava se i autentičnost procesa za kojeg je to pravilo definirano. Ako se ustvrdi da je proces izmijenjen, a nema dozvolu `prAllowModified`, tada funkcija vraća vrijednost `prrModified` koja ukazuje da je došlo do promjene izvršne datoteke procesa.

Funkcija `CheckRule()` vraća vrijednost `prrNotFound` ukoliko ne pronađe čvor s pravilima za zadane procese.

Dozvola	**Opis značenja**
prAllow	Dozvoljava izvođenje procesa.
prAllowModified	Dozvoljava izvođenje procesa čak i ako se izvršna datoteka procesa izmijenila. Ukoliko je uključena ova dozvola, tada dozvola `prAllow` nije potrebna.
prAllowChild	Dozvola za izvođenje djece. Ako proces roditelj posjeduje djecu, a u dozvolama nema ovu dozvolu tada se neće ni jedno dijete moći pokrenuti, makar ona pojedinačno imaju dozvolu `prAllow` za izvođenje.

Tablica 5.1. Tablica dozvola za izvođenje procesa

6. Praktični rad

Cilj praktičnog rada je zaštita operacijskog sustava *MS Windows* XP od pokretanja neželjenih te virusima izmijenjenih programa. Procesna zaštitna stijena definira sigurnosnu politiku za ponašanje procesa. Time se omogućuje kontrolirano izvršavanje programa. Praktični dio podijeljen je u dva dijela:

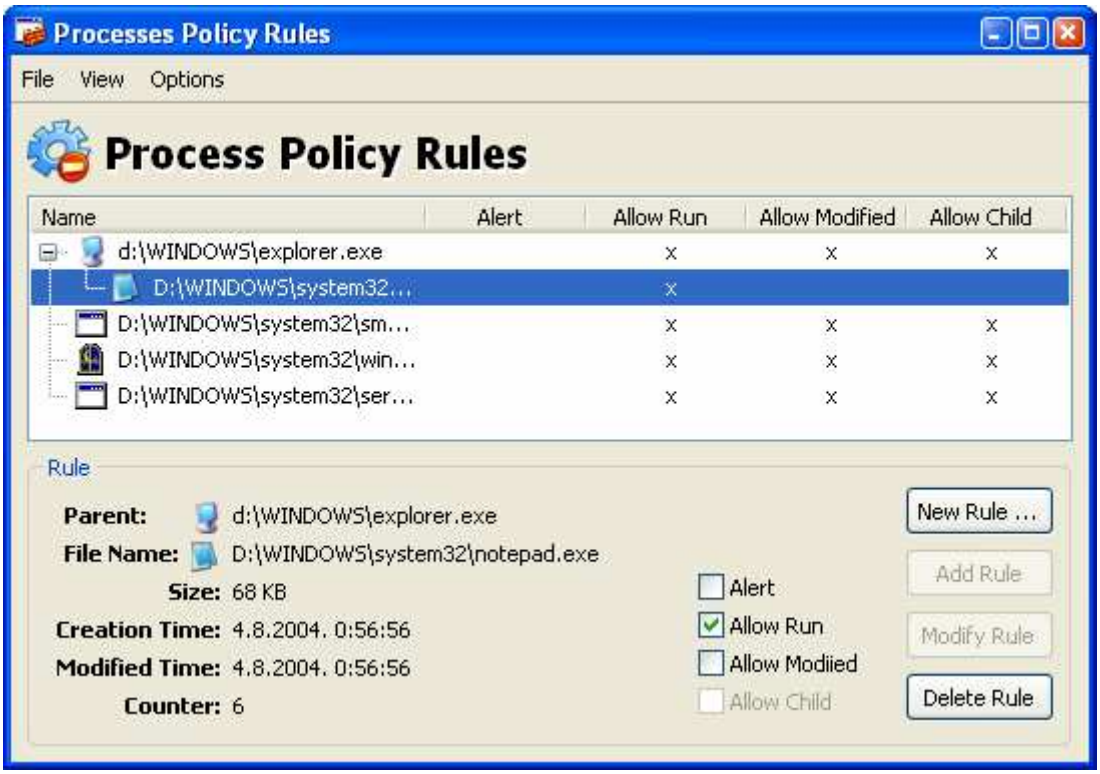
- upravljački program i
- nadzorna aplikacija.

Upravljački program dobiva informaciju od operacijskog sustava o pokrenutom novom procesu, zaustavlja njegovo daljnje izvršavanje, te šalje informaciju nadzornoj aplikaciji.

Nadzorna aplikacija, ostvarena kao sustavski servis, nakon dobivanja informacije koji je proces pokrenut provjerava proces u sustavu sigurnosne politike, te postupa prema definiranom pravilu.

6.1. Definiranje sigurnosne politike za ponašanje procesa

U nadzornoj aplikaciji ostvareno je održavanje sustava sigurnosne politike.



Slika 6.1. Prikaz pravila sigurnosne politike

Na listi su prikazana sva pravila iz sustava sigurnosne politike. Pravila su prikazana po principu roditelj-dijete, odnosno u stablastoj strukturi. Odabirom jednog od pravila s liste, u okviru *Rule* pokazuju se informacije o odabranom pravilu. U okviru *Rule* prikazane su i informacije o izvršnoj datoteci procesa:

- veličina (*Size*),
- vrijeme stvaranja izvršne datoteke (*Creation Time*),
- vrijeme zadnje promjene izvršne datoteke (*Modified Time*),
- izvršna datoteka procesa (*File Name*) i
- izvršna datoteka procesa roditelja (*Parent*).

Pored informacija o izvršnoj datoteci procesa tu se nalazi i brojač primjena pravila te uvjeti izvršavanja:

- Informiranje o provedenom pravilu u *Systrayu* (*Alert*).
- Dopušta se izvođenje procesa (*Allow Run*).
- Dopušta izvođenje procesa, iako je došlo do promjene izvršne datoteke procesa (*Allow Modified*).
- Procesu roditelju dopušta se izvođenje procesa djece (*Allow Child*).

Izmjenom jednog od uvjeta izvršavanja omogućuje se opcija izmjene pravila (*Modify Rule*). Dodavanje novog pravila vrlo je jednostavno. Odabirom opcije *New Rule* otvara se prozor s popisom izvršnih datoteka. Ovdje se odabere izvršna datoteka procesa za koji se želi izraditi novo pravilo. Nakon toga odaberu se željeni uvjeti izvršavanja te pritiskom na gumb *Add Rule* pravilo ulazi u sustav sigurnosne politike. Ukoliko je odabrano jedno od pravila procesa roditelja, tada će ubačeno pravilo biti kao dijete odabranog procesa. U suprotnom novo pravilo će biti roditelj.

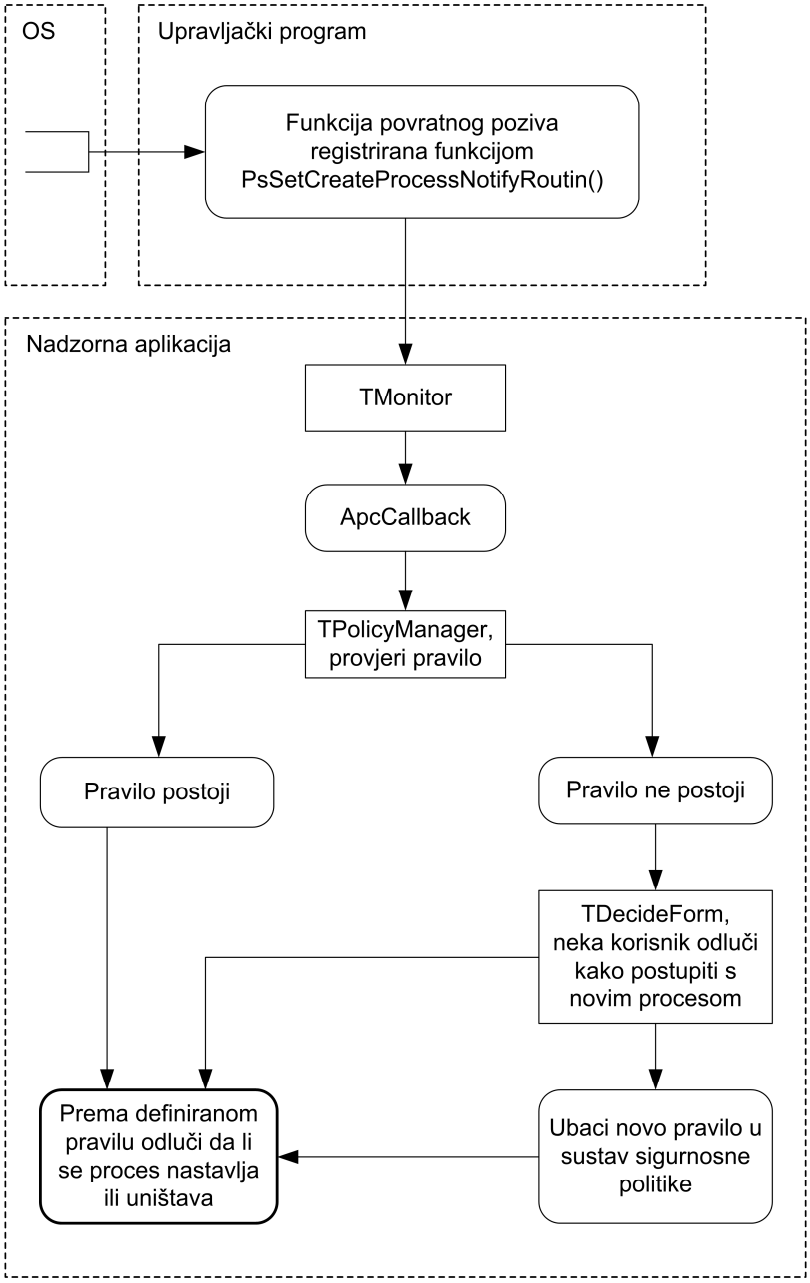
Odabirom pravila i pritiskom na gumb *Delete Rule* briše se obilježeno pravilo.

U slučaju da se doda novo pravilo preko sustava za nadzor pokretanja procesa, lista pravila automatski se osvježava.

6.2. Sustav nadzora pokretanja procesa

Nakon što se novi proces stvori, operacijski sustav poziva funkciju povratnog poziva prijavljenu *API* funkcijom `PsSetCreateNotifyRoutine()`. Funkcija povratnog poziva korištenjem asinkrone procesne komunikacije poziva funkciju nadzorne aplikacije `ApcCallback()` ostvarenu u razredu `TMonitor`. U funkciji `ApcCallback()` poziva se funkcija za provjeru procesa u sustavu sigurnosne politike `CheckRule()` ostvarenu u razredu `TPolicyManager`.

U slučaju da nije definirano pravilo ponašanja za pokrenuti proces, poziva se funkcija `ShowDecideForm()` koja otvara prozor i daje mogućnost korisniku da odluči što će učiniti sa procesom: da li će dopustiti da se nastavi izvođenje ili će ga uništiti.



Slika 6.2. Blok dijagram tijeka detekcije novog procesa, te provjere u sustavu sigurnosne politike

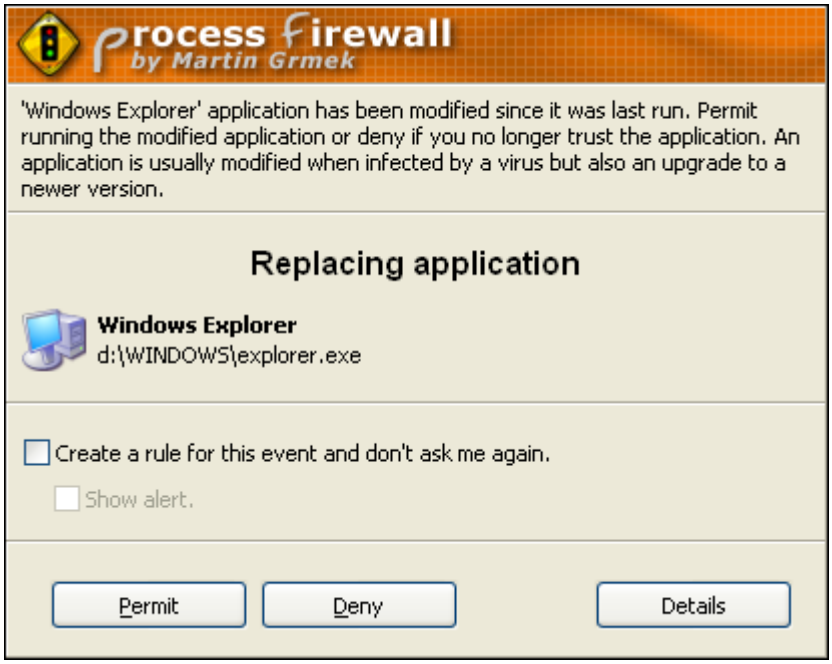
Korisnik ima dvije mogućnosti: nastaviti izvođenje novog procesa (*Permit*) ili zabraniti izvođenje novog procesa, odnosno uništiti ga (*Deny*). Odluka korisnika bit će pohranjena u sustav sigurnosne politike ukoliko korisnik obilježi opciju *Create a rule for this event and don't ask me again*. Odabirom opcije *Allow parent to run other applications* u sustav sigurnosne politike sprema se pravilo koje će dopustiti procesu roditelju da pokreće ostale procese. S ovom opcijom treba biti vrlo pažljiv jer se nakon isključivanja ove opcije zabranjuje roditelju da pokreće djecu, bez obzira da li su definirana pravila za ostalu djecu tog roditelja.

Slika 6.3. Prozor koji se otvara nakon što je pokrenut novi proces, a za taj proces nije definirana sigurnosna politika

Pored detekcije pokretanja novih procesa, ovaj sustav brine se o informiranju korisnika ukoliko se pokrene izmijenjena aplikacija za koju postoji pravilo u sustavu sigurnosne politike. Npr. ako se izvršna datoteka aplikacije *Notepad* zarazi nekim virusom, tada će se promijeniti njena veličina te vrijeme zadnje promjene datoteke.

U ovom slučaju sustav sigurnosne politike prepoznat će da je došlo do izmjene izvršne datoteke procesa te će o tome obavijestiti korisnika.

Korisnik ima dvije mogućnosti: nastaviti izvođenje izmijenjenog procesa (*Permit*) ili zabraniti izvođenje izmijenjenog procesa, odnosno uništiti ga (*Deny*). Kao i kod pokretanja novog procesa, i u ovom slučaju postoji mogućnost *Create a rule for this event and don't ask me again*. Ukoliko se obilježi ova mogućnost, odluka korisnika bit će pohranjena u sustav sigurnosne politike.



Slika 6.4. Prozor koji se otvara nakon što se pokrene proces kojem je izmijenjena izvršna datoteka, a za kojeg je definirano pravilo u sustavu sigurnosne politike

6.3. Kontrola procesa

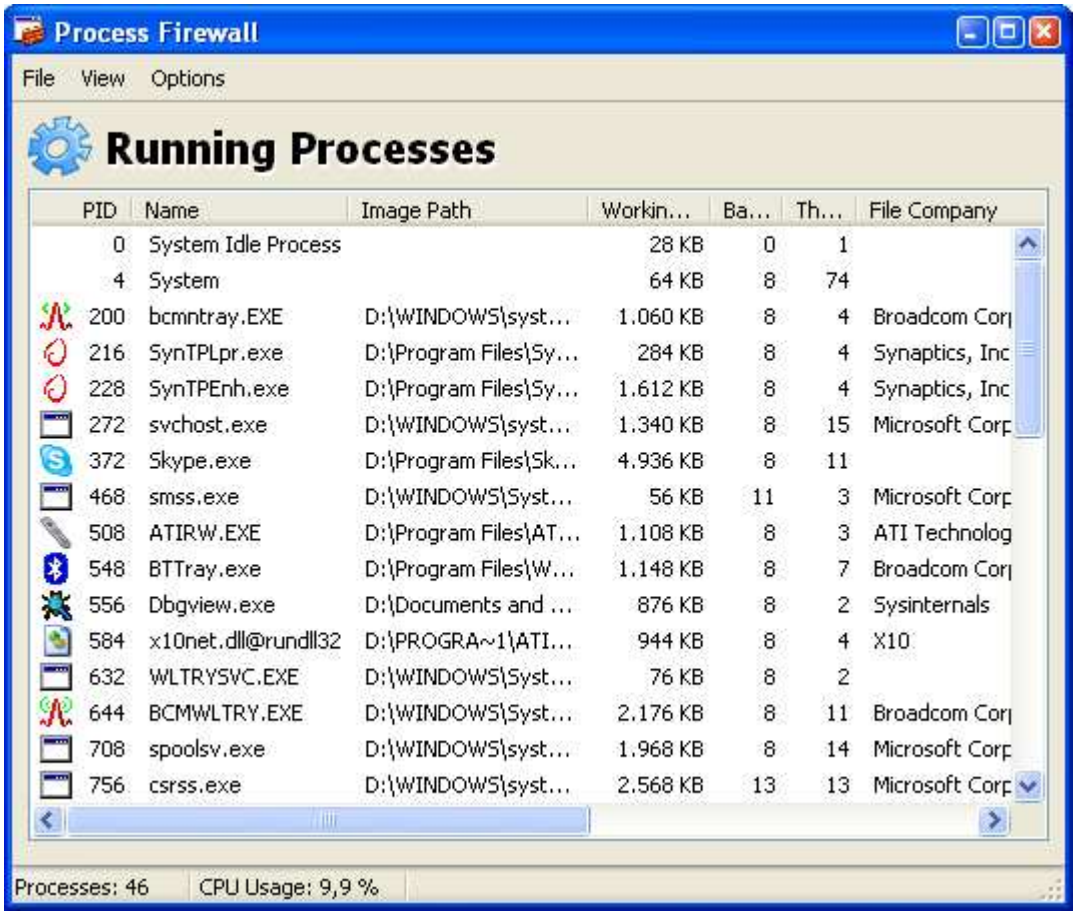
Nadzorna aplikacija ima ostvaren monitor aktivnih procesa. Za svaki aktivni proces prikazana je pridružena ikona. Kako bi se smanjila opterećenost sustava, za dohvaćanje pridruženih ikona koristi se funkcija `SHGetFileInfo()` koja vraća indeks ikona u polju globalnih ikona koje operacijski sustav koristi.

```
function GetFileSystemIconIndex(Path: String; Small: Boolean):
Integer;
var
  FileInfo: TSHFileInfo;
  Fl: Dword;
begin
  Result := 0;
  // Postavljanje zastavica za dohvaćanje indeksa ikone
  if not Small then
    Fl := SHGFI_SYSICONINDEX or SHGFI_LARGEICON
  else Fl := SHGFI_SYSICONINDEX or SHGFI_SMALLICON;

  // poziv funkcije za dohvat infomacije o zadanoj datoteci
  SHGetFileInfo(PChar(Path), 0, FileInfo,
                SizeOf(TSHFileInfo), Fl);
  // funkcija vraća indeks ikone
  Result := FileInfo.iIcon;
end;
```

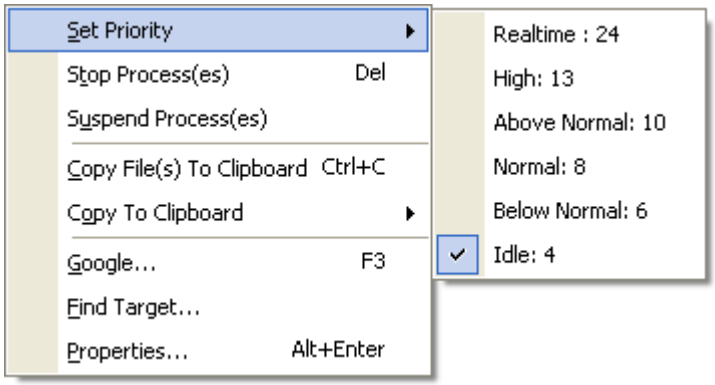
Slika 6.5. Funkcija za dohvaćanje indeksa pridružene ikone u listi ikona operacijskog sustava

Lista aktivnih procesa s ikonama puno je preglednija nego klasična lista procesa koja se koristi u programu *Task Manager*. Vizualno je lakše pronaći traženi proces. Svaki zaustavljeni proces označen je sivom bojom.



Slika 6.6. Lista aktivnih procesa, sa dodatnim informacijama

Pritiskom desne tipke miša na obilježeni proces, prikazuje se izbornik s mogućnošću izbora operacija. Moguće je odabrati i više procesa odjednom te se tada odabrane akcije odnose na sve obilježene procese.

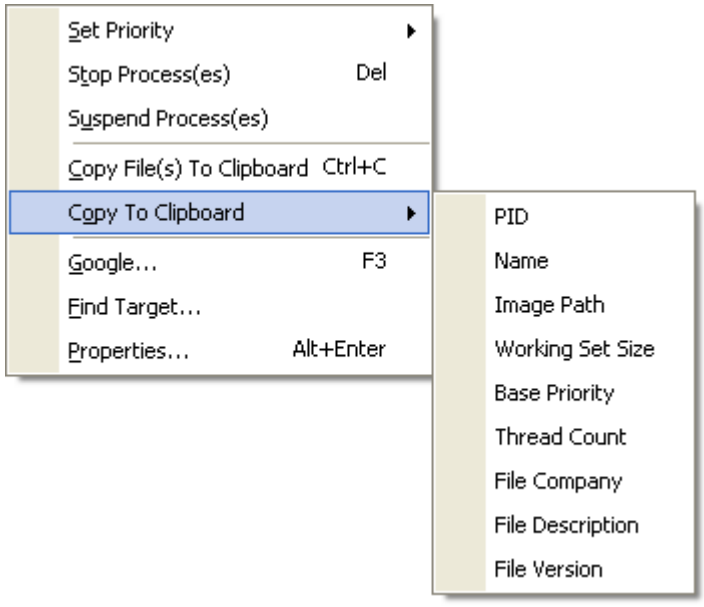


Slika 6.7. Izbornik operacija koje je moguće izvesti nad odabranim procesima

Definirane su operacije:

- Postavljanje prioritetnog razreda obilježenog procesa (*Set Priority*). Određeno je šest razreda prioriteta:
    - razred realnog vremena (*Realtime*),
    - razred visokog prioriteta (*High*) ,
    - razred iznad-normalnog prioriteta (*Above Normal*),
    - razred normalnog prioriteta (*Normal*),
    - razred ispod-normalnog prioriteta (*Below Normal*) i
    - razred praznog hoda (*Idle*).

  Razredi su detaljno objašnjeni u poglavlju 4.3.
- Uništavanje obilježenog procesa (*Stop Process(es)*). Uništavanje procesa objašnjeno je u poglavlju 4.2.
- Zaustavljanje obilježenog procesa (*Suspend Process(es)*). Zaustavljanje procesa objašnjeno je u poglavlju 4.1.
- Kopiranje izvršne datoteke obilježenog procesa u međuspremnik operacijskog sustava. Ova opcija nam omogućuje da korištenjem programa *Windows Explorer* kopiramo datoteku izvršnog procesa na željenu lokaciju.
- Kopiranje informacija vezanih uz obilježeni proces u međuspremnik operacijskog sustava. Moguće je kopirati samo one informacije koje su prikazane kao kolone na popisu aktivnih procesa.



Slika 6.8. Izbornik sa mogućnošću kopiranja raznih informacija o obilježenom procesu u međuspremnik operacijskog sustava

- Pretraživanje Interneta pomoću *Google* pretraživača u potrazi za izvršnom datotekom procesa. Prilikom pokretanja ove funkcije, otvara se Internet pretraživač koji se podrazumijeva za otvaranje Internet stranica.

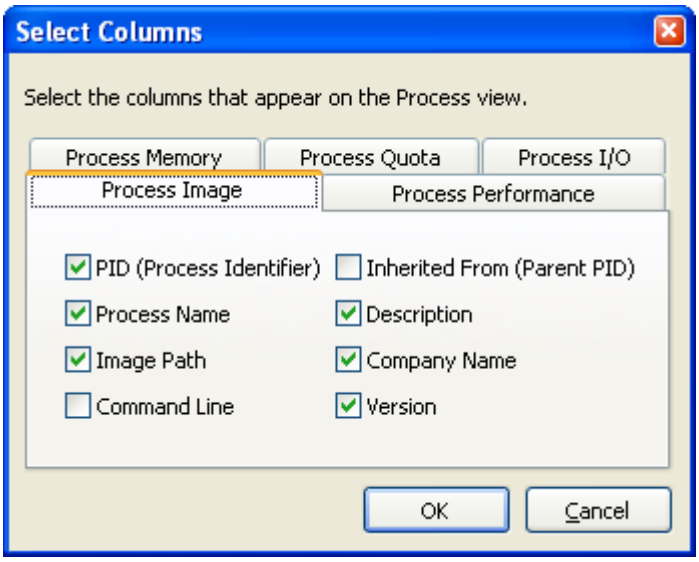
- Pronalazak izvršne datoteke obilježenog procesa (*Find Target*) otvara mapu izvršne datoteke obilježenog procesa programom *Windows Explorer*, te je obilježava.
- Svojstva obilježenog procesa. Otvara se sustavski prozor sa svojstvima izvršne datoteke obilježenog procesa (*Properties*).

Odabirom opcije *View* → *Select Columns* moguće je izmijeniti kolone koje se prikazuju u listi aktivnih procesa. Kolone su grupirane u sljedeće kategorije:

- Izvršna datoteka procesa (*Process Image*).

  Dostupne kolone:

  - Identifikacijski broj procesa (*PID*).
  - Identifikacijski broj procesa roditelja (*Inherited From*).
  - Samo ime izvršne datoteke procesa. Ukoliko se radi o procesu pokrenutom programom *rundll32.exe*, vrijednost kolone bit će *ime_dll_datoteke@rundll32* (*Process Name*).
  - Ime izvršne datoteke procesa s uključenom mapom (*Image Name*).
  - Naredbeni redak predstavlja redak kojim se pokreće proces uključujući zadane parametre (*Command Line*).
  - Opis *Description* iz *Version* bloka uključenog u izvršnu datoteku procesa (*Description*).
  - Opis *Company Name* iz *Version* bloka uključenog u izvršnu datoteku procesa (*Company Name*). Predstavlja naziv kompanije koja je izradila aplikaciju.
  - Opis *Version* iz *Version* bloka uključenog u izvršnu datoteku procesa (*Version*). Predstavlja inačicu aplikacije.

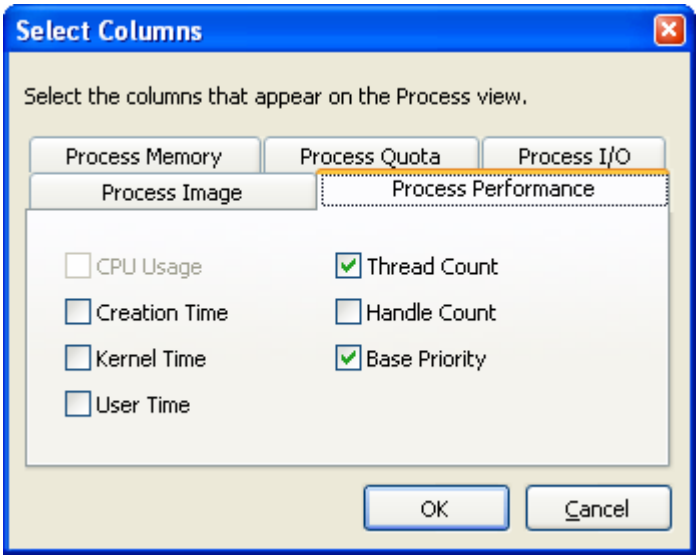


Slika 6.9. Odabir kolona za grupu *Izvršna datoteka procesa*

- Radna svojstva procesa (*Process Performance*).

  Dostupne kolone:

  - Vrijeme kreiranja procesa (*Creation Time*).
  - Vrijeme koje je proces proveo u jezgri (*Kernel Time*).
  - Vrijeme koje proces nije proveo u jezgri (*User Time*).
  - Broj dretvi procesa (*Thread Count*).
  - Broj ručica otvorenih u procesu (*Handle Count*).
  - Prioritetni razred procesa (*Base Priority*).

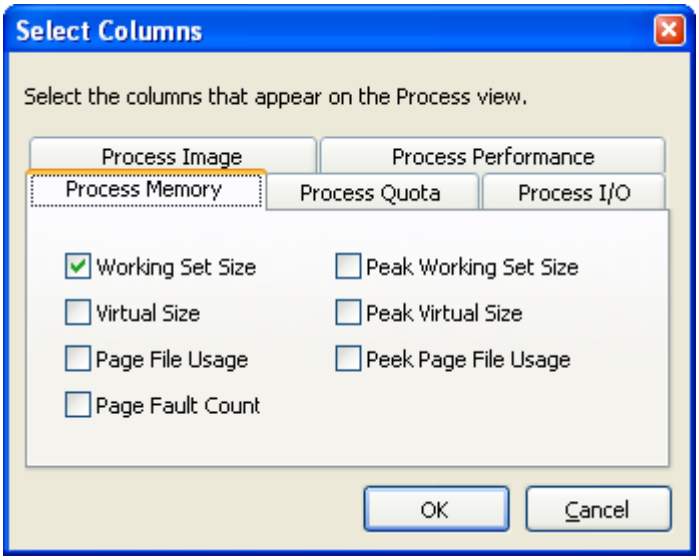


Slika 6.10. Odabir kolona za grupu *Radna svojstva procesa*

- Spremnici procesa (*Process Memory*).

  Dostupne kolone:

  - Veličina radnog spremnika dodijeljenog procesu (*Working Set Size*).
  - Vršna vrijednost radnog spremnika dodijeljenog procesu (*Peak Working Set Size*).
  - Veličina virtualnog spremnika dodijeljenog procesu (*Virtual Size*).
  - Vršna vrijednost virtualnog spremnika dodijeljenog procesu (*Peak Virtual Size*).
  - Skup okvira u aktivnom stanju koji su dodijeljeni procesu (*Page File Usage*).
  - Vršna vrijednost skupa okvira u aktivnom stanju koji su dodijeljeni procesu (*Peak Page File Usage*).
  - Prekid kada aplikacija želi čitati ili pisati na lokaciju u virtualnom spremniku koji je označen kao nedostupan. Ova kolona predstavlja broj koliko je puta podatak bio dohvaćen sa diska zato što nije pronađen u spremniku (*Page Fault Count*).

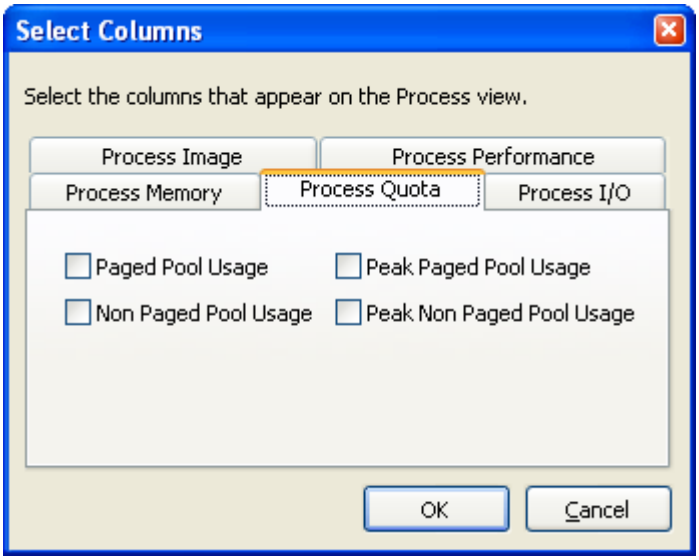


Slika 6.11. Odabir kolona za grupu *Spremnici procesa*

- Kvote procesa (*Process Quota*).

  Dostupne kolone:

  - Virtualni spremnik zauzet od strane operacijskog sustava koji je dodijeljen procesu, koji može biti podijeljen na stranice. Straničenje je micanje dijela spremnika procesa koji se ne koristi učestalo iz radnog spremnika u neki drugi spremnik, najčešće na tvrdi disk. Ova kolona prikazuje veličinu ovako zauzetog virtualnog spremnika korištenog od strane procesa (*Peaged Pool Usage*).
  - Vršna vrijednost virtualnog spremnika zauzetog od strane operacijskog sustava koji je dodijeljen procesu, a može biti podijeljen na stranice (*Peak Peaged Pool Usage*).
  - Veličina spremnika zauzeta od strane operacijskog sustava koji nije nikad bio straničen, odnosno premještan iz radnog spremnika u neki drugi spremnik, najčešće na tvrdi disk (*Non Peaged Pool Usage*).
  - Vršna vrijednost veličine spremnika zauzetog od strane operacijskog sustava koji nije nikad bio straničen (*Peak Non Peaged Pool Usage*).

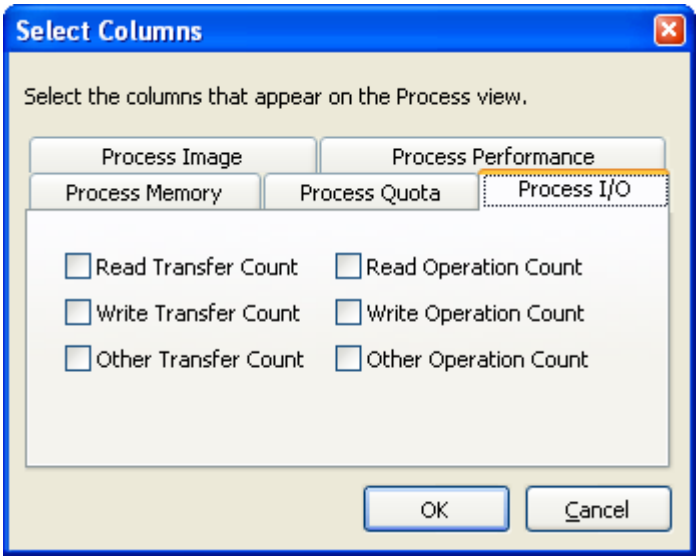


Slika 6.12. Odabir kolona za grupu *Kvote procesa*

- Ulazno-izlazna svojstva procesa (*Process I/O*).

  Dostupne kolone:

  - Broj pročitanih okteta u ulazno-izlaznim operacijama čitanja generiranih od strane procesa, uključujući datoteke, mrežu i ulazno-izlazne jedinice (*Read Transfer Count*).
  - Broj ulazno-izlaznih operacija čitanja generiranih od strane procesa, uključujući datoteke, mrežu i ulazno-izlazne jedinice (*Read Operation Count*).
  - Broj zapisanih okteta u ulazno-izlaznim operacijama pisanja generiranih od strane procesa, uključujući datoteke, mrežu i ulazno-izlazne jedinice (*Write Transfer Count*).
  - Broj ulazno-izlaznih operacija pisanja generiranih od strane procesa, uključujući datoteke, mrežu i ulazno-izlazne jedinice (*Write Operation Count*).
  - Broj zapisanih okteta u ostalim ulazno-izlaznim operacijama generiranih od strane procesa, uključujući datoteke, mrežu i ulazno-izlazne jedinice (*Other Transfer Count*).
  - Broj ostalih ulazno-izlaznih operacija generiranih od strane procesa, uključujući datoteke, mrežu i ulazno-izlazne jedinice (*Other Operation Count*).



Slika 6.13. Odabir kolona za grupu *Ulazno-izlazna svojstva procesa*

7. Zaključak

U radu su prikazani: postupak izgradnje sustava sigurnosne politike za ponašanje procesa i manipulacija aktivnim procesima. Opisane su tehnike kojima je moguća detekcija novih procesa u operacijskom sustavu *MS Windows XP*, čime se omogućuje definiranje sigurnosne politike za ponašanje pojedinog procesa kako bi se zaštitio sustav od neželjenih aplikacija.

Rad sadrži listu procesa koji su trenutno pokrenuti u operacijskom sustavu. Nad aktivnim procesom moguće je izvršiti sljedeće radnje: zaustavljanje procesa, nastavljanje zaustavljenog procesa, uništavanje procesa i izmjena prioritetnog razreda procesa.

Ostvarena je metoda detekcije pokretanja procesa uporabom *API* funkcije `PsSetCreateProcessNotifyRoutine()`. Nažalost, ova metoda za detekciju pokretanja procesa ne omogućuje nadzor nad pokretanjem sustavskih servisa i upravljačkih programa. Ovime se otvara prostor da „crvi“, „trojanski konji“ i neželjeni programi ipak budu pokrenuti bez znanja korisnika, čime cijeli koncept zaštite operacijskog sustava od pokretanja neželjenih programa postaje upitan.

Svaki proces nakon pokretanja se zaustavlja, provjerava u sustavu sigurnosne politike te se potom ponovno pokreće ukoliko je tako definirano pravilom ponašanja. Ovime se usporava cijeli sustav, a i nepotrebno se zauzimaju sredstva potrebna za proces ako pravilo ponašanja blokira izvršavanje procesa.

Kako bi sustav detekcije pokretanja procesa bio u potpunosti siguran, potrebno je ostvariti detekciju izvršavanja procesa izmjenom *SSDT* tablice. Pored detekcije pokretanja procesa potrebno je i praćenje pokretanja sustavskih servisa i upravljačkih programa. Potrebno je pratiti temeljne *API* funkcije za mapiranje adresnog prostora i funkcije za upravljanjem dretvi kako se uljez ne bi ubacio u već postojeći proces čime bi prekrio svoje postojanje. Praćenje svih temeljnih *API* funkcija moguće je izmjenom *SSDT* tablice kako je prikazano u poglavlju 2.3.1.

U usporedbi s antivirusnim programom, ovakav način zaštite operacijskog sustava ipak je bolji. Antivirusni programi dosta opterećuju i usporavaju sustav stalnim pregledavanjem datoteka u potrazi za mnogim oznakama virusa. Pored toga, sustav obnove baze poznatih virusa je trom, te do trenutka izdavanja nove baze virusa može proći i više dana, a za to vrijeme virus se može proširiti cijelim računalnim sustavom.

Dodatak A

```
function GetProcessFileNameCommandLine(PID: DWord;
        var FilePath: String; var CmdLine: String): Boolean;
var
  pbi: PROCESS_BASIC_INFORMATION;
  size: DWord;
  PEBBlock: PEB;
  Block: RTL_USER_PROCESS_PARAMETERS;
  cmd, name: PWideChar;
begin
  cmd := nil; name := nil;

  // otvaranje ručice procesa zadanog identifikacijskim brojem
  // sa svim pravima pristupa
  hProcess := OpenProcess(PROCESS_ALL_ACCESS, False, PID);
  Result := hProcess <> 0;

  // ako je ručica uspješno otvorena
  if Result then begin
    // dohvaćanje PBI bloka (Process Basic Information) u
    // kojem se nalazi adresa PEB bloka
    NtQueryInformationProcess(hProcess, 0, @pbi,
                              sizeof(pbi), @size);

    // dohvaćanje PEB bloka
    Result := ReadProcessMemory(hProcess, pbi.PebBaseAddress,
                          @PEBBlock, sizeof(PEBBlock), size);

    // PEB blok je uspješno dohvaćen
    if Result then begin
      // čitanje RTL_USER_PROCESS_PARAMETERS stukture koja
      // sadrži informacije o stazi izvršne datoteke procesa
      // i naredbenom retku
      Result := ReadProcessMemory(hProcess,
                                  Ptr(PEBBlock.InfoBlockAddress),
                                  @Block, sizeof(Block), size);

      // struktura je uspješno dohvaćena
      if Result then begin
        // zauzimanje spremnika za pohranjivanje staze izvršne
        // datoteke procesa i narebenog retka procesa
        name := AllocMem(Block.ImagePathName.MaximumLength);
        cmd := AllocMem(Block.CommandLine.MaximumLength);

        try
          // čitanje staze izvršne datoteke procesa
          Result := ReadProcessMemory(hProcess,
                    Block.ImagePathName.Buffer, name,
                    Block.ImagePathName.MaximumLength, size);

          // ukoliko čitanje staze izvršne datoteke procesa
          // nije uspjelo, tada se vjerojatno radi o posmaku,
          // a ne o kazaljci
          if not Result then begin
            // proračun nove kazaljke koja pokazuje na stazu
```

```
            // izvršne datoteke procesa
            adr := Cardinal(Block.ImagePathName.Buffer) +
                   PEBBlock.InfoBlockAddress;

            // čitanje staze izvršne datoteke procesa
            Result := ReadProcessMemory(hProcess, Ptr(adr),
                      name, Block.ImagePathName.Length, size);
          end;
          FilePath := name;

          // čitanje naredbenog retka procesa
          Result := ReadProcessMemory(hProcess,
                    Block.CommandLine.Buffer, cmd,
                    Block.CommandLine.MaximumLength, size);

          // ukoliko čitanje naredbenog retka procesa
          // nije uspjelo, tada se vjerojatno radi o posmaku,
          // a ne o kazaljci
          if not Result then begin
            // proračun nove kazaljke koja pokazuje na
            // nareebeni redak procesa
            adr := Cardinal(Block.CommandLine.Buffer) +
                   PEBBlock.InfoBlockAddress;

            // čitanje naredbenog retka procesa
            Result := ReadProcessMemory(hProcess, Ptr(adr),
                      cmd, Block.CommandLine.Length, size);
          end;
          CmdLine := cmd;
        finally
          // oslobađanje sprenika
          if name <> nil then
            FreeMem(name, Block.ImagePathName.MaximumLength);

          if cmd <> nil then
            FreeMem(cmd, Block.CommandLine.MaximumLength);
        end;
      end;
    end;
  end
  else begin
    FileName := '';
    CmdLine := '';
    Result := False;
  end;
end;
```

Literatura

[1] MSDN Knowledge base Q197571

[2] *API* hooking revealed by Ivo Ivanov, dostupno na Internet adresi: `http://www.codeproject.com/system/hooksys.asp`

[3] Windows DDK Documentation, Process Structure Routines, dostupno na Internet adresi: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/Kernel_r/hh/Kernel_r/k108_a0f7bff2-270e-41fb-87d4-d8d533aa0bef.xml.asp`

[4] Hooking the native *API* and controlling process creation on a system-wide basis by Anton Bassov, dostupno na Internet adresi: `http://www.codeproject.com/system/soviet_protector.asp`

[5] Defeating Kernel Native *API* Hookers by Direct Service Dispach Table Restoration, dostuno na Internet adresi: `http://www.security.org.sg/code/SIG2_DefeatingNativeAPIHookers.pdf`

[6] Windows System Call Table (NT/2000/XP/2003), dostupno na Internet adresi: `http://www.metasploit.com/users/opcode/syscalls.html`

[7] DDK - Windows Driver Development Kit, dostupno na Internet adresi: `http://www.microsoft.com/whdc/devtools/ddk/default.mspx`

[8] Yet Another Thread Monitor by Tim Deveaux, dostupno na Internet adresi: `http://www.codeproject.com/threads/YATMon.asp`

[9] Inside NT's Asynchronous Procedure Call by Albert Almeida, dostupno na Internet adresi: `http://www.windevnet.com/documents/s=7653/win0211b/0211b.htm` (potrebna registracija)

[10] Win32 Process Suspend/Resume Tool by Daniel Turini, dostupno na Internet adresi: `http://www.codeproject.com/threads/pausep.asp`

[11] SSDT hook example (hiding processes) correction by Orkblutt, dostuno na Internet adresi: `https://www.rootkit.com/newsread.php?newsid=450`

[12] Finding some non-exported kernel variables in Windows XP by Edgar Barbosa, dostupno na Internet adresi: `http://www.yates2k.net/GetVarXP.pdf`