SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1815

OSTVARENJE PROGRAMSKE ZAŠTITE OD NEOVLAŠTENE UPORABE

Marijan Nikić

Zagreb, listopad 2009.

Sadržaj

1. UVOD

Već dugo vremena računalna industrija je jedna od najvećih i najprofitabilnijih industrija. Od računalnih igara, sve do profesionalnih programa, raznoraznih alata, aplikacija, danas gotovo za svaki aspekt života postoji nekakav računalni program koji pomaže ili rješava neke probleme profesionalne ili osobne prirode. To je industrija u kojoj se vrti gomila novca. Kako je proizvodnja i prodaja računalnih programa kroz povijest postajala sve skuplji i skuplji sport, sve je više bilo ljudi koji su pokušavali na njima zaraditi na ilegalan način. Tako su tzv. pirati počeli sve više distribuirati ilegalne kopije programskih proizvoda, što je, logično, štetilo računalnoj industriji. Stoga su smišljeni načini zaštite računalnih programa od ilegalnog kopiranja i korištenja. Danas je to dovedeno do golemih razmjera. Sve raznolikiji i bolji načini zaštite doveli su i do sve domišljatijih načina probijanja te zaštite, odnosno tzv. *krekiranja* (eng. *cracking*). Postoji sve više i više ljudi koji se bave probijanjem programske zaštite (tzv. „krekera“, eng. *cracker*), i sve više internetskih stranica na kojima se distribuiraju ilegalni programi, probijene izvršne datoteke (eng. *crack*), datoteke za automatsko probijanje zaštite (eng. *patch*), registracijski ključevi, generatori ključeva (eng. *keygen*), itd. Danas je postala normalna stvar da ubrzo nakon izlaska nekog novog programskog paketa, izađe i njegova ilegalna, probijena verzija negdje na Internetu.

Čini se kao da ne postoji zaštita koja se ne može relativno lako ukloniti, pa je stoga naizgled veliki rad koji programer vjerojatno mora uložiti u svoj proizvod želi li ga zaštititi potpuno beskoristan. Pitanje je samo vremena, mogućnosti i strpljenja kad će netko uspjeti probiti i naizgled neprobojnu zaštitu.

No svrha zaštite i nije da spriječi sâmo probijanje, nego prije da se onemogući ispravno uklanjanje zaštite. Tako probijeni program neće raditi, ili će raditi neispravno, što će navoditi više ljudi da kupuju originalne programe. Naravno, tijekom vremena, ispravna probijena verzija izvršne datoteke će se pojaviti, ali što se više vremena može dobiti dobrom zaštitom, veća će biti i zarada, jer će ljudi u tom periodu više kupovati originalne programe.

Ovaj diplomski rad fokusira se uglavnom na današnje trendove programske zaštite, te nekoliko osnovnih trenutno poznatih izvedbi zaštite. Usto, tekst opisuje i ponešto o načinima probijanja, te što je sve potrebno, od alata, do vremenskih zahtjeva i intelektualnih sposobnosti, za uspješno probijanje zaštite računalnih programa.

U sklopu ovog diplomskog rada ostvaren je i kombinirani model programske zaštite baziran na klijent-poslužitelj arhitekturi. Ovaj model također je opisan i analiziran u daljnjem tekstu.

2. PROGRAMSKA ZAŠTITA

2.1. Povijest antipiratske zaštite

2.1.1. Rane godine

Antipiratska zaštita od svojih početaka ponajviše se odnosila na epski rat između proizvođača računalnih igara i pirata. Računalno piratstvo oduvijek je bilo trn u oku industrije računalnih igara, ali u početku, to je bio manji problem nego što je danas. U ranim 80-tima, industrija igara bila je u svom začetku, a računalo se još uvijek koristilo primarno kao poslovni i proizvodni alat. Tada se nije koristila apsolutno nikakva zaštita protiv kopiranja.

Štoviše, piratstvo i kopiranje igara u to vrijeme bilo je iznimno teško. Računalni programi su se distribuirali u obliku raznih formata kaseta, i bilo kome bez nekakvog inženjerskog znanja bilo ih je teško duplicirati.

Sve ovo promijenilo se pojavom 5.25" disketa. Odjednom su se podaci mogli kopirati vrlo jednostavno preko računala s jednog medija na drugi. Programeri su imali iznimne koristi od ovog jer su mogli spremiti više podataka na medij i praviti kompleksnije igre, ali su također bili suočeni s novom generacijom igrača koji su naučili kako je lako kopirati i distribuirati kopirane računalne programe.

Odgovor na to je bio najstariji i najprimitivniji način zaštite od kopiranja. Igre su se ponekad distribuirale na disketama s rupicama na točno određenim mjestima. Drugi su pak zapisivali posebne datoteke na kraj diska koje su onemogućavale instalaciju igre s istog diska više od jednom. Ove su metode, međutim, otežale korištenje računalnih programa legitimnim korisnicima, koji su se počeli obraćati na tzv. BBS zajednice na Internetu, koje su tad bile u svom nastanku, u potrazi za amaterskim programskim zakrpama i aplikacijama za duplikaciju programa. Bilo je vrijeme za neko kreativnije rješenje.

2.1.2. Dekoderska era

Novo rješenje bila je „neračunalna“ zaštita od kopiranja. Najrašireniji oblik bilo je postavljanje pitanja tijekom igre, poput kviza, čiji je odgovor zahtijevao da korisnik pogleda u upute što su se prodavale u paketu s igrom da bi pronašao specifične informacije – primjerice, pitanje bi glasilo da korisnik upiše 3. riječ u 4. paragrafu na stranici 14 u uputama. Neke su igre čak imale kazne u slučaju krivog odgovora: „*Star Command*“ tvrtke SSI zahtijevao je pretraživanje uputa prije svakog sačuvanja pozicije u igri tijekom igranja, dok je „*Master Of Orion*“ postajao sve teži tako da bi konačno bilo nemoguće pobijediti. Možda najpoznatiji primjer je Sierrin „*King's Quest III*“, gdje su se podugački opisi za spravljanje napitaka, koji su bili potrebni za prolaz igre, nalazili u uputama – jedna pogreška u sastojcima značila je kraj igre.

Međutim, i ovaj način zaštite moglo se relativno lako zaobići – jednostavnim fotokopiranjem uputa. Stoga su neki proizvođači koristili poboljšani pristup – u pakete s računalnim igrama stavljali su i posebna pomagala koja su se morala koristiti za prolaz igre. Najpoznatiji i najdomišljatiji je definitivno kotač s licima gusara za „*The Secret Of The Monkey Island*“ od Lucas Artsa. Igra bi prikazala lice gusara sastavljeno od dva dijela, te bi bilo postavljeno pitanje kad je na određenom otoku taj prikazani gusar obješen. Korisnik bi zatim morao kombinirati lica na kotaču, i unijeti odgovarajuću godinu dobivenu uz točnu kombinaciju lica gusara i naziva otoka.

Ovaj način zaštite je funkcionirao neko vrijeme, ali se primijetilo da upiti za pretraživanje uputa često smanjuju užitak igranja. S rastom cijene proizvodnje igara, i pojavom CD-ROM medija, ovaj način zaštite sredinom 90-tih pao je u zaborav.

2.1.3. Era optičkih diskova

Pojava CD-ROM-a značila je da će se igre koje su prije dolazile na gomili disketa sada napokon moći isporučivati na jednom jedinom mediju, što je otvorilo nove multimedijalne mogućnosti u proizvodnji igara. No to je također značilo i pojavu CD-RW uređaja (CD pisača), koji su omogućili lak način zapisivanja i kopiranja, za ono vrijeme, ogromnih količina podataka na vrlo jednostavan način.

Odgovor na ovo u vidu zaštite od kopiranja bio je tehnološki. Prva ideja bila je korištenje nečitljivih sektora i skrivenih datoteka koje tadašnji programi za kopiranje CD-a nisu mogli čitati. Međutim, ovo je propalo čim je izmišljen program koji je mogao čitati i kopirati cijeli disk. Programerske kompanije potom su se okrenule tehnologijama „*SafeDisc*"-a i ranim oblicima „*SecuROM*"-a koje su činile kopije medija koji koriste ove tehnologije nečitljivima.

Nažalost, najčešći problem bila je nekompatibilnost. Jednostavno je bilo nemoguće razviti mehanizam koji bi radio na svakom sustavu ili na svakom optičkom uređaju na tržištu. Ovo je često dovodilo do kontroverznih situacija kao u slučajevima igara poput „*The Sims 2*" i „*Bioshock*" koje mnoštvo legitimnih kupaca nije moglo instalirati radi nekompatibilnosti računalnog sklopovlja, antivirusnih programa i raznih drugih sistemskih komponenti sa zaštitom od kopiranja zasnovanom na optičkim medijima.

Ništa od ovog, međutim, nije izazvalo veće proteste kao što je to bilo sa „*StarForce*" sustavom za zaštitu. Sustav koji je bilo jako teško probiti, ali također i jako teško maknuti s računala pri deinstalaciji igre, optuživali su svi redom, od blogova, do *Penny Arcade* i *CNet*-a, iz razloga što se ovaj sustav ponašao poput raznih štetnih programa (eng. *malware*). Uzevši u obzir i činjenicu da je „*StarForce*" rušio performanse sustava na kojem je bio instaliran, bio je veoma nepopularan čak i među ljudima koje piratstvo i distribucija kopiranih igara nije zanimalo.

2.1.4. Moderno doba

Pojavom brzog Interneta (eng. *broadband*), veoma se raširila upotreba tzv. CD ključa. Računalne igre danas traže unos jedinstvenog niza od 16 znakova (u većini slučajeva) koji dolazi u paketu s kupljenom igrom, i bez kojeg se igra ne može instalirati. Poslije toga, mogu se općenito raditi kopije sve dok se unosi ovaj ključ, iako neki distributeri dozvoljavaju samo ograničen broj kopija. Proizvođač računalnih igara PopCap, primjerice, ograničava kopije na maksimalno 6; svaki put kad se napravi kopija, igra se spaja na PopCap-ov poslužitelj i provjerava koliko je puta iskorišten CD ključ. Mrežne igre za više igrača ne dopuštaju da se dva korisnika logiraju s istim ključem u isto vrijeme.

Konačno, neke iznimno uspješne mrežne igre poput Blizzard-ovog „*World Of Warcraft*"-a nemaju potrebe za kontrolom nad piratskim kopijama, jer svatko tko koristi program mora dati broj svoje kreditne kartice kako bi se pretplatio na korištenje igre i postao dio svjetske „*WoW*" komune.

2.2. Osnovni tipovi zaštite računalnih programa

Teško je danas definirati neke specifične tipove zaštite. Svaki tip zaštite za pojedini program ovisit će o domišljatosti i znanju programera koji ga je isprogramirao. Štoviše, preporučljivo je koristiti neke inovativne načine zaštite, bez obzira koliko prokušanih i poznatih načina može postojati i biti dostupno, jer je za nekog tko će pokušati probiti zaštitu posao sigurno teži kad ne zna ni po „vanjskom" izgledu programa procijeniti o kakvoj se zaštiti radi i odakle da počne.

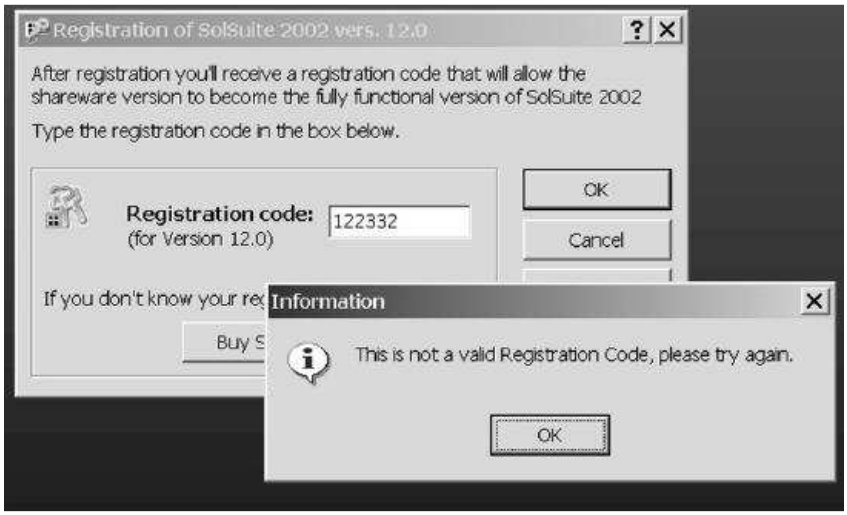
Ipak, moguće je definirati nekoliko osnovnih načina koji se koriste (uz, naravno, redovitu modifikaciju). Slijedi samo šturi opis vrsti zaštite kakve koristi većina programa s mogućnošću roka za isprobavanje prije registracije (eng. *trial*), programa s ograničenim opcijama prije registracije (eng. *shareware*) i sličnih proizvoda, pa i popriličan broj profesionalnih aplikacijskih paketa. Ovi tipovi zaštite koriste se uglavnom u međusobnim kombinacijama, i uz dodatne autorske varijacije.

2.2.1. Zaštita upotrebom registracijskog ključa

Najraširenij način zaštite računalnih programa danas je zaštita pomoću registracijskog broja/ključa, odnosno serijskog broja. Programeri koriste zaštitu s registracijskim ključem na više načina, ovisno o potrebi, neki od kojih su:

1) *uvijek isti registracijski ključ*

Ovo je najjednostavniji način zaštite s registracijskim ključem, iako se generalno ovaj tip zaštite kombinira s nekim drugima, koji će biti objašnjeni kasnije. Prednost je ovog načina zaštite što ispravan registracijski broj ne mora biti čuvan u memoriji za usporedbu s unešenim registracijskim brojem, jer je integriran s programskim kodom. Međutim, ovo olakšava posao probijanja, jer sve što je potrebno je saznati taj jedan i jedini, jedinstveni, registracijski ključ. Zbog toga se ovom metodom ne zaključava program u cjelini, već se zaključavaju pojedine njegove funkcije, tako da pri samom upisu registracijskog ključa točno upisan ključ ne otključava odmah cijeli program, već da se pojedini dijelovi programa otključavaju što kasnije, po mogućnosti pri pokretanju programa, ili, još bolje, tek kad se pojedina od tih funkcija koristi (primjerice funkcija za sačuvanje datoteke). Ako se funkcija dodatno enkodira nakon korištenja, onda program neće nikad biti dekodiran u memoriji u cijelosti, pa izravno dohvaćanje segmenata iz memorije pri pokušaju probijanja zaštite neće puno pomoći.



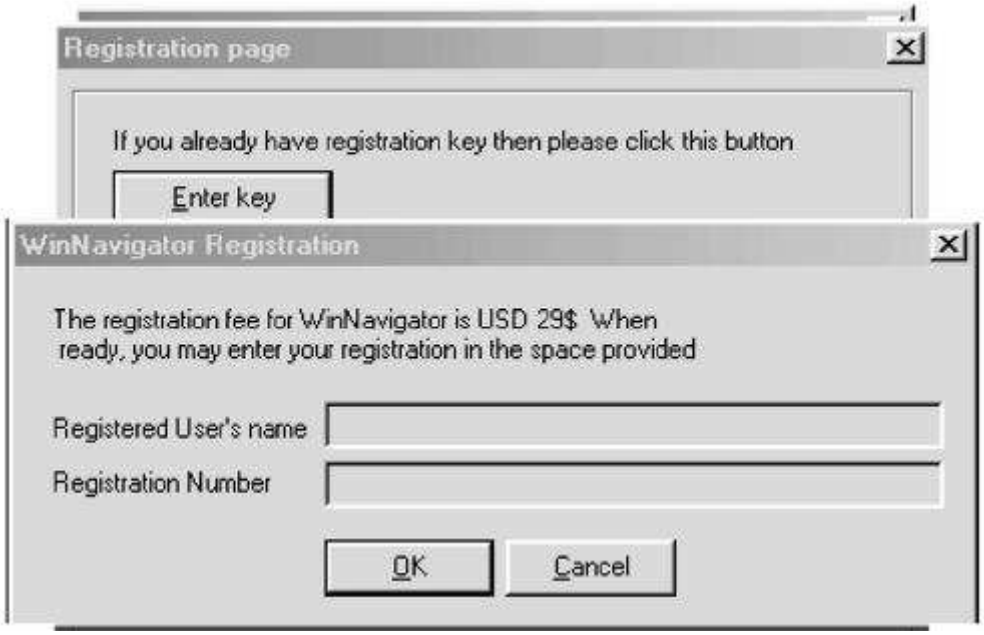
Slika 1. Primjer programa s uvijek istim registracijskim ključem

2) *registracijski ključ se mijenja u ovisnosti o drugim podacima, poput korisničkog imena, naziva tvrtke, i sl.*

Ova metoda radi tako da se uz registracijski ključ unose neke druge informacije, poput imena, prezimena, naziva tvrtke i sl. Za razliku od prethodne metode, ovdje ne postoji jedinstveni registracijski ključ, već programer smišlja

matematički algoritam kojem se kao varijable predaju informacije o korisniku koje je prethodno unio, a algoritam izračunava iz tih podataka registracijski ključ, koji se onda u memoriji uspoređuje s registracijskim ključem koji je korisnik upisao na upit. U ovom slučaju, dakle, jedinstven je matematički algoritam koji računa registracijski broj, dok je registracijski broj varijabilan. Jačina zaštite uglavnom će ovisiti o znanju programera i kompleksnosti algoritma, no ipak, ova metoda probija se relativno lako praćenjem aktivnosti memorije (jer se izračunati registracijski ključ sprema u memoriju radi usporedbe) i praćenjem izvođenja programa liniju po liniju koda (eng. *tracing*, *debugging*), te praćenjem stanja memorijskih resursa, registara i stanja stoga.

Za postizanje snažnije zaštite, može se algoritam isprogramirati tako da korisničko ime i registracijski broj moraju kombinirani davati određeni rezultat. Međutim, bez provjere registracijskog broja, postoji cijeli niz mogućih kombinacija koje daju traženo rješenje, pa se to može iskoristiti prilikom probijanja zaštite i izmisliti registracijski broj po vlastitoj želji. Jedno od mogućih rješenja ovog problema je da se u program implementira skriveni dodatni algoritam koji će naknadno provjeravati točnost registracijskog broja.



Slika 2. Primjer programa s registracijskim ključem ovisnim o drugim podacima

3) registracijski ključ mijenja se u ovisnosti o internim podacima i stanju računala

Ova metoda poprilično je neugodna za probijanje, jer je zavisna o računalu na kojem se instalira program, pa čak i da se uspije probiti zaštitu i napraviti piratsku verziju programa, ta piratska verzija neće moći raditi niti na jednom računalu osim onog na kojem je program probijen. Pri registraciji programa dohvaća se dodatno neki podatak s računala domaćina koji je jedinstven baš za to računalo, primjerice serijski broj tvrdog diska i sl., te se to kombinira dodatno s korisničkim imenom, nazivom kompanije itd. (da bi program smjela koristiti samo jedna osoba), i izračunava se registracijski broj pomoću posebnog algoritma, te uspoređuje s upisanim registracijskim brojem (kao u prethodnoj metodi). Veoma je bitno da se sakrije dio kôda u kojem se dohvaćaju sistemske informacije, kao i dio gdje se provjerava ispravnost registracijskog broja.

Slika 3. Primjer programa s registracijskim ključem ovisnim o računalu na kojem se instalira

4) registracijski ključ se provjerava putem Interneta

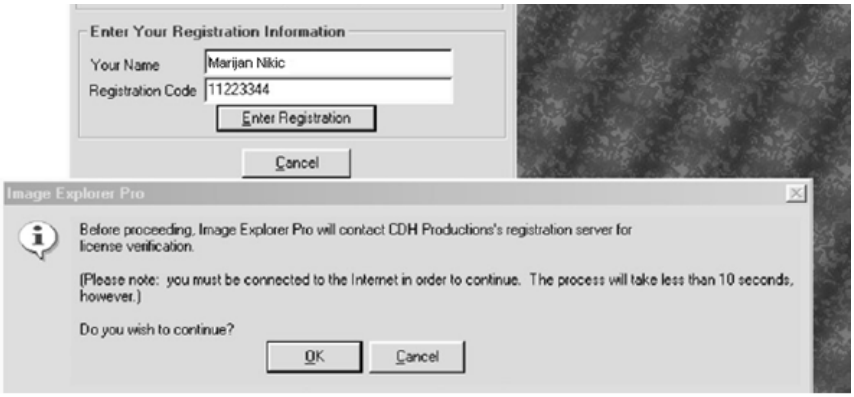
Neki noviji programi koriste ovu moderniju i sofisticiraniju varijantu provjere ispravnosti registracije. Program prikuplja registracijske informacije od korisnika, zatim ih šalje na poslužitelj putem Interneta, poslužitelj provjerava podatke (uspoređuje ih s podacima u bazi podataka), te šalje izvještaj u kojem piše da li su podaci točni ili ne natrag klijentu.

Nešto naprednije od ovoga je konstantna slučajna provjera registracije (što naravno zahtijeva stalnu vezu prema Internetu). Ukoliko program „sazna“ da je ilegalno registriran, šalje poruku s podacima o korisniku proizvođaču programa putem Interneta.

Ovakvu verifikaciju nije uputno koristiti za bilo koji program. Pošto je za ovakvu vrstu provjere potrebna stalna veza prema Internetu, preporučljivo je ovaj tip zaštite koristiti s programima koji koriste Internet vezu u svom radu.

Ovakva zaštita međutim može se probiti ako se može „uvjeriti“ program da je poslužitelj potvrdio registraciju. To bi značilo da osoba koja probija zaštitu recimo presretne komunikaciju između programa i poslužitelja, te programu da lažne informacije. Ispravak ovog nedostatka može biti, recimo, da server uz potvrdu o uspješnoj registraciji šalje i djelić kôda potreban za rad neke enkodirane funkcije programa, ili primjerice ključ za dekodiranje tih određenih zaključanih funkcija.

Ovaj tip zaštite preporučljivo je kombinirati sa zaštitom korištenjem sistemskih informacija (npr. serijski broj tvrdog diska), tako da svako računalo ima svoj registracijski broj, kako se prilikom pokušaja probijanja zaštite ne bi mogao iskoristiti točan registracijski broj za otključavanje zaključane funkcije. Također je dobar način da se, primjerice, ograniči broj upotreba registracijskog ključa na jednu, ili da se ograniči njegova buduća upotreba na IP adresu na kojoj je inicijalno registriran.



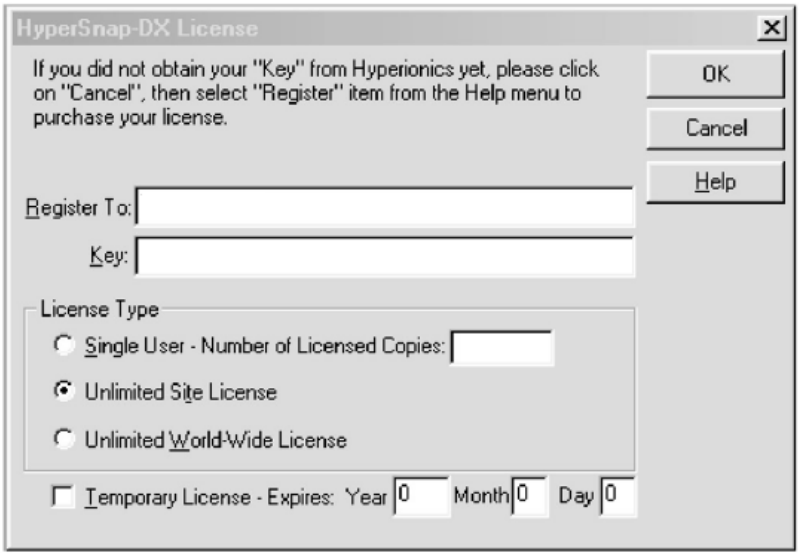
Slika 4. Primjer programa s registracijskim ključem koji se provjerava putem Interneta

2.2.2. Programi s vremenskim ograničenjem

Ovi programi, kako i sam naslov kaže, imaju vremenski rok za korištenje, nakon kojega je korištenje onemogućeno dok se npr. ne unese ispravan registracijski broj. Ovo nije najbolji način zaštite jer ako se uspije probiti vremensko ograničenje, cijeli program postaje dostupan. Bolji je način onesposobiti neke od funkcija neregistriranog programa, tako prisiljavajući korisnika da kupi cijelu i kompletno funkcionalnu verziju. Programi s vremenskim ograničenjem razlikuju se kroz nekoliko implementacija:

1) vremensko ograničenje se uklanja unošenjem ispravnog registracijskog ključa

Ova metoda vrlo je slična prethodno opisanim klasičnim metodama s registracijskim ključem, a jedina je razlika što ovdje dodatno postavljamo vremensko ograničenje za uporabu programa prije unošenja registracijskog broja. Kad istekne taj vremenski rok, program se u potpunosti ne može koristiti dok se ne unese ispravan registracijski broj. Unutar vremenskom roku program se može staviti na korištenje korisniku sa svim funkcijama, ali preporučljivo je zaključati neke funkcije kako bi se korisnika natjeralo da kupi kompletan programski paket, ili se te funkcije mogu otključavati pri unošenju ispravnog registracijskog broja, kako je opisano u prijašnjim odjeljcima. Dio kôda s unosom registracijskog broja ovdje bi trebao biti što pažljivije i kompleksnije izveden, sa što duljim registracijskim ključem, dok se dio s vremenskim ograničenjem može izvesti jednostavnije, primjerice da se zapiše sistemsko vrijeme prve uporabe i onda odbrojava jednostavnim brojačem, jer će se prilikom probijanja kod ovog tipa zaštite više fokusa stavljati na probijanje registracijskog ključa.

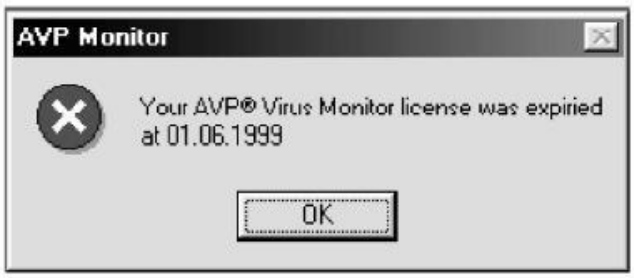


Slika 5. Primjer programa s vremenskim ograničenjem i registracijskim ključem

2) vremensko ograničenje se uklanja unošenjem .REG datoteke

Ovo je jako dobar, ali i poprilično zahtjevan način zaštite računalnih programa. .REG datoteka je datoteka s registracijskim ključem, koja poželjno može sadržavati i dodatan dio kôda za uklanjanje vremenskog ograničenja. Može se slati recimo putem Interneta. Ovakav će se način zaštite najčešće napadati izravnim napadom na vremensko ograničenje, odnosno provjeru isteka vremenskog roka, pa je vrlo važno dobro zaštititi taj dio kôda. Također je uputno izbjegavati metode koje traže .REG datoteku u programskom direktoriju i provjeravaju da li su podaci u njoj ispravni, već

je poželjno zaštićeni dio kôda staviti u samu registracijsku datoteku, što je puno efektivnije, ali i teže za implementirati.



AVP 3.0 Key File■■■■■■■■■■■■■■■■■AD.8■■p■ÐÃ<■■■w■■■CĬX■ű aúb90ö■đxňyěr~ěě2■ëç
Í■ëá■■áÜi%Ů;%Kunn%■b1z+S]■ĚĚ9đ68Ć:;■■

Slika 6. Primjer programa s vremenskim ograničenjem i kriptiranom registracijskom datotekom

3) vremensko ograničenje se ne može ukloniti; korisnik mora kupiti kompletan program

Ova se metoda koristi za demo verzije programa. U takvim programima postoji rok za isprobavanje (eng. *time trial*), a poslije toga se program ne može koristiti, već se može samo naručiti puna verzija.

Kod ovog tipa zaštite fokus kod probijanja biti će na rutini koja provjerava istek vremenskog roka demonstracijske uporabe programa, te pokušaja direktne izmjene tog dijela programskog kôda. Dobar je način zaštite od ove opasnosti korištenje zaštitnog programa koji će pri pokretanju provjeravati da li je bilo izmjena u izvršnoj datoteci izračunavanjem posebne funkcije, tzv. „*checksum*“ (funkcija sažetka (eng. *hash*) koja izračunava iz dane datoteke, prema posebnom algoritmu, broj koji je jedinstven za konkretnu datoteku) izvršne datoteke aplikacije, jer ukoliko je izvršni kôd izmijenjen, ova će funkcija dati krivi rezultat. Ukoliko je rezultat različit od onog što smo trebali dobiti, program izbacuje pogrešku, i pokretanje aplikacije se zaustavlja.

Slika 7. Primjer programa s vremenskim ograničenjem bez mogućnosti uklanjanja vremenskog ograničenja

4) vremensko ograničenje uz određen broj pokretanja

Ovo je u principu jednak način zaštite kao svi dosad navedeni načini zaštite uz vremensko ograničenje, samo što ne odbrojava dane, već broj zasebnih pokretanja programa. To je čak bolji pristup od običnog vremenskog ograničenja, jer otežava posao probijanja pošto ne dohvaća sistemsko vrijeme, već samo pamti broj pokretanja u registrima ili u datoteci. Usto je i implementacijski jednostavniji.

2.2.3. Zaštita pomoću sklopovskog ključa (eng. *dongle*)

Zaštita pomoću sklopovskog ključa rjeđe se koristi, jer je skuplja u odnosu na programske metode zaštite. Sastoji se od toga da se sa svakom kopijom programa pakira i poseban uređaj koji se mora priključiti na neki I/O port na računalu (najčešće USB) kako bi se program mogao pokrenuti.

Princip rada više-manje je jednostavan: program prilikom pokretanja šalje poruke na računalni ulaz na kojem se treba nalaziti sklopovski ključ, i ukoliko ne dobije odgovor s tog ulaza, neće se pokrenuti. Preporučljivo je kriptirati poruke koje šalju program i sklopovski ključ jedan drugome. Nešto bolja izvedba je da se na ključ stavi EPROM čip, na kojem je upisan dio kôda neophodan za rad programa, što čini bilo kakav pokušaj probijanja zaštite gotovo nemogućim, jer je u takvom slučaju (gotovo) nemoguće pokrenuti program bez odgovarajućeg ključa.

Preporučljivo je ne oslanjati se previše na API pozive i programsku podršku proizvođača sklopovskog ključa kad se koristi ovaj način zaštite. Ljudi koji se bave probijanjem zaštite obično su unaprijed upoznati s ovim programskim podrškama i API pozivima, pa je puno bolje koristiti vlastiti programski kôd za ostvarivanje komunikacije sa sklopovskim ključem.

Problem je s ovim načinom zaštite što postoje razni programi imitatori koji glume sklopovske ključeve (eng. *emulator*) za pojedine vrste. Oni imitiraju prisustvo sklopovskog ključa, te mogu prevariti program. Ukoliko ne postoji dostupan imitator, još jedan način probijanja je ubacivanje imitacijskog kôda izravno u program. Ovo se može spriječiti višestrukim CRC provjerama protiv izmjena u programu i u datoteci i u memoriji.

Slika 8. Primjer sklopovskog ključa („Sentinel hardware key“)

2.3. Programi za automatsku zaštitu

Uz programersku zaštitu, koju piše i postavlja sâm programer programa koji se zaštićuje, danas postoji niz komercijalnih programa za automatsku zaštitu (trenutno na tržištu postoji preko stotinu takvih programa dostupno), koji integriraju svoje funkcije sa ciljnom aplikacijom i implementiraju generičku zaštitu, a te funkcije uključuju sve od kompresije ciljnog programa i instalacijskog paketa, kontrolirane dekompresije, enkodiranja zasebnih funkcija ili cijelog programa, automatske zaštite registracijskim ključem, kriptiranja ključa, vremenskih ograničenja, pa do korištenja trikova sprečavanja praćenja izvođenja programa (eng. *antidebugging*) i sprečavanja dohvaćanja asemblerskog kôda iz izvršne datoteke (eng. *antidisassembling*).

Ovakvi programi namijenjeni su programerima koji nisu dovoljno vješti u implementiranju vlastite programske zaštite, ili jednostavno nemaju vremena ili volje truditi se previše oko ovog segmenta razvoja programskog proizvoda. Jasna je stvar da je i dalje najbolja zaštita originalna zaštita isprogramirana od strane programera, uz pretpostavku da je programer dovoljno vješt u području zaštite i da zna što radi (što najčešće nije slučaj). Razlog tomu je što je dosta ljudi iskusnih u probijanju zaštite često upoznato s funkcionalnostima postojećih komercijalnih programa za automatsku zaštitu, pa uz informaciju koji je program korišten za svrhu zaštite, znaju gdje potražiti slabu točku. Također, po Internetu kruže i programi za automatsko probijanje ovakve zaštite, pa i to predstavlja svojevrstan problem. No, opet, kakva god bila situacija, među velikim brojem ovih programa, postoje neki koji, uz pametno korištenje, nude vrhunsku zaštitu koju je iznimno teško probiti. Slijedi nekoliko primjera.

2.3.1. ASProtect

ASProtect je vjerojatno najbolji alat za automatsku zaštitu današnjice. Proizvod je tvrtke ASPack Software. Zamišljen je prvenstveno za rad sa manjim programima internetske distribucije, ali je pogodan i za profesionalne aplikacijske pakete.

Kako ASProtect radi? Prva i osnovna stvar je kompresija i dekompresija originalnog programa prije samog pokretanja. Ovo rade svi programi za automatsku zaštitu, ali ASProtect koristi ASPack algoritam, koji je daleko najbolji na ovom području. Dekompresija služi za otkrivanje da li je bilo pokušaja izmjene originalne datoteke. Ujedno, ovim se sprečava dohvaćanje asemblerskog kôda iz izvršne datoteke (eng. *disassembling*), jer se, logično, izvršni kôd ne može dohvatiti prije dekompresije.

Također, prilikom rada programa, ASProtect sprečava izvođenje nekih funkcija programa prije nego što je program registriran. Korištenje ove funkcionalnosti ASProtecta vrlo je jednostavno – jedino što programer treba napraviti je kriptirati funkciju koju želi onemogućiti pomoću ASProtect algoritma, a ASProtect će se pobrinuti da prilikom registracije otključa tu funkciju pomoću ključa koji je sadržan u registracijskom ključu, tako da nema načina da se zaključane funkcije omoguće bez ispravnog registracijskog ključa. ASProtect koristi veoma snažnu enkripcijsku funkciju, tako da sprečava čak i „sirove“ napade metodom isprobavanja svih mogućih kombinacija (eng. *brute-force*).

2.3.2. InstallShield

Installshield je najpoznatiji i najbolji program za kreaciju instalacijskih paketa. Koristi API pozive iz eksternih biblioteka, što je veoma dobro za postizanje zaštite. Također koristi enkripciju paketa, a često i provjeru sklopovskog ključa.

Installshield koristi svoj vlastiti programski jezik, sličan strukturi programskog jezika C. Prilikom instalacije, sve datoteke potrebne za instalaciju se dekompresiraju u privremeni

direktorij, te se instalacija pokreće samo s tog mjesta, što otežava izmjenu datoteka u samom instalacijskom paketu.

Najbolji dio zaštite Installshielda leži u zaštiti uprogramiranoj u Installshieldov jezik za skriptiranje, jer mu je vrlo teško pratiti izvođenje po kôdu zbog kompleksne i zbunjujuće strukture, a i skript datoteka je zaštićena CRC provjerom, tako da je ovu datoteku jako teško izmijeniti.

2.3.3. Armadillo

Punim imenom Armadillo Software Protection System, komercijalni je alat za programsku zaštitu pod licencom tvrtke Silicon Realms Toolworks. Koristi slučajno generirani ključ za kriptiranje i kompresiju programa.

Jedna od posebnih funkcija Armadilla je da omogućuje programerima unos ilegalnih registracijskih ključeva (dobivenih prijašnjim probijanjima zaštite) u posebnu bazu podataka kako bi se onemogućilo njihovo daljnje korištenje u budućim verzijama programa. Također koristi i posebne trikove za sprečavanje praćenja izvođenja programa kao zaštitu od programa SoftICE (jedan od trenutno najjačih programa za praćenje izvođenja programa – o njemu će više riječi biti kasnije).

Među brojnim funkcijama, Armadillo nudi:

- sklopovsko zaključavanje – korištenje sklopovskog ključa
- zaštitu od vraćanja sistemskog vremena
- registraciju certifikatima
- posebnu zaštitu od probijanja vremenske zaštite

Koliko god je Armadillo dobar program, radi jedan veliki propust. Nakon dekodiranja programa, puni cijeli dekodirani program u memoriju, tako da sve što je potrebno napraviti za uklanjanje Armadillove zaštite je snimiti sadržaj radne memorije (eng. *memory dump*), napraviti nekoliko modifikacija, i program će raditi.

2.3.4. The Slovak Protector (SVKP)

Po sučelju i nekim funkcijama sličan ASProtect-u, SVKP ipak je dosta različit od ostalih programa za automatsku zaštitu. Veliku prednost daje mu činjenica što je pisan u asembleru, što ga čini bržim i omogućava korištenje nekih posebnih programerskih tehnika koje je nemoguće izvesti u nekim višim programskim jezicima.

Kao i ASProtect, SVKP koristi enkripciju pojedinih funkcija, koje se mogu dekodirati samo pomoću ispravnog registracijskog ključa. Za ovu enkripciju SVKP koristi 2048-bitni RSA algoritam, što čini nemogućim generiranje lažnih registracijskih ključeva, jer uz današnju raspoloživu računalnu moć, nemoguće je u smislenom vremenu probiti 2048-bitni ključ.

SVKP ne koristi kompresiju, kako bi se osiguralo brže pokretanje programa. Umjesto toga, zaštićeni program kriptira se pomoću tzv. *Rijndael* algoritma (AES), kako bi se spriječili „sirovi“ napadi.

Od ostalih funkcija, SVKP također nudi API pozive za komunikaciju s vlastitom zaštitom, kao i funkcije za aktivnu zaštitu i sprečavanje praćenja izvođenja.

Možda najzanimljivija funkcija koju implementira SVKP (iako ne u potpunosti) je tzv. metamorfizam. To je tehnologija zaštite novijeg datuma, i radi na principu izmjene zaštite nakon svake uporabe. Naime, činjenica je da je veliki dio zaštićenih programa zaštićeno na slične načine, tako da je sve lakše i brže probijati zaštite iznova i iznova, što je više sličnih zaštita već probijeno. Ova tehnologija pokušava spriječiti taj efekt. SVKP puno lakše

implementira metamorfizam od nekih ostalih programa zbog činjenice da je pisan u asembleru.

3. PROBIJANJE ZAŠTITE

Ponekad je najbolji način obrane – napad. U svakom slučaju, da bi se postigla pravilna i dobra zaštita, potrebno je poznavati način i alat napada. Tako je i sa programskom zaštitom. Stoga se preporuča svakom tko želi što bolje zaštititi svoj program, poznavanje tehnika probijanja, jer je to najbolji način da se nađu slabosti u vlastitom programu i isprave, da ih osobe s namjerom probijanja zaštite ne bi mogle pronaći i iskoristiti. U idućem tekstu opisani su, vrlo kratko, alati koji se koriste u svrhu probijanja zaštite. Dani su i primjeri za svaku skupinu alata, te je na kraju i jedan primjer, recimo, programskog unazadnog inženjerstva (eng. *reverse engineering*) (pravo probijanje zaštite je dosta kompliciranije), ali bit će dovoljan za prikaz osnovnog principa. Važno je napomenuti da alati koji se koriste u probijanju zaštite nisu ilegalni niti su zamišljeni za korištenje u ovakve svrhe, ali uz funkcije koje nude, vrijedna su pomagala za shvaćanje načina rada programa. Osvrt na legalnost probijanja zaštite i raznih njezinih segmenata i posljedica nalazi se u kasnijem tekstu.

3.1. Alati za probijanje zaštite

3.1.1. Programi za dohvaćanje asemblerskog kôda (eng. *disassembler*)

Disasembleri su računalni programi koji vraćaju unazad rad asemblera, odnosno iz strojnog kôda dohvaćaju kôd pisan u asembleru. Oni rade gotovo identično programima za dohvaćanje izvornog kôda (eng. *decompiler*), ali za razliku od njih, koji vraćaju kôd u nekom od jezika više razine, disasembleri vraćaju kôd na asemblerskom nivou. Ovo olakšava probijanje zaštite, jer osoba koja to radi ne mora poznavati neki viši jezik u kojem je pisan dotični program, već samo asembler, koji dopušta ujedno i puno bolji uvid u funkcionalnost programa.

Treba napomenuti da dobiveni izlaz iz programa za dohvaćanje asemblerskog kôda nije pogodan kao ulaz za asembler, već je više štura verzija originalnog asemblerskog kôda, formatirana često da bude čitljivija čovjeku. Neki od ovih programa dodatno komentiraju izlazni asemblerski kôd radi lakšeg razumijevanja. Uz njih se obično veže i interaktivni program za praćenje izvođenja programa po linijama kôda (eng. *debugger*).

Postoji niz dostupnih disasemblerskih alata na tržištu, što besplatnih, što profesionalnih, komercijalnih izdanja, među kojima su najpoznatiji:

- **IDA (Interactive Disassembler)** – IDA je najpoznatiji i najraširenij komercijalni disasembler. Razvio ga je Ilfak Guilfanov, sistemski arhitekt i programer belgijske tvrtke DataRescue, kao „*shareware*“ program, a kasnije je tvrtka komercijalizirala njegov proizvod, koji se danas prodaje pod nazivom IDA Pro. IDA Pro podržava veliki niz operativnih sustava (x86 Windows, MacOS, Linux, ...), formata izvršnog koda (PE, ELF, Netware, OS/2, ...) i procesora (Intel 80x86, ARM, Motorola, Zilog, ...). Odlikuje se svojom interaktivnošću.
- **ILDASM (Intermediate Language Disassembler)** – ILDASM je alat sadržan u SDK-u Microsoftovog .NET Framework-a. Koristi se za dobivanje asemblerskog kôda iz PE izvršnih datoteka dobivenih iz Microsoft-ovog CIL (Common Intermediate Language) programskog jezika. (CIL je u stvari najniži nivo programskog jezika .NET arhitekture. Sve što se prevede u .NET Framework-u, prevodi se u CIL, a zatim se iz CIL-a asemblira u binarnu izvršnu datoteku prilikom pokretanja programa na virtualnoj platformi .NET-a.)

- **WinDASM (Windows 16/32-bit Disassembler)** – Uz IDA Pro, WinDASM je najbolji disasembler u Windows okruženju. Jednostavniji je od IDA-e i daje nešto kraći izlazni kôd. Nažalost, za razliku od IDA-e, ovaj se program više ne razvija, i može se jedino ponegdje pronaći na Internetu dostupan za skidanje kao „*shareware*“ program.

3.1.2. Programi za praćenje izvođenja programa po izvornom kôdu (eng. *debugger*)

Ovi računalni programi koriste se za testiranje i rješavanje problema kod programa. U suštini, rade tako da prilikom greške u izvođenju programa ukazuju na točno mjesto u kôdu gdje je došlo do pogreške i stanje memorije/stôga, vrijednosti varijabli, itd. Postoje neki koji rade na nivou izvornog kôda (tzv. simbolički *debugger*-i) – služe najčešće za više programske jezike; oni će prilikom pogreške u radu programa pokazati na točnu liniju izvornog kôda gdje je došlo do pogreške. Druga vrsta su programi za praćenje izvođenja na niskom nivou (eng. *low-level debugger*) koji rade nad strojnim kôdom, pa će oni koristiti disasemblirani kôd za praćenje mjesta nastanka greške u programu (ovakvi će se programi za praćenje u probijanju zaštite najčešće koristiti). U što je višem programskom jeziku program napisan, teže će biti pratiti njegov tijek pomoću programa za praćenje, jer će kôd dobiven vraćanjem asemblera biti netočniji, ali je zato dosta teže izvesti kvalitetnu zaštitu, ako se radi u višim programskim jezicima.

Programi za praćenje naravno ne služe samo za otkrivanje mjesta pogreške u programu; današnji takvi programi nude mogućnosti praćenja izvođenja programa korak po korak, praćenja promjena vrijednosti varijabli u stvarnom vremenu, praćenja vrijednosti u registrima, stanja na stogu itd., pa čak i izravnu promjenu vrijednosti bilo koje varijable tijekom izvođenja programa radi testiranja daljnjeg ponašanja programa s izmijenjenim vrijednostima varijabli.

Neki od najpoznatijih i najraširenijih programa za praćenje izvođenja su:

- **SoftICE** (ICE = „in-circuit emulator“) – Najbolji program za praćenje izvođenja svih vremena u DOS okruženju, danas radi jako dobro na svim verzijama Windows-a. Glavna stvar kod SoftICE-a je da radi u jezgrenom modu (eng. *kernel mode*), što znači da može pratiti sve programe što su pokrenuti u dotičnom operativnom sustavu, pa čak i servise i sklopovsku programsku podršku koji se nalaze u zaštitnom prstenu 0 operativnog sustava. Koristi visoku interakciju s operativnim sustavom, pa omogućuje korisniku dubinski uvid u interne mehanizme OS-a. SoftICE je proizvod tvrtke NuMega, a kasnije je razvoj preuzela tvrtka Compuware.
- **TRW2000** je program za praćenje s nešto siromašnijim opcijama od SoftICE-a, ali je ujedno i jednostavniji i intuitivniji. Radi na Windows 9x sustavima, i to, poput SoftICE-a, također u jezgrenom modu. Dostupan je na raznim Internet stranicama za skidanje distributabilnih programa.
- **OllyDbg** je 32-bitni program za praćenje izvođenja kroz asembler za Windows okruženja, s vrlo intuitivnim sučeljem. Preporučljiv je zbog svoje jednostavnosti u odnosu na SoftICE. Usto je i besplatan za osobnu i komercijalnu upotrebu.

Navedeni programi rade uz naglasak na analizu binarnog kôda. Oni se koriste uglavnom za istraživanje rada programa za koje nemamo priložen izvorni kôd. Uz njih, svako razvojno okruženje uključuje i alat za praćenje izvođenja programa po linijama izvornog kôda u dotičnom programskom jeziku (*Visual Studio*, *Java Platform*, *Eclipse*, *Turbo Debugger*, ...).

Također i većina programa za dohvaćanje asemblerskog kôda dolazi u paketu s nekakvim programom za praćenje izvođenja, koji se doduše ne može po funkcijama mjeriti s

profesionalnim samostalnim programima, no za osnovne svrhe praćenja izvođenja, posebno kad se radi o asemblerskom kôdu, dovoljno je imati dobar disasembler.

3.1.3. Programi za dohvaćanje izvornog kôda (eng. *decompiler*)

Programi za dohvaćanje izvornog kôda su programi koji, logično, rade suprotan proces od prevodioca izvornog kôda (eng. *compiler*). Oni uzimaju kao ulaz nekakav kôd na jako niskom nivou apstrakcije (uglavnom strojni ili asemblerski kôd), te ga pretvaraju u programski kôd više razine apstrakcije, koji je čitljiviji čovjeku. Dobar program za dohvaćanje izvornog kôda iz danog izvršnog programa dat će vrlo precizan kôd na visokom nivou apstrakcije, tako da, kad bismo predali taj dobiveni kôd prevodiocu, rezultat bi bio identičan izvršni program kakav smo imali na početku.

Ovi programi pružaju detaljan uvid u način na koji određeni program funkcionira, ali da bi se moglo dohvatiti izvorni kôd nekog programa, potrebno je znati u kojem je jeziku taj program inicijalno pisan, odnosno s kakvim je prevodiocem preveden. Za svaki viši programski jezik postoji poseban program za dohvaćanje izvornog kôda. Usto, trebalo bi naravno znati i baratati tim jezikom u kojem je pisan program čiji se izvorni kôd dohvaća, kako bismo dobiveni kôd znali pročitati.

Kvaliteta dobivenog izvornog kôda ovisit će uvelike o količini informacija koje program za dohvaćanje izvornog kôda može izvući iz izvršnog kôda. Mnoge virtualne mašine (*Java VM*, *.NET Framework CLR*) u svojim oktetnim formatima koriste pozamašne metapodatke koji omogućuju visoku izvedivost dekompilacije. Strojni jezik ima, razumljivo, puno manje metapodataka, pa ga je zbog toga puno teže koristiti za dekompilaciju.

Dekompilacija će ovisiti prvenstveno o jeziku u kojem je izvorno pisan program za koji dohvaćamo izvorni kôd, pa će tako o tome ovisiti i izbor samog programa koji ćemo koristiti za tu svrhu. Postoji pozamašan broj komercijalnih programa za dohvaćanje izvornog kôda za razne programske jezike; ovdje je nekoliko primjera na koje sam naišao na Internetu:

- **VB Decompiler** – Kao što mu samo ime kaže, VB Decompiler je program za dohvaćanje Visual Basic izvornog kôda, za aplikacije (EXE, DLL, OCX) pisane u programskom jeziku Visual Basic, verzije 5.0/6.0. Proizvodi ga tvrtka Gpch Software.
- **Dis# .NET Decompiler** – Program za dohvaćanje izvornog kôda Microsoft .NET aplikacija. Omogućava editiranje lokalnih varijabli i imena, te spremanje istih u projektnu datoteku. Dobiveni kôd sličan je izvornom (ali ne isti).
- **DJ Java Decompiler** – Program za dohvaćanje izvornog kôda programskog jezika Java i disasembler za Windowse; uzima prevedene .CLASS datoteke, te iz njih izvlači izvorni Java kôd. Vrlo je dobar i precizan čak i s kompleksnim Java *applet*-ima.

3.1.4. Heksadecimalni editori

Heksadecimalni (ili skraćeno heks) editor je vrsta programa koja omogućava korisniku manipulaciju binarnim datotekama. Korištenjem heks editora, korisnik može vidjeti ili editirati sirovi i egzaktni sadržaj datoteke za razliku od interpretacije datoteke preko aplikacijskog programa više razine, koji predviđeno čita takav format datoteka. Primjerice, ovdje se može raditi o sirovim podacima neke JPEG slikovne datoteke (u oktetima) u odnosu na način na koji JPEG sliku prikazuje nekakav aplikacijski program (recimo preglednik slika, u ovom slučaju) poput IrfanView ili ACDSee-a.

Za izvršne datoteke ovo znači da će nam heks editor omogućiti uvid i izmjenu binarnog zapisa naredbi procesoru za izvršavanje programa. Izvršna datoteka sastoji se od niza binarnih naredbi (strojnih riječi) posloženih po memorijskim lokacijama – svakoj naredbi u asembleru odgovara određeni binarni kôd (eng. *opcode = operation code*). Kada dobijemo

asemblerski kôd programa (izlaz iz disasemblera), vidjet ćemo memorijske adrese, binarne kodôve operacija i asemblerski zapis naredbi procesoru, u odgovarajućem redoslijedu (po memorijskim lokacijama). Kada pronađemo odgovarajuću naredbu koju treba promijeniti, u heks editoru treba pronaći odgovarajući binarni kôd i izmijeniti bitove na odgovarajućoj memorijskoj lokaciji – to je osnovna upotreba heks editora u svrhe probijanja zaštite.

U većini heks editora datotečni podaci prikazani su kao heksadecimalne vrijednosti grupirane u 2 stupca od po 8 okteta i 1 stupac od 16 ASCII znakova u svakom retku, među kojima se s „.“ označavaju neprintabilni znakovi.

Među heks editorima nema nekih velikih razlika, osim u licenci, operativnom sustavu i intuitivnosti sučelja; evo nekoliko popularnijih:

- **Hiew (Hacker's View)** – Ponajbolji heks editor za DOS okruženja. Licenca za ovaj program je besplatna za nekomercijalnu upotrebu.
- **XVI32** – Potpuno besplatan heks editor za Windows 9x i Windows NT okruženja.
- **010 Editor** – Profesionalan tekstualni i heks editor s intuitivnim sučeljem i mnoštvom opcija.
- **Hex Workshop** – Profesionalan heks editor tvrtke Breakpoint Software Inc. Pruža mnoštvo opcija, uključujući i izvoz u RTF/HTML format.
- **HEdit** – Vrlo jednostavan početnički alat za osnovne radnje editiranja binarnih datoteka.

3.2. Primjer probijanja zaštite – crackme0.asm

Radi demonstracije svega dosad spomenutog, evo jednog, najjednostavnijeg mogućeg, primjera – riječ je o programu pisanom u asembleru za 32-bitne Intelove procesore. Cijeli opisani proces uključivat će asembliranje s programom „masm32“ (Microsoft's Assembler), pa potom disasembliranje pomoću programa „WinDasm“ (Windows 16/32 Disassembler), pronalaženje kritičnog dijela u programu – točke gdje se provjerava registracija programa, te konačno izmjene odgovarajućih instrukcija u izvršnom kodu, odnosno konačno „probijanje zaštite“ (izmjena izvršne datoteke) pomoću heks editora.

3.2.1. Prevođenje

Ovo je prikaz asemblerskog kôda programa koji ćemo probiti:

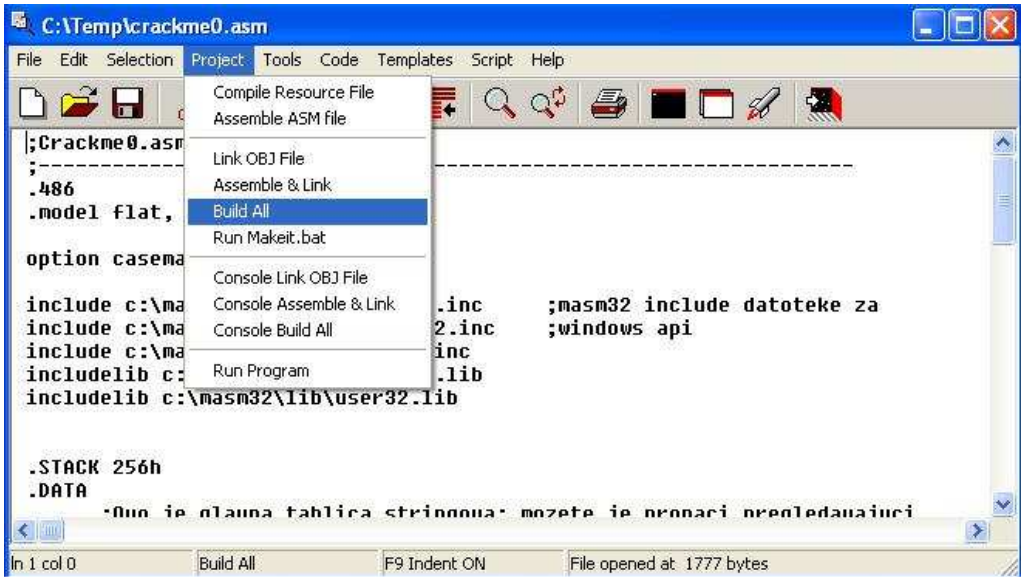
```
;Crackme0.asm
;------------------------------------------------------------------
.486
.model flat, stdcall

option casemap:none

include c:\masm32\include\windows.inc    ;masm32 include datoteke za
include c:\masm32\include\kernel32.inc   ;windows api
include c:\masm32\include\user32.inc
includelib c:\masm32\lib\kernel32.lib
includelib c:\masm32\lib\user32.lib


.STACK 256h
.DATA
      ;Ovo je glavna tablica stringova; mozete je pronaci pregledavajuci
      ;program pomocu heks editora
```

```
      DLGCAP    db      "Crackme0",0
      REG       db      "Program je registriran",0
      UNREG     db      "Program nije registriran",0
      ISREG     dd 0    ;ovo je globalna varijabla koja nam govori da li
                        ;je program registriran ili ne

.CODE


;Glavna funkcija – slucajno nazvana Main (moze se zvati bilo kako)

Main PROC
    mov     eax, ISREG        ;u eax registar stavljamo vrijednost globalne
                              ;varijable za provjeru registracije
    cmp     eax, 0            ;Usporedba – da li je program registriran?
    jne     prog_registered   ;if(isreg != 0) goto prog_registered
    jmp     prog_unregistered ;else goto prog_unregistered

prog_registered:
    ;ovdje se na stog šalju parametri za MessageBox, a zatim se MessageBox
    ;i zove
    push    MB_OK
    push    offset DLGCAP
    push    offset REG        ;string obavijesti o uspjesnoj registraciji
    push    0
    call    MessageBoxA
    jmp     theend

prog_unregistered:
    push    MB_OK
    push    offset DLGCAP
    push    offset UNREG      ;string obavijesti o neuspjesnoj registraciji
    push    0
    call    MessageBoxA
    jmp     theend

theend:
    push    0                 ;zavrsi program
    call    ExitProcess

Main ENDP
End Main
;-------------------------------------------------------------------
```

Ovo je jedan vrlo jednostavan program koji u *„Hello world!“* maniri imitira registracijsku proceduru. Naime, sve što on koristi je jedna globalna varijabla, ustvari konstanta upisana u lokaciju u memoriji, i ona je postavljena na 0. Ova konstanta govorit će nam da li je program registriran. (Mijenjanje ove konstante nije potrebno za uspješno probijanje ovog programa.)

Glavni dio programa odmah je na početku glavne funkcije, gdje uspoređujemo vrijednost registra *eax* (u koji smo prethodno upisali vrijednost varijable `ISREG`) s nulom; ako vrijednost u *eax* registru nije jednaka nuli (`jne` naredba – eng. ***j**ump if **n**ot **e**qual*), program skače na labelu `prog_registered`, otvorit će se okvir poruke (*MessageBox*) u kojem piše tekst *„Program je registriran“*, što je zapisano u znakovnoj konstanti `REG`; u suprotnom, program skače na labelu `prog_unregistered`, te se u *MessageBox*-u ispisuje *„Program nije registriran“*, što je zapisano u znakovnoj konstanti `UNREG`.

Ovaj program prvo ćemo provući kroz asembler „masm32“ (***M**icrosoft's Macro **Ass**e**m**bler*) kako bi dobili binarnu izvršnu datoteku, iz koje ćemo poslije izvući asemblerski kôd pomoću odgovarajućeg programa. Iz izlaza tog programa ćemo zatim iščitati naredbe koje moramo promijeniti kako bi program „bio registriran“, te potrebne izmjene provesti u heks editoru.

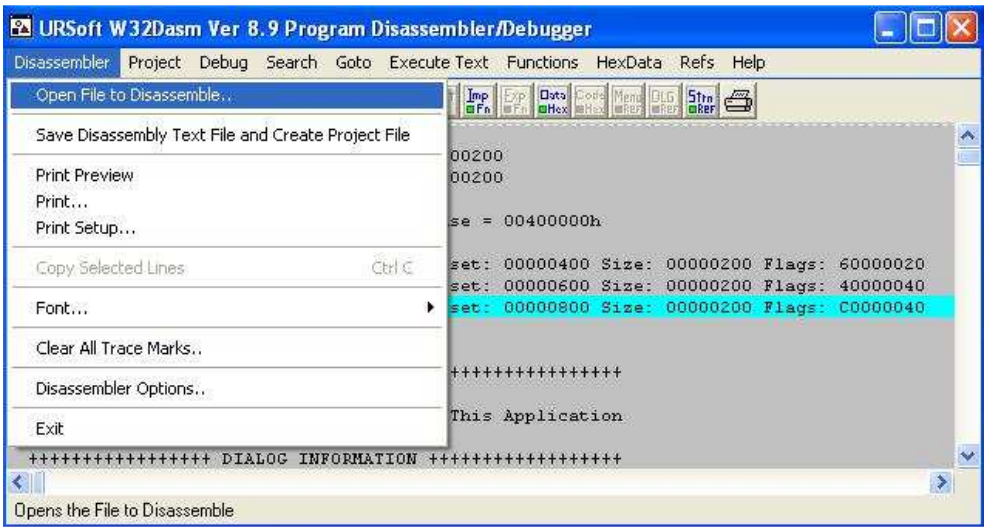


Slika 9. MASM Editor – Proces prevođenja datoteke „crackme0.asm“

Slika 10. Nakon prevođenja, pokretanjem programa dobivamo gornju poruku

3.2.2. Dohvaćanje asemblerskog iz izvršnog kôda

Prevođenjem smo dobili izvršnu datoteku „crackme0.exe“. Iz te datoteke dohvatit ćemo asemblerski kôd pomoću programa „Win32DASM“ (**Win**dows **32**/16-bit **D**is**a**sse**m**bler) tvrtke UrSoftware.



Slika 11. W32Dasm – Disasembliranje

WinDasm prvo ispisuje osnovne informacije poput raznih memorijskih odmaka (eng. *offset*), predefiniranih memorijskih lokacija, te objekata od kojih se dobiveni program u asembleru sastoji (naredbe, razni podaci (konstante), uvozi (iz eksternih DLL datoteka i sl.)). Pa krenimo redom:

```
Disassembly of File: crackme0.exe
Code Offset = 00000400, Code Size = 00000200
Data Offset = 00000800, Data Size = 00000200
```

- *Code offset* – relativna memorijska lokacija početka programskog kôda u odnosu na memorijsku lokaciju početka izvršne datoteke.
- *Data offset* – relativna memorijska lokacija početka podatkovnog segmenta programa (*string, integer* i sl. – programske konstante) u odnosu na memorijsku lokaciju početka izvršne datoteke.

```
Number of Objects = 0003 (dec), Imagebase = 00400000h

   Object01: .text    RVA: 00001000 Offset: 00000400 Size: 00000200 Flags: 60000020
   Object02: .rdata   RVA: 00002000 Offset: 00000600 Size: 00000200 Flags: 40000040
   Object03: .data    RVA: 00003000 Offset: 00000800 Size: 00000200 Flags: C0000040
```

- *Number of objects* – broj objekata sadržanih u programu (riječ *dec* znači dekadsku bazu).
- *Imagebase* – adresa u memoriji na koju će se program pokušati upisati nakon pokretanja; ukoliko već ne postoji nešto na toj adresi, Windowsi će mu to dopustiti.

Slijedi opis pojedinačnih objekata u programu:

- *.text* – programski kôd
- *.rdata* – informacije o uvozima
- *.data* – podaci (programske konstante)

Za svaki objekt ispisani su i neki parametri:

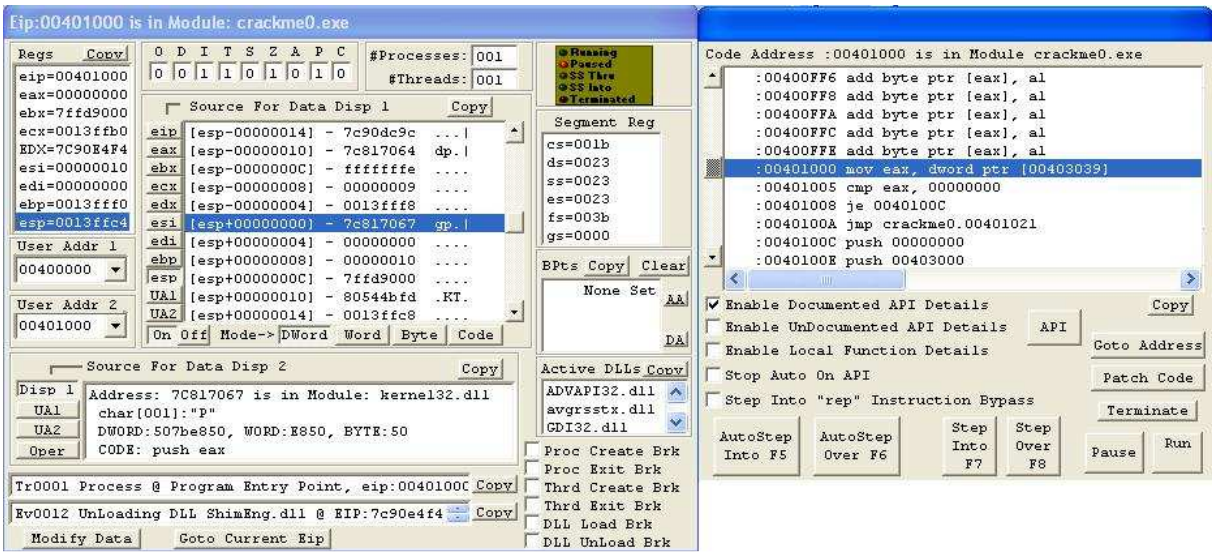
- *RVA* – memorijski odmak početka ovog objekta od memorijske lokacije početka programa (ovaj odmak dodaje se na *Imagebase* da bi se dobila točna memorijska lokacija konkretnog dijela programa (odnosno objekta)).
- *Offset* – prikazuje memorijski odmak konkretnog objekta unutar same *.exe* datoteke.
- *Size* – veličina konkretnog objekta u memoriji (u oktetima).

Nakon tih uvodnih informacija, WinDasm je ispisao i sam kôd, i to ne samo u obliku asemblerskih naredbi, već i kao heksadecimalni ispis binarnih kôdova naredbi programa (eng. *opcodes*). Primjerice naredba `jne 0040100C` ima kôd 7502 – ovdje je 75 kôd same naredbe `jne`, a 02 je relativni odmak mjesta u memoriji na koje program od ovog mjesta skače (ukoliko je ispunjen uvjet „*not equal*“) – dakle na naredbu koja je u memoriji druga iza dotične naredbe `jne` (programsko brojilo (eng. *PC = Program Counter*) se uvećava za 2). Ovdje je važno primijetiti da WinDasm neće dati nazive labela onako kako su one definirane u izvornom kôdu, već će ih pretvoriti u relativne adrese ili odmake u memoriji, za odredišne lokacije skokova, ili na memorijske lokacije u koje se piše, odnosno s kojih se čita. Razlog tomu je što nam labele kod pisanja u asembleru služe za lakše snalaženje u kôdu, a pri prevođenju kôda u izvršni program, u prvom prolazu asemblera, labele se zamjenjuju relativnim adresama, tako da se prilikom asembliranja nazivi labela nepovratno gube. Stoga je

dobiveni kôd malo manje preglednosti nego izvorni kôd, no i preglednost će ovisiti o kvaliteti programa za dohvaćanje asemblerskog kôda. Naime, WinDasm nam uz asemblerski kôd daje i dodatna (moguća) objašnjenja pojedinih dijelova kôda, a neki napredniji disasembleri generiraju i vlastite labele prilikom procesa dohvaćanja da bi kôd bio što bliži izvornom, i da bi se moglo što lakše snalaziti u njemu, no to ponekad nije baš najbolje izvedeno.

```
//******************** Program Entry Point ********
:00401000 A139304000              mov eax, dword ptr [00403039]
:00401005 83F800                  cmp eax, 00000000
:00401008 7402                    je 0040100C
:0040100A EB15                    jmp 00401021
:0040100C 6A00                    push 00000000

* Possible StringData Ref from Data Obj ->"Crackme0"
                                  |
:0040100E 6800304000              push 00403000

* Possible StringData Ref from Data Obj ->"Program je registriran"
                                  |
:00401013 6809304000              push 00403009
:00401018 6A00                    push 00000000

* Reference To: user32.MessageBoxA, Ord:019Dh
                                  |
:0040101A E82B000000              Call 0040104A
:0040101F EB15                    jmp 00401036
:00401021 6A00                    push 00000000

* Possible StringData Ref from Data Obj ->"Crackme0"
                                  |
:00401023 6800304000              push 00403000

* Possible StringData Ref from Data Obj ->"Program nije registriran"
                                  |
:00401028 6820304000              push 00403020
:0040102D 6A00                    push 00000000
```

Slika 12. Prikaz (dijela) dobivenog kôda u W32DASM disasembleru



Slika 13. Win32Dasm Debugger

Kod probijanja zaštite najbitnije je pronaći odgovarajući skok – koliko god se kôda „trošilo“ na provjeru i usporedbe registracijskih ključeva i slično, kad-tad će se morati obaviti skok na potvrdu uspješne (ili neuspješne) registracije i nastaviti s normalnim izvođenjem programa, ili se vratiti na traženje registracijskog ključa, izaći iz programa ili već nešto, ukoliko registracija nije uspješno obavljena. Pošto je ovo jednostavan program, već je na

početku uočljivo da postoje svega 2 moguća pravca izvođenja programa (kod prvog skoka) – ukoliko usporedba *eax* registra s nulom nije istinita, preskačemo iduću naredbu bezuvjetnog skoka i ako pratimo dalje izvođenje programa (ovdje to možemo i izravnim čitanjem kôda; za bilo što složenije, logično bi bilo koristiti program za praćenje izvođenja), vidimo prvo da se nešto stavlja na stog (naredbom `push`); ovo nam nije toliko bitno, to su podaci potrebni za prikaz *MessageBox*-a – dakle reference na memorijske lokacije na kojima se nalaze znakovni nizovi „*`Crackme0`*“ (naslov *MessageBox*-a), te znakovni niz „*`Program nije registriran.`*“, koji se ispisuje kao poruka u *MessageBox*-u. Ovo nam sve, uostalom, javlja WinDasm s porukama „*Possible string data...*“ i „*Reference to...*“. Iz ovoga je jasno što nam je činiti da bismo „probili zaštitu“ ovog programa – nekako spriječiti da se ovaj skok izvrši, kako bi program otišao u alternativnom pravcu.

Postoji više načina da ovo napravimo. Možemo:

- promijeniti usporedbu: naredbu „`CMP eax, 0`“ možemo zamijeniti naredbom „`CMP eax, 0FF`“ ili nešto slično

- promijeniti lokaciju skoka: „`JNE 40100C`“ zamijeniti s „`JNE 401021`“

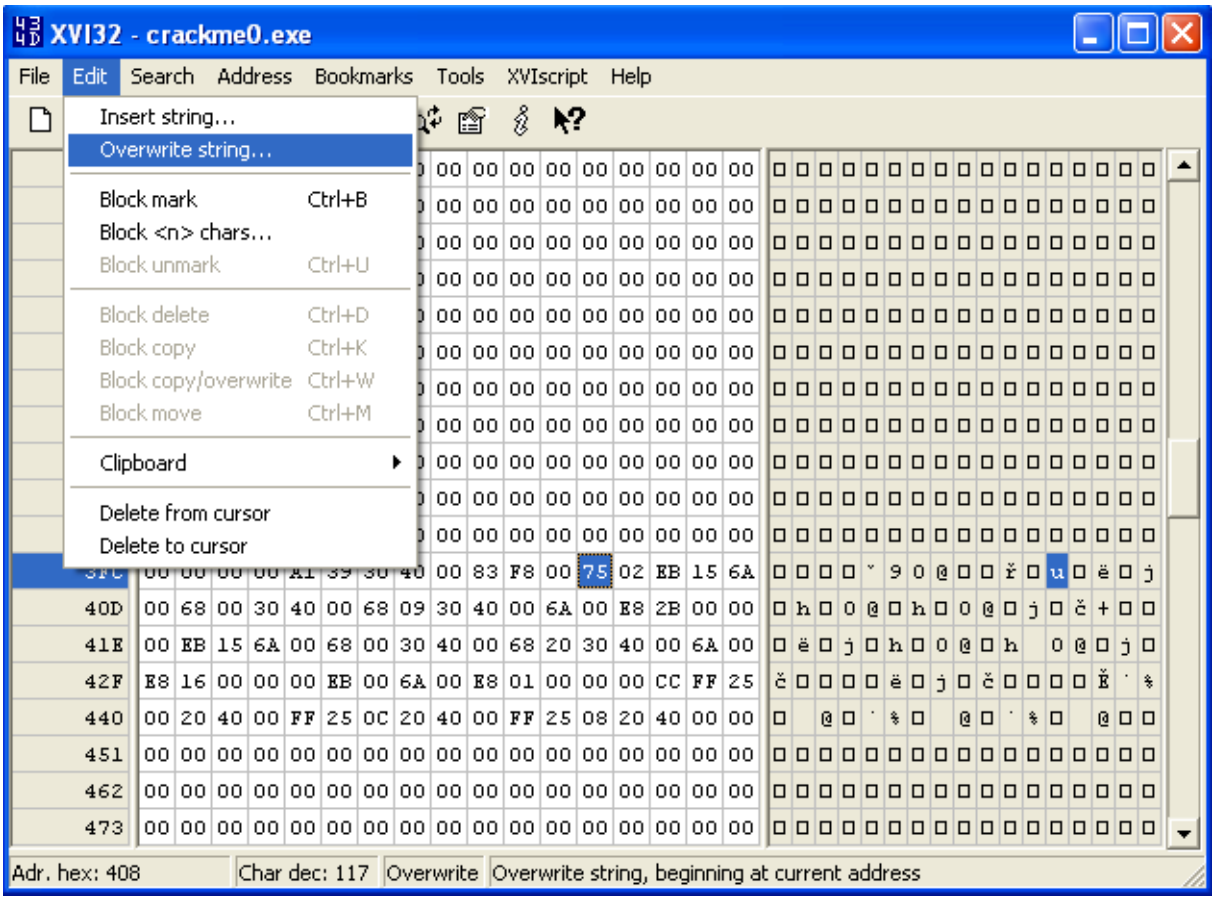
- konačno, promijeniti i samu naredbu skoka: zamijeniti `JNE` s `JE`

Ovu izmjenu na najnižem (binarnom) nivou kôda obavit ćemo pomoću heks editora.

3.2.3. Izmjena izvršnog kôda programa (eng. *patching*)

Sad kad znamo što trebamo promijeniti, potrebno je odgovarajuće mjesto u kôdu pronaći u heks editoru, kako bismo mogli izmjeniti kôd. Za ovaj primjer koristit ću XVI32 heks editor.

Kod većih programa teško ćemo pronaći odgovarajuće mjesto u kôdu upisivanjem sekvence samo jedne naredbe u pretragu, jer je velika vjerojatnost da se takvih naredbi u kôdu nalazi više od jedne. Da ne bi bilo dvojbe uzet ćemo uzorak od više susjednih naredbi, u ovom slučaju uzet ćemo kôdove naredbi „`CMP eax 0`“ i „`JNE 40100C`“ zajedno, što odgovara heksadecimalnom nizu „83F8007502“ (za kôd normalne duljine uzorak bi morao biti 10 i više naredbi u slijedu). Recimo da smo izabrali promijeniti `JNE` naredbu u `JE` naredbu, dakle morat ćemo promijeniti kôd 75 u nešto drugo. Za ovo ćemo morati potražiti referencu naredbe `JE` u listi operacijskih kodôva u MASM-ovoj „*opcodes.hlp*“ pomoćnoj datoteci (*MASM Editor* → *Help* → *Opcodes help*). Operacijski kôd naredbe `JE` je 74, pa ćemo kôd 75 u heksadecimalnom editoru promijeniti u 74. Nakon izmjene sačuvamo promijenjenu .EXE datoteku. Sada, ako probamo pokrenuti ovu datoteku, primjećujemo da se više ne ispisuje tekst „*Program nije registriran.*“, već „*Program je registriran.*“. Time je „probijanje zaštite“ završeno.



Slika 14. Izgled kôda „crackme0.exe“ u XVI32 hex editoru; označeno mjesto je mjesto kôda koji je u ovom primjeru potrebno promijeniti

Slika 15. Nakon cijelog postupka, pokretanjem programa „crackme0.exe“ dobivamo ovo

4. TEHNIKE PREVENCIJE POTENCIJALNIH POKUŠAJA PROBIJANJA (eng. *ANTI-CRACKING*)

Malo je programera svjesno činjenice koliko je bitno, uz samu programsku zaštitu, provesti adekvatnu zaštitu od pokušaja praćenja izvođenja programa i dohvaćanja asemblerskog kôda. Ako netko s namjerom probijanja zaštite može doći do asemblerskog kôda i pratiti izvođenje programa, onda će kad-tad, na lakši ili teži način, pronaći i shvatiti i metodu zaštite, a zatim i probiti zaštitu programa. Srećom, čak i uz vrlo jednostavne trikove, može se iznimno otežati, ako ne i spriječiti, praćenje izvođenja i dohvaćanje asemblerskog kôda programa.

4.1. Prevencija praćenja izvođenja programa

Tehnike prevencije praćenja izvođenja programa odnose se na otkrivanje prisustva programa za praćenje u okviru operativnom sustavu, i sprečavanja pokretanja programa u takvom slučaju. To bi podrazumijevalo testiranje prilikom samog početka pokretanja programa. Ovo prvo testiranje često je relativno jednostavno ukloniti, pa je preporučljivo postaviti testiranje prisustva programa za praćenje i kasnije, na nekom slučajnom mjestu tijekom izvođenja programa.

Zbog činjenice da su programi za praćenje u korelaciji s jezgrom operativnog sustava, trikovi prevencije praćenja izvođenja ovisit će o operativnom sustavu na kojem se program izvodi, jer će se za otkrivanje prisustva takvog programa koristiti uglavnom API pozivi sustava. Tako će se prevencija praćenja izvođenja različito izvoditi na Win9x i WinNT sustavima. Ovdje ćemo se koncentrirati na WinNT sustave, kakvi su Windows XP i Windows Vista.

Evo nekoliko trikova:

1) **kernel32!IsDebuggerPresent**
   API poziv funkcije *IsDebuggerPresent* vraća 1 ako je izvođenje procesa praćeno, a 0 ako nije. Ovaj API jednostavno čita zastavicu *PEB!IsDebugged*. (PEB je *Process Environment Block* za trenutni proces.) Ovaj trik može se zaobići postavljanjem zastavice *IsDebugged* na 0.
2) **PEB!NtGlobalFlags**
   NtGlobalFlags su zastavice koje određuju ponašanje procesa prema raznim API-jima prilikom njegova stvaranja. Ove zastavice nalaze se u polju na odmaku 0x68 u PEB-u. Ako se proces prati, postavljene će biti neke zastavice za kontrolu stoga: FLG_HEAP_ENABLE_TAIL_CHECK, FLG_HEAP_ENABLE_FREE_CHECK, FLG_HEAP_VALIDATE_PARAMETERS. Ovo se može zaobići resetiranjem *NtGlobalFlags* polja.
3) **NtQueryInformationProcess**
   Ako ovu API funkciju pozovemo tako da prije toga upišemo 7 u *ProcessDebugPort* konstantu, sustav će u *ProcessInformation* vratiti -1 ukoliko se proces prati. Ovo je vrlo jak trik za prevenciju praćenja jer se ne može zaobići. *ProcessInformation* može se mijenjati tek nakon povratka iz sistemskog poziva. Na isti način djeluje i API funkcija *kernel32!CheckRemoteDebuggerPresent*. Ona interno poziva funkciju *NtQueryInformationProcess*.
4) **Samopraćenje**
   Proces može otkriti da je praćen i ako sam pokrene praćenje svog izvođenja, stvaranjem novog procesa, i pozivanjem API funkcije

kernel32!DebugActiveProcess(pid), koja poziva API funkciju *ntdll!DbgUiDebugActiveProcess*, a ova opet poziva sistemski poziv *ZwDebugActiveProcess*. Poanta je, ako se proces već prati, onda se ovaj zadnji poziv neće moći izvršiti, pa se tako neće moći izvršiti vanjsko praćenje izvođenja.

5) **INT 3**
   Ovo je jeftini trik za zavaravanje slabijih programa za praćenje izvođenja. Poziv prekida `INT 3` će u izvođenju programa kontrolu izvođenja predati procesu za obradu iznimki (eng. *Exception Handler*). Sa druge strane, programi za praćenje koriste ovu adresu prekida za označavanje prekidnih točaka u praćenju programa. Ako se izvođenje procesa zaista prati, `INT 3` bi mogao zavarati program za praćenje da pomisli da je to jedna od njegovih prekidnih točaka, pa tako proces za obradu iznimki neće dobiti kontrolu, i izvođenje programa će se nastaviti drugačije. Da bi se ova zaštita zaobišla, treba samo koristiti pametan program za praćenje izvođenja koji pamti gdje je postavio svoje prekidne točke.
6) **Vremenske oznake**
   Za prevenciju praćenja izvođenja moguće je također koristiti brojače visoke preciznosti, koji broje cikluse procesora od starta programa. Njih se pokreće instrukcijom `RDTSC` asembleru. Ako je razmak između dvije instrukcije `RDTSC` prevelik, to može značiti da se proces prati (odnosno da je pod kontrolom programa za praćenje). Ovo se temelji na činjenici da se puno procesorskog vremena troši procesiranjem iznimaka (eng. *exception*) u programu za praćenje i vraćanjem kontrole programu čije se izvođenje prati.

4.2. Tehnike prevencije dohvaćanja asemblerskog kôda

Za razliku od prevencije praćenja izvođenja programa, dohvaćanje asemblerskog kôda se baš neće moći spriječiti, jer disasembler čita binarni zapis datoteke, adresu po adresu, instrukciju po instrukciju, te preko operacijskih kôdova naredbi vraća asemblerske naredbe. Umjesto toga, trik će biti zbuniti disasembler, napraviti kôd što nečitljivijim ili što težim za razumijevanje. Primjerice, ubacivanje kratkih kôdova koji se često ponavljaju između instrukcija programa, vrlo često će izazvati pogrešku kod WinDasm-a. IDA je, međutim, puno naprednija, pa će ju biti teže zbuniti. No postoje makroi kojima se i to može postići.

Najjednostavniji primjer bilo bi recimo korištenje malo drugačije, neintuitivne sintakse za naredbu skoka. Primjerice, niz naredbi

```
Push offset label
Ret
```

imat će isti učinak kao i naredba

```
jump label
```

Često korištenje ovako jednostavnog trika može zaista zbuniti disasembler ili u krajnjem slučaju otežati čitljivost i praćenje programa, no međutim ovako jednostavna metoda teško da će ikoga iznenaditi. Ipak, mogućnosti kompliciranja ovog kratkog koda su mnoge. Nekoliko mogućnosti bilo bi:

- možemo koristiti *eax* registar kao međuspremnik za memorijski odmak prilikom stavljanja na stog i skidanja sa stoga

- odmak za skok možemo spremiti i u podatkovni dio programa kao konstantu, koju dohvaćamo preko labele, te dodatno koristimo *eax* registar kao međuspremnik kod slanja i skidanja sa stoga
- možemo skok tretirati kao poziv potprograma, pa umjesto `jump`, korisititi `call`

itd.

Ovo su samo neke od mogućnosti, ali ljudska mašta je neograničena. Bitno je samo, koji god se makro koristio, da funkcionalnost ostane ista.

4.3. Kako dobro zaštititi svoj program?

Česta pojava kod programske zaštite je da programeri, u nastojanju da što jače, ili bolje, zaštite svoj proizvod, pretjeraju, ili premalo testiraju (ponekad i općenito zanemare testiranje), pa onda dolazi do raznih problema u kompatibilnosti s platformama i sklopovljem i/ili operativnim sustavom na kojem legalni korisnici pokreću proizvod. Ponekad se dešavaju kontraefekti poput nemogućnosti unošenja ispravnog registracijskog ključa i sl.

Stoga je potrebno, uz pametnu programsku zaštitu, ujedno temeljito testirati program na raznim platformama kako bi se uvjerili da implementirana zaštita neće, uz ilegalnu, spriječiti i legalnu uporabu programa.

Evo nekoliko praktičnih savjeta za bolju programsku zaštitu i otežavanje (onemogućavanje) probijanja iste:

- Metoda u programskom kôdu koja služi za uspoređivanje ključeva ili potvrdu registracije ne treba imati samoopisujuće ime, poput npr. `provjeriKlju č()`. Prilikom probijanja zaštite prvo će se potražiti ovakva metoda, i ako se ta metoda može prepoznati po imenu, lako je unaprijed zaključiti na koji dio kôda se treba koncentrirati.
- Treba izbjegavati nepotrebne poruke o pogreškama; ukoliko program izbacuje grešku nakon pogrešno unesenog registracijskog ključa, moguće je preko pogreške pronaći dio kôda za provjeru registracije. Preporučljivo je po mogućnosti koristiti dinamičko generiranje poruka o pogreškama, ili kriptirati metodu za generiranje pogrešaka tako da ju je u dohvaćenom asemblerskom kôdu teže pronaći.
- Preporučljivo je koristiti što jaču enkripciju za registracijske datoteke i ključeve, po mogućnosti 1024-bitni (ili jači) RSA. Ovu vrstu enkripcije danas je nemoguće probiti u nekom razumnom vremenu.
- Korištenje nasumičnih provjera registracije vrlo je dobra metoda u borbi protiv probijanja zaštite. Primjerice, ako netko i uspije ukloniti prvobitni zahtjev za ključem, nasumične zahtjeve tijekom rada programa teško je ili nemoguće predvidjeti, pa bi to predstavljalo puno veći posao prilikom probijanja zaštite.
- Dobar je trik „prisiliti“ program da se ponovno pokrene nakon unosa registracijskog ključa prije testiranja ispravnosti ključa. Tako će biti puno teže pratiti aplikaciju pomoću programa za praćenje izvođenja.
- Korištenje i provjera funkcije sažetka (*checksum*) vrlo je korisna, jer njom se može saznati da li je došlo do nekakvih izmjena u originalnom izvršnom kôdu. Većina pokušaja probijanja zahtijeva ovakve promjene (heks editor!), pa će ovo dodatno otežati posao bilo kome tko pokuša probiti zaštitu.
- Provjera registracijskog ključa može se podijeliti na više metoda, tako da ukoliko se prilikom probijanja zaštite propusti jedna od metoda, probijanje neće uspjeti.

- Korištenje lažnih procedura za provjeru registracijskog ključa zbunjuje u pokušajima dohvaćanja asemblerskog kôda.
- Ispravne podatke o registraciji treba držati na pametno odabranom mjestu, primjerice u nekoj .DLL datoteci unutar sistemskog direktorija Windowsa, umjesto u nekoj .TXT datoteci ili u *registryju*. Ovo je također zbunjujuće u pokušajima probijanja zaštite, a dodatno otežanje probijanja postiže se ako se taj registracijski ključ raspodijeli ili kopira na više datoteka.
- Enkripcija registracijskog ključa nekakvim jedinstvenim sklopovskim serijskim brojem (npr. serijski broj tvrdog diska) onemogućit će korištenje programa s istim registracijskim podacima na drugom računalu.
- Nije loše koristiti dugačke registracijske ključeve. Dugački registracijski ključevi teži su za dohvatiti, a mogu i sadržavati važne podatke za pokretanje programa, ili čak dijelove koda, tako da uslijed bilo kakve promjene izvršnog kôda program ne bi radio.
- Kod programa s vremenskim ograničenjem lako je prevariti program jednostavnom promjenom sistemskog vremena. U tom slučaju dobro je, uz čitanje sistemskog vremena, čitati i zadnje vrijeme modifikacije više sistemskih datoteka. Ako naiđemo na neki datum koji je veći od sistemskog vremena, znači da se sistemsko vrijeme mijenjalo.
- Što je vremenski dulja rutina za testiranje registracijskog ključa, puno će dulje trajati dohvaćanje asemblerskog kôda ili praćenje izvođenja.
- Ako neki program u svojoj verziji s periodom isprobavanja ima limitirane neke funkcije, te limitirane funkcije ne smiju biti uključene u kôd. Dobra je ideja da se ove funkcionalnosti integriraju kao dio registracijskih podataka (npr. kroz registracijsku datoteku).
- Često izbacivanje novih verzija programa i česte pojave programskih izmjena i poboljšanja (eng. *version update*, *version upgrade*) mogu navesti i najupornije da odustanu od probijanja zaštite.
- Ispravan registracijski ključ nikada se ne smije pojaviti u memoriji, jer bi se u tom slučaju mogao dohvatiti praćenjem izvođenja. Umjesto toga, može se kriptirati ispravan i unešeni registracijski ključ, pa ih takve uspoređivati, ili se mogu uspoređivati sažeci ključeva (sažetak = eng. *hash*).
- U današnje vrijeme, jedan od najboljih načina zaštite je inzistiranje na registraciji putem Interneta.
- Preporučljivo je koristiti trikove prevencije dohvaćanja asemblerskog kôda i praćenja izvođenja. Ako je nemoguće doći do asemblerskog kôda i/ili pratiti izvođenje programa, nemoguće mu je i probiti zaštitu.

5. ZAKONSKE DIREKTIVE

Teško je danas definirati legalnost probijanja zaštite. Svaka zemlja ima svoj zakon o probijanju zaštite i ilegalnim (piratskim) verzijama programa, neke ga zemlje niti nemaju, no ovi se zakoni više odnose na povrede zakona o kopiranju (eng. *copyright*), te na distribuciju ilegalnih programa, uz ili bez profita, te također na distribuciju probijenih i izmijenjenih izvršnih datoteka, ilegalnih serijskih brojeva, generatora ključeva i sl., prvenstveno na njihovu cjelokupnu uporabu. Opće mišljenje je da je legalno probiti programsku zaštitu (odnosno izmijeniti izvršnu datoteku tako da program može raditi bez zaštite), pa čak i slati tu izmijenjenu izvršnu datoteku na Internet ili skidati s Interneta, sve dok se ona ne koristi za probijanje programske zaštite, odnosno u konačnici za ilegalno korištenje programa. Drugim riječima, dopušteno je probiti zaštitu legalno kupljenog programa za vlastito korištenje, ali je zabranjeno dalje probijeni program distribuirati, koristiti na drugim računalima i slično.

Ako razmotrimo slučaj pobliže, shvatit ćemo da probijanje zaštite nije ništa osim unazadnog inženjerstva (eng. *reverse engineering*). Programsko unazadno inženjerstvo odnosi se upravo na dohvaćanje asemblerskog kôda iz izvršnih datoteka, dohvaćanje izvornog kôda u višim programskim jezicima i slične radnje, koje osobe sa znanjem probijanja zaštite redovno koriste kako bi shvatili pozadinu zaštite programa koji probijaju. Unazadno inženjerstvo je bilo koja radnja poduzeta u svrhu dobivanja izvornog kôda programa, u jeziku u kojem je pisan, iz izvršnog kôda, radi boljeg razumijevanja rada programa, bilo kakvog istraživanja ili jednostavno gubitka izvornog kôda nekog vlastitog programa. Pitanje je samo kome je dozvoljeno unazadno inženjerstvo, a kome ne.

U pravne dileme neću puno ulaziti. Vrijedi samo još spomenuti dvije pravne direktive koje su definirale zakone o pravima digitalnog kopiranja u Americi i Europi, pa se tako dotiču i unazadnog inženjerstva, odnosno problema probijanja programske zaštite (iako ne preizravno). Obje direktive su često oštro kritizirane radi svoje nejasnosti i nepotpunosti. To su „*The Digital Millenium Copyright Act*“ iz 1996. (za SAD) i „*EU Copyright Directive*“ iz 2001. (za Europsku Uniju). Obje ove direktive slažu se oko toga da je distribucija bilo kakvih sredstava, pomagala ili vodiča za zaobilaženje programske zaštite ilegalna, a recimo po DMCA direktivi osoba smije zaobići zaštitu za svoje vlastite potrebe (ovo je definirano i kroz određene iznimke), ali to mora postići sama, jer je, kako sam već spomenuo, distribucija bilo kakvih alata i informacija o zaobilaženju zaštite zabranjena. Po europskoj direktivi ovako nešto je također zabranjeno.

Informacije o ovim odredbama mogu se pronaći na web adresama navedenima u literaturi.

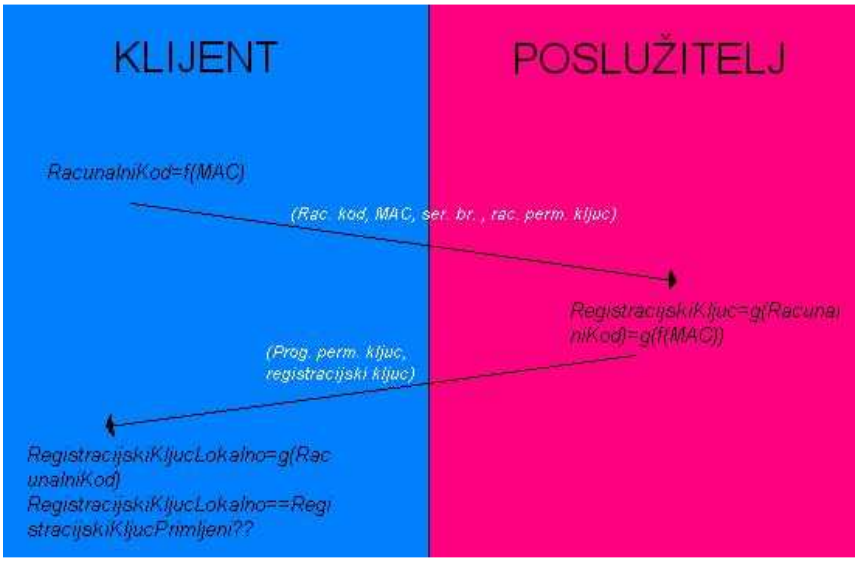
6. OSTVARENJE KOMBINIRANE PROGRAMSKE ZAŠTITE

Zadatak u sklopu ovog diplomskog rada bio je isprojektirati i isprogramirati kombiniranu programsku zaštitu temeljenu na klijent-poslužitelj arhitekturi. Na temelju proučenog materijala iz literature i brojnih postojećih funkcionalnih rješenja programske zaštite osmislio sam jednostavan model programske zaštite pomoću registracijskog ključa koji bi mogao biti korišten u svrhe sprečavanja ilegalnog i neovlaštenog korištenja programskih EXE i DLL datoteka, te za praćenje instalacija nekog programskog proizvoda na kojem se zaštita koristi.

Osmišljeni model zaštite odgovara kombinaciji tipova 3 i 4 zaštita koje sam opisao u odlomku 2.2.1. ovog diplomskog rada. On, dakle, kombinira provjeru registracijskog ključa putem Interneta s povezivanjem registracijskog ključa s internim podacima računala, odnosno serijskim brojevima sklopovlja. Ovakav model nije nikakva novost u ovom području računarske znanosti, dapače, ovakav tip ostvarenja programske zaštite vrlo je popularan u današnje vrijeme, jer 'veže' autentični programski proizvod prilikom prve instalacije (odnosno aktivacije proizvoda) uz računalo na kojem se instalira, čime se ograničava uporaba proizvoda na samo jedno računalo, što bitno smanjuje mogućnost probijanja programske zaštite, jer čak i da napadač sazna ključ, proizvod s ukradenim ključem ne bi radio na drugom računalu osim na onom na kojem je aktiviran.

Osmišljeni model programske zaštite koristit će MAC adresu računala domaćina kao argument u funkciji za izračunavanje registracijskog ključa. Ovime će se osigurati veza između registracijskog ključa programskog proizvoda i računala na kojem se proizvod instalira. Princip je vrlo jednostavan: klijent generira računalni kôd po određenoj funkciji u ovisnosti o MAC adresi računala klijenta, te ga šalje poslužitelju; poslužitelj na osnovu primljenog računalnog kôda izračunava, opet po posebnoj funkciji, registracijski ključ, kriptira ga i šalje klijentu. Računalni kôd vidljiv je svima (i ne kriptira se), a registracijski ključ zapisuje se samo u poslužiteljskoj bazi podataka. Ove funkcije mogu se prikazati na sljedeći način:

$$RacunalniKod = f(MAC)$$
$$RegistracijskiKljuc = g(RacunalniKod) = g(f(MAC))$$



Slika 16. Model računanja i slanja ključeva

Bilo bi poželjno, a i prijeko potrebno (radi adekvatnog nivoa sigurnosti) da ove funkcije budu dovoljno komplicirane da budu gotovo neprobojne. Radi primjera, u ovom projektu korištene su funkcije koje rade jednostavnu slučajnu permutaciju i na kraju računaju sažetak varijable predane u argumentu. Naravno, za realnu uporabu modela zaštite ovo nije ni izdaleka dovoljno snažan algoritam.

Uz izračunati računalni kôd, klijent šalje poslužitelju i nekakav specifičan kôd, koji je jedinstven za svaku pojedinu (legalnu) kopiju programa koji se instalira. Dakle to bi mogao biti nekakav serijski broj CD-a (ja sam ga u projektu nazvao upravo tako), ukoliko je instalacija programa smještena na optičkom mediju, ili bi to jednostavno mogao biti nekakav jedinstveni ID proizvoda koji se dobiva na prodajnom mjestu (gdje je kupljen programski paket), ili se šalje preko elektronske pošte i sl. Pošto ovaj serijski broj mora ostati tajan, poslužitelju se šalje samo njegov dio, uz sažetak cijelog (sažetak = eng. *hash*). Ovaj dio serijskog broja koji se šalje također će biti sam po sebi jedinstven, pa će se i prema njemu samome moći identificirati i pronaći korisnik, odnosno ID kopije proizvoda koja se instalira (moći će se jednoznačno pronaći u bazi podataka). Sažetak će, pak, poslužiti kao osiguranje da korisnik koji se pokušava registrirati posjeduje uistinu cijeli serijski broj.

Prilikom instalacije programa (ili prilikom prvog pokretanja) korisnika će se pitati da unese serijski broj kopije koju trenutno instalira, a ukoliko mu ovo nije prva instalacija (recimo ponovna instalacija nakon formatiranja tvrdog diska) tražit će ga se upis računalnog kôda. Registracijski broj, kao što sam ranije napomenuo, nalazit će se samo u bazi podataka na poslužitelju, dakle niti sam korisnik ga neće znati.

Tako registracija programa neće ovisiti samo o jednom, već o dva 'ključa', naime računalnom kôdu i serijskom broju kopije, a time će biti otežana eventualna krađa registracijskog broja. Naravno, određenu tajnu informaciju će morati posjedovati i sam korisnik, jer ga je bez toga nemoguće autenticirati. Pošto su računalni kôd i MAC adresa nezaštićeni i može ih eventualno bilo tko pročitati i/ili lažirati, potrebno je držati serijski broj u tajnosti.

Upis serijskog broja i računalnog kôda bit će jedini koraci koje korisnik treba napraviti u procesu registracije, sve ostalo obavlja se automatski. U pozadini se odvija cijeli niz radnji, među kojima je i slanje podataka poslužitelju, izračun spomenutih funkcija za dobivanje računalnog kôda i registracijskog ključa, kriptiranje, te slanje kriptiranih podataka natrag klijentu. Klijent i poslužitelj razmjenjivat će ove podatke preko znakovnih nizova posebnog formata. Sve ove radnje opisane su korak po korak u opisu načina rada algoritma, te u pseudokodu u daljnjem tekstu.

Važno je još jednom napomenuti da je ovaj projekt nastao samo kao primjer implementacije jedne programske zaštite, i ima vrlo vjerojatno velik broj nedostataka, koji bi se mogli ispraviti dodatnim poboljšanjima kôda. Ove nedostatke i načine ispravljanja tih nedostataka analizirat ću također u kasnijem tekstu.

6.1. Opis rada algoritma

Prije samog pseudokôda, dodatno ću pojasniti veličine s kojima će klijent i poslužitelj baratati:

MAC adresa – fizička adresa mrežne kartice, jedinstveni identifikator koji će se koristiti kao osnova za izradu računalnog kôda;
Serijski broj – jedinstveni identifikator kopije programa koja se instalira i registrira;
Računalni permutacijski ključ – slučajno ispremiješani niz od prvih 6 prirodnih brojeva (MAC adresa je duljine 6 okteta) prema kojem ćemo unutar funkcije za izračunavanje računalnog kôda ispremiješati redoslijed okteta u MAC adresi;

Računalni kôd – niz od 20 okteta koji će se koristiti u funkciji izračunavanja registracijskog ključa; duljina mu je 20 okteta jer će se računalni kôd dobivati kao 20-oktetni SHA-1 sažetak permutirane MAC adrese
Programski permutacijski ključ – slučajno ispremiješani niz prvih 20 prirodnih brojeva, koji će se koristiti kao ključ za permutiranje okteta računalnog kôda
Registracijski ključ – konačni tajni ključ kojim će se registrirati program (na prvoj i budućim instalacijama); taj je ključ poznat samo poslužiteljskom servisu (čuva se u bazi podataka na poslužitelju)
***fromClient* znakovni niz** – znakovni niz posebnog formata koji će klijent slati poslužitelju, a u kojem će se nalaziti MAC adresa, dio serijskog broja, sažetak serijskog broja, javni ključ za enkripciju, te računalni kôd
***toClient* znakovni niz** – znakovni niz posebnog formata koji će poslužitelj slati klijentu, i koji će biti kriptiran, a u kojem će se nalaziti registracijski ključ i programski permutacijski ključ
Javni ključ – RSA javni ključ koji će klijent slati poslužitelju, a s kojim će poslužitelj kriptirati simetrični ključ za kriptiranje *toClient* znakovnog niza
Simetrični ključ – AES ključ za kriptiranje *toClient* znakovnog niza, koji poslužitelj kriptira javnim ključem primljenim od klijenta

6.1.1. Pseudokôd

```
Klijentski_program
{
      Dohvati MAC;
      Generiraj racunalni permutacijski kôd;
      Izracunaj racunalni kôd;
      Zatrazi korisnika upis serijskog broja;
      AKO (proizvod_aktiviran)
            Trazi korisnika upis racunalnog kôda;
      INACE
            Prikazi korisniku racunalni kôd;
      Kreiraj fromClient znakovni niz;
      kriptiraniSimetricniKljuc = WebMetoda(fromClient, ref
      kriptiraniToClient);
      simetricniKljuc = Dekriptiraj(kriptiraniSimetricniKljuc,
      privatniRSAkljuc);
      toClient = Dekriptiraj(kriptiraniToClient, simetricniKljuc);
      Dohvati registracijski permutacijski kljuc iz toClient znakovnog
      niza;
      Izracunaj lokalno registracijski kljuc;
      Dohvati registracijski kljuc iz toClient znakovnog niza;
      Usporedi lokalni registracijski kljuc s primljenim;
      AKO (registracijski_kljucevi_jednaki)
            Registracija uspjesna;
      INACE
            Registracija neuspjesna;
}

String Posluziteljski_program(fromClient, ref kriptiraniToClient)
{
      Dohvati serijski broj iz fromClient znakovnog niza;
      Aktiviran = ProvjeriAktivaciju(serijskiBroj);
      AKO (aktiviran)
            Dohvati reg. kljuc, progr. perm. kljuc, racunalni kôd i
            racunalni perm. kljuc iz baze;
            Dohvati MAC adresu iz fromClient znakovnog niza;
            Izracunaj racunalni kôd pomocu MAC adrese i racunalnog perm.
            kljuca;
            Usporedi izracunati rac. kôd s onim iz baze;
```

```
            AKO (racunalni_kôdovi_se_ne_poklapaju)
                  Generiraj pogrešku;
      INACE
            Dohvati racunalni kôd i rac. perm. kljuc iz fromClient
            znakovnog niza;
            Generiraj progr. perm. kljuc i izracunaj registracijski kljuc;
            Spremi racunalni kôd, rac. perm. kljuc, registracijski kljuc i
            progr. perm. kljuc u bazu;
      Kreiraj toClient znakovni niz;
      kriptiraniToClient = Kriptiraj(toClient, simetricniKljuc);
      Dohvati javni kljuc iz fromClient znakovnog niza;
      kriptiraniSimetricniKljuc = Kriptiraj(simetricniKljuc, javniKljuc);
      Vrati kriptiraniSimetricniKljuc;
}
```

6.1.2. *SoftProtSystem* okruženje

Iako ne bih ulazio previše u sam kôd projekta (u programskom jeziku C#), smatram da je u određenoj mjeri potrebno, za razumijevanje samog načina rada algoritma, opisati barem ukratko metode i veličine koje se koriste u okruženju ovog modela programske zaštite, kojeg sam prigodno nazvao `SoftProtSystem` (eng. ***Soft****ware* ***Prot****ection* ***System*** = sustav za programsku zaštitu).

Inače, `SoftProtSystem` okruženje sama je jezgra ostvarene programske zaštite u ovom projektu, a može se kao takvo koristiti i kao temelj za izgradnju različitih varijacija programske zaštite (bazirane na klijent-poslužitelj modelu), tako da klijentska i poslužiteljska aplikacija koriste reference na DLL datoteke izgrađene iz klasa u ovom okruženju. `SoftProtSystem` se sastoji od dvije klase, `SoftProtClient` i `SoftProtServer`. Kako im samo ime kaže, jednu će klasu u radu koristiti klijent, a drugu poslužitelj u zamišljenom modelu programske zaštite. Ove klase sadržavat će metode i varijable potrebne za funkcioniranje zaštite i siguran rad i komunikaciju klijenta i poslužitelja.

Same klase sa svojim prototipima izgledaju ovako:

```
public class SoftProtClient
{
      private int[] _racPermKljuc;
      private int[] _progPermKljuc;
      private byte[] _racunalniKod;
      private RSACryptoServiceProvider rsa;
      private bool aktiviran;

      #region Postavljanje i dohvat racunalnog koda

      public byte[] RacunalniKod;
      public void PostaviRacunalniKod(string racunalniKod);
      public string DohvatiRacunalniKodKaoString();

      #endregion

      #region WINAPI DllImport

      private extern static void GetSystemTime(ref SYSTEMTIME
      lpSystemTime);
      static extern int UuidCreateSequential(out Guid guid);
      private static Guid UuidCreateSequential();

      #endregion

      public SoftProtClient();
```

```
      #region Pomocne funkcije za pretvorbu

      private string ByteArray2HexString(byte[] array);
      private byte[] HexString2ByteArray(string key);
      private string IntArray2String(int[] array);
      private int[] String2IntArray(string permutation);

      #endregion

      #region Generiranje kljuceva

      public byte[] DohvatiMACAdresu();
      private int GenerirajSjeme();
      private int[] GenerirajRacunalniPermKljuc(int seed);
      public void IzracunajRacunalniKod();
      public string IzracunajRegistracijskiKljuc();

      #endregion

      #region Dekripcija

      public void DekriptirajSimetricniKljuc(byte[]
      kriptiraniSimetricniKljuc, byte[] kriptiraniIV, out byte[]
      dekriptiraniSimetricniKljuc, out byte[] dekriptiraniIV);
      public void Dekriptiraj_ToClient_String(byte[] kriptiraniTekst, out
      string cistiTekst, AesCryptoServiceProvider aes);

      #endregion

      #region Parsiranje to_client stringa

      public string DohvatiProgKljucIz_ToClient_Stringa(string toClient);
      private void DohvatiProgPermKljucIz_ToClient_Stringa(string
      toClient);

      #endregion

      public void Kreiraj_FromClient_String(string serial, out string
      fromClient);
}

public class SoftProtServer
{
      private Aes aes;

      #region WINAPI DllImport

      private extern static void GetSystemTime(ref SYSTEMTIME
      lpSystemTime);
      private struct SYSTEMTIME
      {
            public ushort wYear;
            public ushort wMonth;
            public ushort wDayOfWeek;
            public ushort wDay;
            public ushort wHour;
            public ushort wMinute;
            public ushort wSecond;
            public ushort wMilliseconds;
      }
```

```
        #endregion

        public SoftProtServer();

        #region Pomocne funkcije za pretvorbu

        public string ByteArray2HexString(byte[] array);
        public byte[] HexString2ByteArray(string key);
        public string IntArray2String(int[] array);
        public int[] String2IntArray(string permutation);

        #endregion

        #region Generiranje kljuceva

        private int[] IzracunajProgPermutacijskiKljuc(int seed);
        public string IzracunajRegistracijskiKljuc(string RacunalniKod, out
        string progPermKljuc)
        public string IzracunajRacunalniKod(string MAC, string permutacije);

        #endregion

        #region Enkripcija

        public void Kriptiraj_ToClient_String(string cistiTekst, out byte[]
        kriptiraniTekst);
        public void KriptirajSimetricniKljuc(out byte[]
        kriptiraniSimetricniKljuc, out byte[] kriptiraniIV,
        RSACryptoServiceProvider rsa);

        #endregion

        #region Parsiranje from_client stringa

        public string DohvatiRacunalniKodIz_FromClient_Stringa(string
        fromClient);
        public string DohvatiSerialIz_FromClient_Stringa(string fromClient);
        public string DohvatiSerialHashIz_FromClient_Stringa(string
        fromClient);
        public string DohvatiJavniKljucIz_FromClient_Stringa(string
        fromClient);
        public string DohvatiRacPermKljucIz_FromClient_Stringa(string
        fromClient);
        public string DohvatiMACIz_FromClient_Stringa(string fromClient);

        #endregion

        public void Kreiraj_ToClient_String(string softverskiKljuc, string
        softPermKey, out string toClient)
}
```

Iz naziva metoda poprilično je intuitivno izvući zaključak o njihovoj namjeni. Pošto obje klase sadrže relativno velik broj metoda, one se mogu grupirati po funkcionalnosti kako bi ih bilo jednostavnije objasniti. U opisu klasa skupine su naznačene pretprocesorskim naredbama `#region` i `#endregion` u C# programskom jeziku.

U klasi `SoftProtClient` prva skupina su metode za rad s računalnim kôdom. Računalni kôd čuva se kao javno dostupna varijabla u instanciranom objektu tipa `SoftProtClient`. Metoda `PostaviRacunalniKod` rabi se za postavljanje varijable

`_racunalniKod` koja se dohvaća preko pristupne metode varijable `RacunalniKod` kao polje okteta, dok se u obliku znakovnog niza dohvaća metodom `DohvatiRacunalniKodKaoString`.

Druga skupina metoda su pozivi dvije funkcije iz eksternih sistemskih DLL datoteka Windows API-ja: `GetSystemTime` i `UuidCreateSequential`. Metodi `GetSystemTime` predaje se posebna struktura `SYSTEMTIME` u koju se pri vraćanju iz sistemskog poziva preko reference upisuje precizno sistemsko vrijeme (godina, mjesec, dan u tjednu, dan, sat, minute, sekunde i milisekunde). Ova funkcija poslužit će nam za dobivanje sjemena za instance generatora pseudoslučajnih brojeva za dobivanje računalnog permutacijskog ključa. `UuidCreateSequential` metoda, pak, služi za kreiranje UUID (***U**nique **U**niversal **Id**entifier*, eng. *jedinstveni univerzalni identifikator*) strukture, koja je garantirano univerzalno jedinstven niz okteta, te ju se koristi u mnogim područjima gdje je potrebna autentikacija, pa tako i u sigurnosti odnosno programskoj zaštiti. Mi ju ovdje, međutim, nećemo koristiti u sigurnosne svrhe, već za dohvaćanje MAC adrese mrežne kartice računala, jer UUID struktura, među svoja 4 podatkovna elementa od sveukupno 16 okteta, sadrži i MAC adresu, i to u posljednjih 6 okteta. UUID struktura definirana je ovako:

```
typedef struct _GUID
{
    unsigned long Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char Data4[8];
} GUID, UUID;
```

U sistemskom smislu, UUID se koristi u RPC bibliotekama za provjeru kompatibilnosti između klijenata i poslužitelja, te za odabir među različitim implementacijama sučelja. `GetSystemTime` se nalazi u sistemskoj datoteci *kernel32.dll*, a `UuidCreateSequential` u sistemskoj datoteci *rpcrt4.dll*. (Opaska: MAC adresa se jednostavnije može dohvatiti WQL upitima preko WMI objektnog upravitelja na sistemski WMI repozitorij (baza objekata za dohvat svih informacije o sklopovskom i programskom sastavu računala), no WMI objekti se na različitim operativnim sustavima nalaze na različitim mjestima i dohvaćaju različitim upitima, pa je bolje u ovu svrhu koristiti spomenuti WINAPI.)

Sljedeća skupina metoda sadržanih u `SoftProtClient` klasi su pomoćne metode za pretvaranje između različitih formata varijabli koje koristimo. Tako `ByteArray2HexString` služi za pretvaranje polja okteta u heksadecimalni niz znakova (uglavnom za spremanje ključeva u znakovne nizove za slanje između klijenta i servera, te za prikaz korisniku). `HexString2ByteArray` radi suprotno, dakle pretvara heksadecimalni niz znakova u polje okteta (za interno korištenje unutar klasa; sve računanje obavlja se s oktetima). Metoda `IntArray2String` služi za pretvaranje permutacijskih polja u tekstualni zapis (znakovni nizovi kao nizovi brojeva s razmacima; također služi za stvaranje znakovnog zapisa za slanje između klijenta i poslužitelja), dok `String2IntArray` radi obratno, dakle pretvara znakovni niz u polje prirodnih brojeva. Ove metode sadržane su također i u klasi `SoftProtServer`.

Dalje imamo skupinu metoda koje zajedno i međusobno služe za generiranje ključeva. To su, konkretno, metode: `DohvatiMACAdresu`, `GenerirajSjeme`, `GenerirajRacunalniPermKljuc`, `IzracunajRacunalniKod` i `IzracunajRegistracijskiKljuc`. Metoda `DohvatiMACAdresu` interno poziva WINAPI funkciju `UuidCreateSequential`, te iz povratne strukture dohvaća i vraća MAC adresu. `GenerirajSjeme` koristi WINAPI funkciju `GetSystemTime`, te zbrojem svih vrijednosti iz strukture `SYSTEMTIME` računa i vraća pseudoslučajno sjeme za generiranje pseudoslučajnih permutacija redoslijeda okteta u MAC adresi. Ovako generirano sjeme predaje se kao argument metodi `GenerirajRacunalniPermKljuc`, koja će kreirati novu instancu ugrađene .Net klase `Random` koristeći to isto

sjeme, te će 'ispremiješati' redoslijed prvih 6 prirodnih brojeva i vratiti permutirani niz. Ovako dobivenu permutaciju koristit će metoda `IzracunajRacunalniKod`. Ona će prema toj permutaciji 'ispremiješati' redoslijed okteta u MAC adresi (dobivenoj iz poziva metode `DohvatiMACAdresu`), te dodatno izračunati 20-oktetni SHA-1 sažetak koji će se dalje koristiti kao računalni kôd. Konačno, metoda za generiranje registracijskog ključa `IzracunajRegistracijskiKljuc`, koristit će u tu svrhu računalni kod dobiven iz poziva metode `IzracunajRacunalniKod`, te programski permutacijski kljuc, koji joj se predaje u argumentu, a koji je dobiven s poslužitelja u sklopu *toClient* znakovnog niza.
Kao posebnu skupinu funkcija možemo gledati i dvije metode za dekripciju podataka dobivenih s poslužitelja. Prva od njih, `DekriptirajSimetricniKljuc`, koristit će privatni ključ iz RSA objekta kreiranog pri stvaranju instance klase `SoftProtClient` za dekriptiranje simetričnog ključa, koji se unutar objekta `AesCryptoServiceProvider` predaje kao argument metodi `Dekriptiraj_ToClient_String`, za dekriptiranje znakovnog niza primljenog od poslužitelja, koji sadrži registracijski ključ i programski permutacijski ključ.

Konačno, posljednja skupina metoda u `SoftProtClient` klasi odnosi se na metode za parsiranje *toClient* znakovnog niza, odnosno za dohvaćanje registracijskog ključa i programskog permutacijskog ključa iz primljenog znakovnog niza nakon poziva web-metode. To su metode `DohvatiProgKljucIz_ToClient_Stringa` i `DohvatiProgPermKljuc-Iz_ToClient_Stringa` . One rade jednostavnim principom čitanja redak po redak iz primljenog znakovnog niza, te dijeljenja redaka teksta na podnizove i usporedbe s ključnim riječima. Metoda `DohvatiProgPermKljucIz_ToClient_Stringa` tipa je `void` jer smo rekli da programski permutacijski ključ i registracijski ključ moraju ostati tajni, pa ova metoda služi samo za upis u privatnu varijablu `_progPermKljuc`. Ovo je, doduše, malo nespretno izvedeno, ali neophodno, jer moramo imati programski permutacijski ključ kako bismo izračunali registracijski ključ potreban za lokalnu usporedbu i registraciju.

Zasebno smo ostavili još i metodu za stvaranje znakovnog niza za slanje poslužitelju, imena `Kreiraj_FromClient_String`, koja jednostavno zapisuje redak po redak u teksta u znakovni niz s odgovarajućim ključnim riječima i podacima koji se šalju, te dodatno prema varijabli `aktiviran` odlučuje o kojem tipu *fromClient* znakovnog niza (od dva ranije navedena) se radi.

Metode klase `SoftProtServer` također bismo mogli grupirati u skupine koje su gotovo analogne onima iz klase `SoftProtClient`. Tako je prva skupina ona s metodama koje pozivaju funkcije iz WINAPI-ja, konkretno u ovoj klasi koristimo samo `GetSystemTime`, dok `UuidCreateSequential` u ovoj klasi nije potreban (ne trebamo dohvaćati MAC adresu poslužitelja). `GetSystemTime` funkcija objašnjena je ranije u sklopu klijentske klase. Druga skupina su pomoćne funkcije za pretvorbu polja u znakovne nizove i obratno, što je također već objašnjeno.

Treća skupina je skupina metoda za generiranje ključeva. Ovakvu skupinu metoda sadrži i klasa `SoftProtClient`, razlika je u tome što ovdje umjesto metode za generiranje računalnog permutacijskog ključa imamo metodu `IzracunajProgPermutacijskiKljuc` za izračunavanje ključa za permutiranje redoslijeda okteta računalnog kôda kao jedan od međukoraka u generiranju konačnog registracijskog ključa. Ostale dvije metode (`IzracunajRegistracijskiKljuc` i `IzracunajRacunalniKod`) iz ove skupine identične su onima iz klase `SoftProtClient`, samo što se ovdje varijable koje se koriste pri generiranju ključeva predaju kao argumenti jer se ne pamte u instanci klase, već se računaju ili dohvaćaju iz klijentskog znakovnog niza.

Za sljedeću skupinu također postoji analogna skupina metoda na klijentskoj strani, samo što se ovdje, umjesto o dekripciji, radi o enkripciji simetričnog ključa i poslužiteljskog znakovnog niza.

Zadnje ostaju metode za parsiranje/dohvat iz *fromClient* znakovnog niza, koje rade na isti način kao već opisane slične metode iz klase `SoftProtClient` (ondje smo već opisali metode za parsiranje *toClient* znakovnog niza).

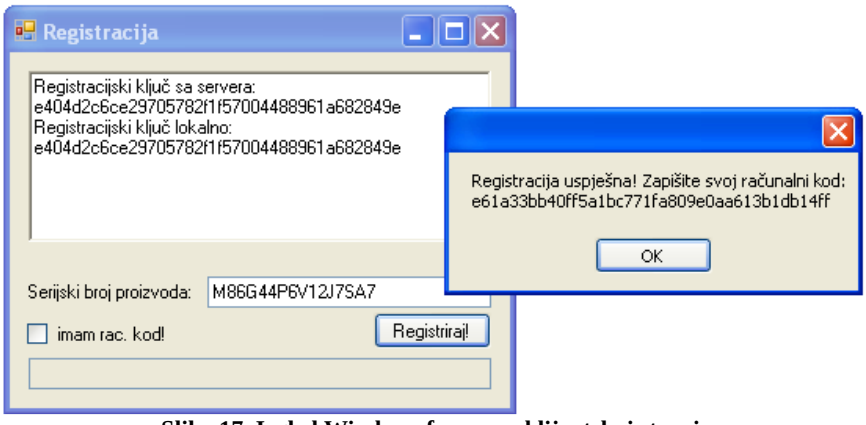
Na kraju ostaje, opet slično kao kod klijentske klase, još i metoda za kreiranje odgovarajućeg znakovnog niza `Kreiraj_ToClient_String`, ovaj put namijenjenog za slanje klijentu.

6.1.3. Klijentski i poslužiteljski program

Metode u klasama `SoftProtClient` i `SoftProtServer` općeniti su „*framework*“ za ostvarivanje ranije opisanog modela programske zaštite, a osim takvog modela, moguće je spomenute klase rabiti i za varijacije ovog modela, ovisno o ideji i kreativnosti programera. Dok su klase i metode u klasama fiksne i koriste se preko referenci na DLL datoteke, ostavljeno je na izbor programeru smisliti algoritam zaštite, odnosno klijentsku i poslužiteljsku aplikaciju, te na koji će način klijent i poslužitelj koristiti metode ponuđene u SoftProtSystem arhitekturi.

Poslužiteljska strana može biti rješena na doista razne načine. Ja sam za ovaj projekt odabrao web-servis kojem se kroz web-metodu predaje *fromClient* znakovni niz, a on kroz istu metodu vraća *toClient* znakovni niz klijentu. Alternativne bi metode bile: čisto slanje znakovnih nizova (samo uz uspostavljanje konekcije između klijenta i poslužitelja, ali bez korištenja web-servisa); ili recimo web-stranica u kojoj bi korisnik ručno upisivao potrebne podatke za *fromClient* znakovni niz, a ActiveX kontrola bi dalje radila posao na poslužitelju; slanje računalnog kôda može se obaviti čak i telefonski, uz male preinake (ne bi nam bila potrebna kriptografija, i koristila bi se drugačija baza podataka). Da ne ulazim preduboko, u ovom tekstu koncentrirat ću se samo na rješenje kakvo sam ja odabrao i realizirao, a to je komunikacija putem web-servisa.

Što se klijentskog programa tiče, predviđeno je da se izvodi kao međukorak pri instalaciji proizvoda, ili pri prvom pokretanju već instalirane aplikacije, gdje se punjenje aplikacije u memoriju blokira, pa se nudi registracija, ili demo korištenje, ili nekakvo korištenje uz ograničenje s vremenskim periodom i sl. Pošto je moj zadatak kroz ovaj diplomski rad bio samo ostvariti funkcionalnu programsku zaštitu, ja sam klijentsku stranu samo simulirao jednostavnom Windows formom, u koju korisnik upisuje potrebne podatke, te pritiskom na tipku pokreće 'proces registracije'. Ova forma prikazana je na donjoj slici. Korisnik u odgovarajuće kućice upisuje serijski broj, i, ukoliko se ne radi o prvoj instalaciji, računalni kôd, te pritišće tipku '*Registriraj!*'. Nakon što algoritam u pozadini obavi svoj posao (proračun i komunikaciju s poslužiteljem), u tekstualnom okviru iznad ispisuju se registracijski ključevi, te se pojavljuje poruka koja govori da li je program uspješno registriran (da li se ključevi poklapaju) ili ne.



Slika 17. Izgled Windows forme na klijentskoj strani

6.1.4. Detaljan opis izvođenja algoritma zaštite korak po korak

Opis koji slijedi specifičan je za način na koji je u projektu u sklopu ovog diplomskog zadatka ostvaren model programske zaštite, kao jedan demonstrativan primjer. On opisuje korak po korak izvođenje algoritma od trenutka kad korisnik pokrene aplikaciju za testiranje (dakle klijentski program), pa sve do trenutka kad se na ekranu ispiše informacija o uspješnoj ili neuspješnoj registraciji.

Klijentska aplikacija opisana je djelomično već u prethodnom tekstu. Dakle, pri pokretanju ove aplikacije korisnik pred sobom dobiva Windows formu s 2 manje kućice za upis teksta, jednom većom kućicom za ispis teksta, te oznakom '*Imam računalni kod!*' i gumbom '*Registriraj!*'. Ukoliko je riječ o prvoj instalaciji proizvoda (odnosno aktivaciji), dakle ukoliko uz serijski broj proizvoda koji želimo instalirati u bazi podataka ne piše još ništa osim '*False*' pod poljem '*aktivacija*', upisujemo u ovu formu u odgovarajuću kućicu samo serijski broj i pritišćemo gumb '*Registriraj!*'. Ako je, pak, riječ o nekoj ponovljenoj instalaciji proizvoda (dakle ako vrijedi: '*aktivacija*' = '*true*'), označit ćemo kvačicu na '*Imam računalni kod!*' i upisati računalni kôd koji smo trebali dobiti prilikom prve instalacije i sačuvati. To je više-manje sve što se traži od korisnika. Sve ostalo odvija se u pozadini, skriveno u odnosu na korisnika.

Algoritam na klijentskoj strani neće se konkretno granati u ovisnosti o aktivaciji, već će se ova informacija čuvati u varijabli `aktiviran` u instanci klase `SoftProtClient` koja se instancira na početku algoritma. Pri instanciranju objekta tipa `SoftProtClient` preko konstruktora klase `SoftProtClient()` pozivaju se interno metode `GenerirajRacunalniPermKljuc` i `IzracunajRacunalniKod` pri čemu se izračunati računalni permutacijski ključ i računalni kôd upisuju u odgovarajuće privatne varijable unutar objekta. Nakon poziva konstruktora klase i upisa informacije o aktivaciji, poziva se metoda `Kreiraj_FromClient_String`, kojoj će se kao argument predavati samo serijski broj, dok će sve ostale potrebne informacije pokupiti interno iz objekta `SoftProtClient`. Ukoliko programer želi, moguće je pomoću adekvatnih metoda izračunati 'ručno' računalni kôd, ukoliko ne želi koristiti onaj izračunat automatski pri pozivu konstruktora klase. Generiranje računalnog kôda koristit će se, naravno, samo u slučaju da se radi o aktivaciji programskog proizvoda, inače se u objekt `SoftProtClient` upisuje preko metode `PostaviRacunalniKod` računalni kôd iz odgovarajuće tekstualne kućice iz Windows forme.

Dva su različita formata *fromClient* znakovnog niza u ovisnosti da li se radi o aktivaciji proizvoda ili ne. Ako je riječ o aktivaciji, *fromClient* znakovni niz izgledat će ovako:

```
[from_client]
hardware_key;<računalni_kod>
serialA;<prva_polovica_serijskog_broja>
serial_hash;<SHA-1_sažetak_serijskog_broja>
public_key;<RSA_javni_ključ>
hard_permutation_key;<računalni_permutacijski_ključ>
```

Primjer ovakvog *fromClient* znakovnog niza:

```
[from_client]
hardware_key;ff7baa1caddb0b7f5b6ae7044b998f955ece34d1
serialA;XM4B6WA5
serial_hash;8a10982be8dcc5849378b0e777e7ea6f7255bc3d
public_key;<RSAKeyValue><Modulus>r6DmvnyGFxWVueNmi4syNspyNPvsG7sWfbC+we6rr2
0NeIK1kwNBwMAmilUGQXOmA2Tiq7g9rab4tCim1jTpBdKGwNBsQFck4tYmHMl/0/kDeqo2m7t2v
GtIVCYujFY4Gx9MTWPxL7s6neUSEtNYLK8yvj8CYiHIRwccHdSnbr8=</Modulus><Exponent>
AQAB</Exponent></RSAKeyValue>
```

```
hard_permutation_key;3 1 0 5 4 2
```

Ukoliko je riječ o ponovljenoj instalaciji (uvjet `aktiviran==true` je ispunjen), *fromClient* znakovni niz izgledat će ovako:

```
[from_client]
MAC;<MAC_adresa_mrežne_kartice>
serialA;<prva_polovica_serijskog_broja>
serial_hash;<SHA-1_sažetak_serijskog_broja>
public_key;<RSA_javni_ključ>
```

Primjer ovog drugog formata:

```
[from_client]
MAC;001a4b748a49
serialA;V4J21DYY
serial_hash;0bf6b1a943611514d0071737dbcc0f94240d4706
public_key;<RSAKeyValue><Modulus>zfQR0DDjclBo/KmpJ3vPScwb7FZ5v1lwnNIaQP1pCr
RG7BiieBfQuXtt8caWF1kRSOL1k8WOjg8XEcpnV+tJ7aRXBvVXqsEkpa6vCJIpQPSjCyIkkPiEQ
QY3Qf5KyEEJg/PRA2Ktsgwbev7+ln7YWFv99r6mz2psh0YFIx05Vxc=</Modulus><Exponent>
AQAB</Exponent></RSAKeyValue>
```

Metoda `Kreiraj_FromClient_String` interno odlučuje koji će format stringa koristiti u ovisnosti o privatnoj varijabli `aktiviran` unutar matičnog objekta. Vidljivo je da se serijski broj ne šalje u cijelosti preko mreže, već samo jedan njegov dio, dok je sažetak izračunat od cijelog serijskog broja. Ovo je potrebno radi sigurnosti, jer serijski broj treba ostati tajan. Javni ključ razmjenjuje se u XML formatu kakvog koristi .NET klasa `System.Security.Cryptography.RSACryptoServiceProvider`, koju i koristim u ovom projektu.

Jedna bitna razlika primjećuje se između dva formata znakovnog niza, a to je da se u drugom umjesto računalnog kôda i računalnog permutacijskog ključa šalje MAC adresa klijenta. Ovom je razlog činjenica što su računalni kôd i računalni permutacijski ključ upisani u bazu prilikom aktivacije proizvoda, pa ih je nepotrebno slati (uostalom klijent niti ne zna koji je permutacijski ključ korišten prilikom prve instalacije, jer se pri svakom pokretanju aplikacije on dobiva slučajno), a MAC adresa je potrebna kako bi poslužitelj mogao izračunati računalni kôd u kombinaciji s računalnim permutacijskim ključem (koji dohvaća iz baze podataka), te ga usporediti s onim računalnim kôdom koji ima u bazi podataka, kako bi se mogao uvjeriti da je računalo s kojeg je ovaj put poslan zahtjev za registracijom uistinu ono računalo s kojeg je proizvod i aktiviran.

Nakon kreiranja *fromClient* znakovnog niza, on se šalje poslužitelju. Ovo se obavlja pozivom web-metode (koju sam u kôdu nazvao `ObaviKomunikaciju`) preko reference na web-servis. Prototip web-metode `ObaviKomunikaciju` izgleda ovako:

```
public string[] ObaviKomunikaciju(string from_client, out string to_client)
```

Razmjena informacija ovdje će teći na slijedeći način: *fromClient* znakovni niz se pri pozivu web-metode predaje u prvi argument, a *toClient* znakovni niz vraćat će se po referenci kroz drugi argument. Preko `return` ključne riječi web-metoda vraćat će polje od dva znakovna niza, gdje će na nultom indeksu biti kriptirani simetrični ključ, a na indeksu 1 kriptirani IV (inicijalizacijski vektor).

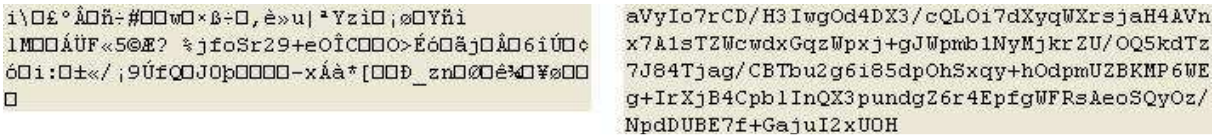
Prilikom poziva web-metode, daljnje izvođenje algoritma privremeno preuzima poslužitelj. On će prvo napraviti instancu klase `SoftProtServer` pomoću odgovarajućeg konstruktora, pri čemu će se interno kreirati simetrični ključ i IV u privatnoj varijabli .Net

objektu tipa `System.Security.Cryptography.Aes`. Zatim se metodom `DohvatiSerialIz_FromClient_Stringa` dohvaća serijski broj iz primljenog znakovnog niza, te se provjerava aktivacija (i općenito postojanje serijskog broja u bazi podataka) za taj serijski broj. Ovo se radi pozivom privatne web-metode `ProvjeriAktivaciju`. Web-metoda `ProvjeriAktivaciju` spaja se na bazu podataka i traži zapis sa zadanim serijskim brojem. Ako ne pronađe takav zapis, baca pogrešku, a ako pronađe, vraća vrijednost `false` ako serijski broj nije aktiviran, ili `true` ako jest. Međutim, `ProvjeriAktivaciju` utvrđuje (ne)poklapanje samo dijela serijskog broja, pa se dodatno unutar metode `ObaviKomunikaciju` dohvaća cijeli serijski broj iz baze, računa njegov SHA-1 sažetak, te uspoređuje sa sažetkom iz *fromClient* znakovnog niza. Ako se sažeci ne poklapaju, baca se pogreška.

Ako je proizvod već aktiviran, dohvaća se iz baze računalni kôd i registracijski ključ, te permutacijski ključevi. MAC adresa dohvaća se iz *fromClient* znakovnog niza, te se računa računalni kôd i uspoređuje s onim dohvaćenim iz baze podataka. Ako se računalni ključevi ne poklapaju, baca pogrešku.

U drugom slučaju, ako se uspostavi da proizvod nije aktiviran, dohvaća se iz *fromClient* znakovnog niza računalni kôd i računalni permutacijski ključ, računa se registracijski ključ pomoću metode `IzracunajRegistracijskiKljuc`, te se svi ključevi zajedno upisuju u bazu podataka, u n-torku koja odgovara poslanom serijskom broju. Metoda za izračunavanje registracijskog ključa poziva interno metodu `IzracunajProg-PermutacijskiKljuc`, a programski permutacijski ključ vraća preko reference kroz varijablu predanu kao drugi argument u pozivu.

Nakon što su obavljene provjere, računanja i upisi u bazu, konekcija s bazom se zatvara, te se poziva metoda `Kreiraj_ToClient_String`. Odmah nakon nje, poziva se i metoda `Kriptiraj_ToClient_String`, kojoj predajemo čisti tekst *toClient* znakovnog niza, i referencu na varijablu u koju će se upisati kriptirani *toClient* znakovni niz. Za kriptiranje znakovnog niza ova metoda interno koristi objekt tipa `Aes` iz imenovnog prostora `System.Security.Cryptography` instanciran pri pozivu konstruktora klase `SoftProtServer()`. Kriptirani *toClient* znakovni niz potrebno je pretvoriti u tekst u bazi 64, jer bi inače pri slanju kriptiranog teksta Microsoft XML Parser (program u sklopu .NET Framework-a koji je odgovoran za čitanje svega što se šalje pri komunikaciji s web-servisom) javljao pogrešku radi pojave neprintabilnih znakova. Ovaj znakovni niz upisuje se u varijablu iz drugog argumenta poziva metode `ObaviKomunikaciju` koja je predana preko reference.



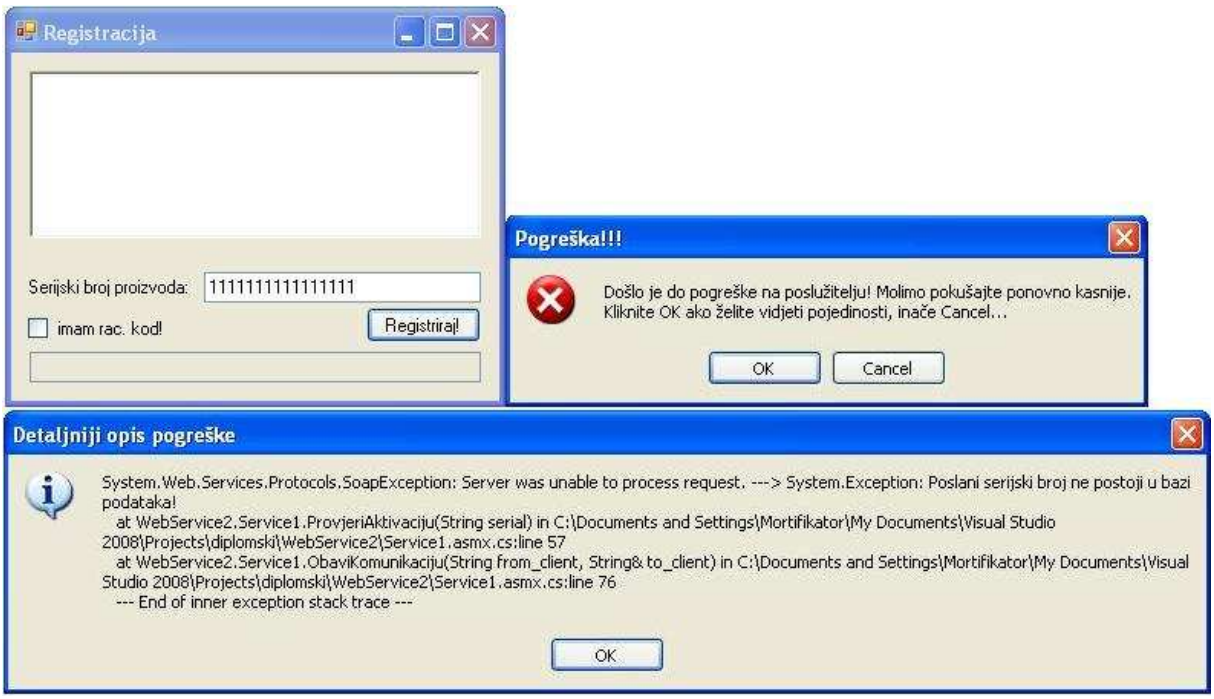
Slika 18. Kriptirani znakovni niz prije i poslije enkodiranja u bazu 64

Na kraju se kreira instanca .Net klase `RSACryptoServiceProvider`, u nju se dohvaća iz *fromClient* znakovnog niza zadnja nepreuzeta informacija, a to je javni ključ, te se poziva metoda `KriptirajSimetricniKljuc`, kojoj se u argumente predaju preko reference varijable u koje će se upisati kriptirani ključ i IV, te kao treći argument `RSACryptoServiceProvider` objekt s klijentskim javnim ključem. Podsjetimo se da je tajni ključ sadržan u privatnoj varijabli objektu tipa `System.Security.Cryptography.Aes` u instanci klase `SoftProtServer`. Kriptirani simetrični ključ i inicijalizacijski vektor također se transkodiraju u bazu 64, te se vraćaju klijentu u povratku iz poziva web-metode.

Pri povratku izvršavanja algoritma na klijent, klijentu je ostalo još samo dekriptirati, izračunati lokalno i usporediti registracijske ključeve. On će prvo dekodirati primljeni kriptirani *toClient* znakovni niz, te kriptirane simetrični ključ i IV iz baze 64, zatim dekriptirati simetrični ključ i IV pozivom metode `DekriptirajSimetricniKljuc`, te nakon

toga predati dobiveni simetrični ključ u poziv metode `Dekriptiraj_ToClient_String`, te dekriptirati i *toClient* znakovni niz. Sve dekriptirane veličine dohvaćaju se u varijablama predanima preko reference u argumente odgovarajućih metoda.

Klijent će iz dekriptiranog *toClient* znakovnog niza pomoću metoda za parsiranje *toClient* znakovnog niza dohvatiti registracijski ključ, te programski permutacijski ključ, zatim će pomoću permutacijskog ključa i računalnog kôda (koji je još uvijek sadržan u početno instanciranom objektu tipa `SoftProtClient`) lokalno izračunati registracijski ključ, usporediti ga s onim iz dohvaćenim iz *toClient* znakovnog niza i zaključiti da li su jednaki ili ne (odnosno da li je registracija uspješna ili neuspješna). Registracijski ključevi ispisat će se potom korisniku u veliku tekstualnu kućicu u klijentskoj formi, a u prozoru dijaloga ispisat će se poruka o uspješnoj ili neuspješnoj registraciji. Ukoliko je kroz bilo koji od ovih koraka došlo do pogreške, u prozoru dijaloga ispisat će se obavijest o tome da li se pogreška dogodila na klijentu ili poslužitelju. Klikom na gumb OK u ovom prozoru dijaloga, može se dobiti i detaljnije informacije o pogrešci.



Slika 19. Prikaz pogrešaka u klijentskoj formi

7. TESTIRANJE I ANALIZA OSTVARENOG PROJEKTA

7.1. Testiranje korištenja

Dosad smo spomenuli osnovnu namjenu i načine uporabe programske zaštite ostvarene u sklopu ovog diplomskog zadatka. Dva su glavna scenarija. Prvi je aktivacija, odnosno prva instalacija proizvoda, pri čemu unosimo samo serijski broj, te se generiraju računalni kôd na klijentskoj strani i registracijski ključ na poslužiteljskoj strani i upisuju u bazu podataka. U drugom scenariju radi se o instalaciji već aktiviranog proizvoda, koji je svojom aktivacijom vezan za računalo s kojeg se instalira; ovdje korisnik upisuje serijski broj i računalni kôd u klijentsku formu, oni se šalju na poslužitelj i uspoređuju s već upisanim vrijednostima u bazi podataka.

Međutim, nije dovoljno pretpostaviti da će se program koristiti samo na ova dva načina. Postoji više načina na koje korisnici, slučajno ili namjerno, mogu krivo postaviti parametre koji se šalju. Stoga je potrebno testirati rad aplikacije i adekvatno prilagoditi reakcije na ovakve situacije. U tu svrhu osmislio sam i testirao nekoliko najvjerojatnijih scenarija korištenja klijentske aplikacije. Među njih sam uvrstio i ona dva ispravna načina korištenja.

Početno stanje baze podataka je 16 'praznih' zapisa sa slučajno generiranim 16-znakovnim serijskim brojevima (slika).

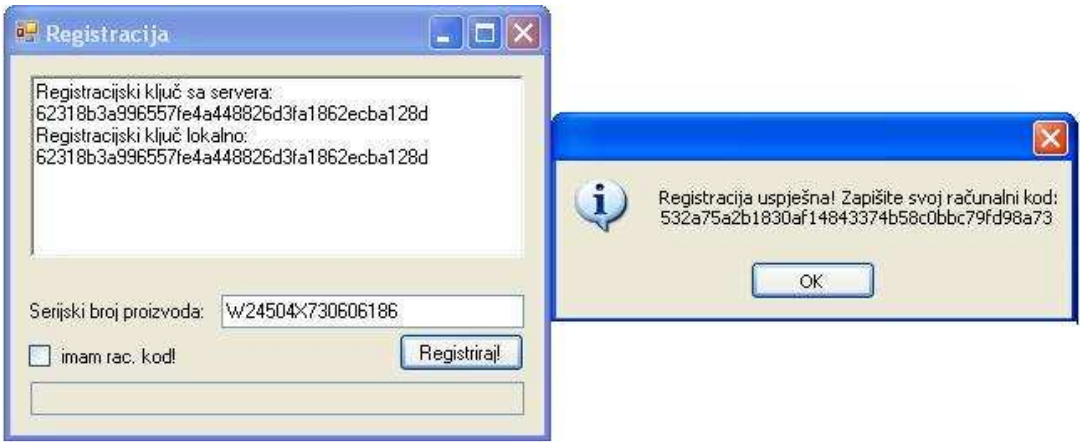
serial	hard_key	soft_key	hard_perm_key	soft_perm_key	aktivacija
W24504X730606186	NULL	NULL	NULL	NULL	False
458J32VJGL610200	NULL	NULL	NULL	NULL	False
76XSN1SN0P6H13O0	NULL	NULL	NULL	NULL	False
5P1J6IGPLDK57T61	NULL	NULL	NULL	NULL	False
Y62YE2CK732T5OF8	NULL	NULL	NULL	NULL	False

Slika 20. Izgled baze podataka prije testiranja aplikacije

Ovo su scenariji i rezultati testiranja:

1) *Korisnik prvi put instalira program koristeći ispravni serijski broj*

   Ako upišemo prvi serijski broj iz baze (na slici označen plavo) i pokrenemo proces registracije, sve će proći ispravno; na poslužitelj se šalje prva od dvije verzije *fromClient* znakovnog niza, ispitivanje aktivacije ne generira nikakvu pogrešku (poslužitelj će pronaći odgovarajući (postojeći) serijski broj u bazi prema poslanom dijelu serijskog broja, a u polju '*aktivacija*' u bazi za ovaj serijski broj upisana je vrijednost *false*), poslužitelj će upisati odgovarajuće vrijednosti u bazu, poslati *toClient* znakovni niz, lokalna usporedba registracijskih ključeva također dobro prolazi i korisnika se obavještava o uspješnoj registraciji.



Slika 21. Klijentska forma nakon uspješne aktivacije

2) *Korisnik prvi put instalira program, ali upisuje pogrešan (nepostojeći) serijski broj (prvi dio serijskog broja (onaj što se šalje) je točan, ali drugi dio nije)*

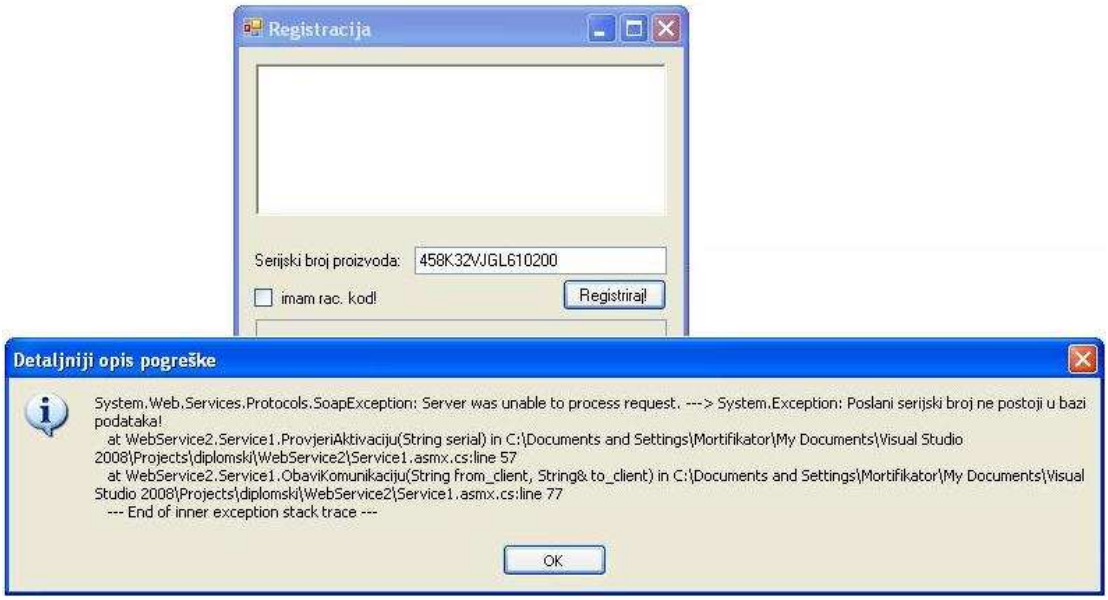
Pod serijski broj u formu upisujemo drugi po redu serijski broj sa slike, ali s npr. izmijenjenim posljednjim znakom (0 mijenjamo u 1). Time dobivamo serijski broj koji ne postoji u bazi podataka Klijent će poslužitelju poslati prvu verziju *fromClient* znakovnog niza, s točnim dijelom serijskog broja, ali netočnim SHA-1 sažetkom. Testiranje aktivacije na poslužitelju proći će (jer metoda `ProvjeriAktivaciju` traži zapis u bazi samo prema poslanom dijelu serijskog broja, te vraća što je upisano pod poljem aktivacija). No, poslije provjere aktivacije poslužitelj dohvaća cijeli pronađeni serijski broj iz baze podataka, računa SHA-1 sažetak i uspoređuje s onim primljenim u *fromClient* znakovni niz. Sažeci se ne slažu i poslužitelj generira pogrešku.

Slika 22. Klijentska forma nakon neuspješnog pokušaja aktivacije s neispravnim sažetkom serijskog broja

3) *Korisnik prvi put instalira program, ali upisuje pogrešan (nepostojeći) serijski broj (prvi dio serijskog broja je netočan)*

Testiramo unos serijskog broja čiji je prvi dio (onaj što se šalje u *fromClient* znakovnom nizu kao čisti tekst) netočan. Ovdje čak nije ni bitno što se nalazi u drugom dijelu serijskog broja, jer poslužitelj neće ni doći do provjere sažetaka, već će se pri provjeri aktivacije stvarati pogreška, jer će metoda `ProvjeriAktivaciju` u bazi

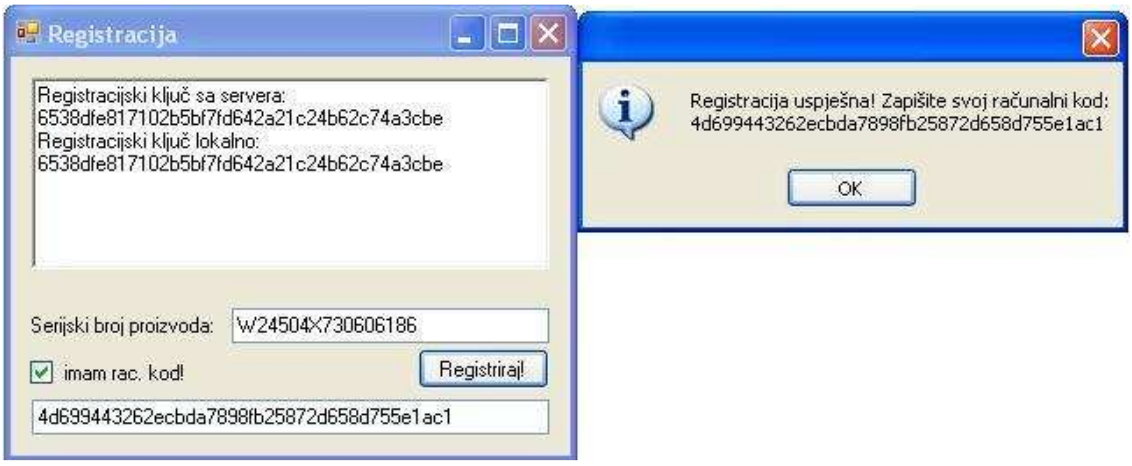
tražiti zapis koji odgovara primljenom dijelu serijskog broja, i neće ga naći, dakle serijski broj ne postoji u bazi podataka.



Slika 23. Klijentska forma nakon neuspješnog pokušaja aktivacije s nepostojećim serijskim brojem

4) Korisnik instalira već aktiviran proizvod s ispravnim računalnim kôdom

U formu upisujemo onaj serijski broj što je aktiviran pri testiranju prvog scenarija i upisujemo računalni kôd koji je ispisan u poruci o uspješnoj registraciji (slika 21)). Sad će nakon pokretanja procesa registracije klijent slati poslužitelju onu drugu verziju *fromClient* znakovnog niza, koja sadržava MAC adresu umjesto računalnog kôda. Pri provjeri aktivacije poslužitelj će pronaći odgovarajući serijski broj u bazi, i „vidjeti“ da je aktiviran. Provjera sažetaka će također proći (jer je serijski broj ispravan). Poslužitelj će izračunati računalni kôd i usporediti ga s onim iz baze, to će također proći uredno. Poslužitelj će poslati *toClient* znakovni niz, usporedba registracijskih ključeva će biti uspješna, i klijent će ispisati na ekranu poruku o uspješnoj registraciji.

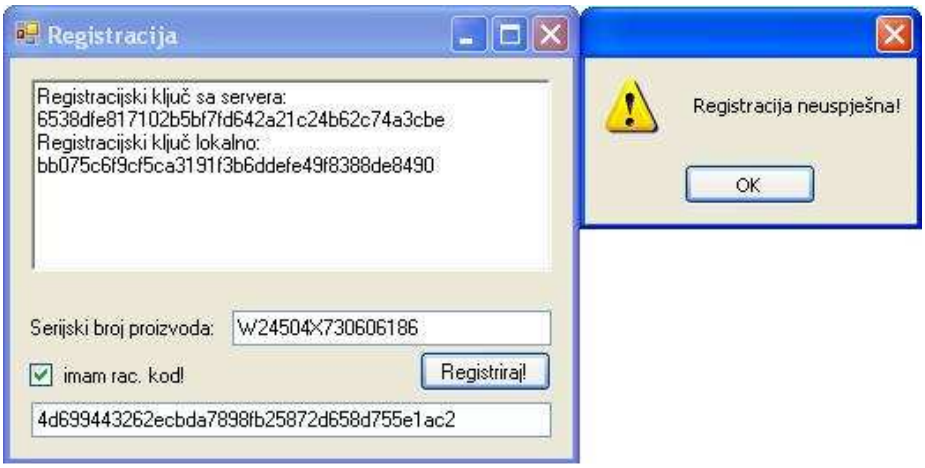


Slika 24. Klijentska forma nakon uspješne registracije pomoću računalnog kôda

5) *Korisnik instalira već aktiviran proizvod, ali upisuje neispravan računalni kôd*

Klijent će poslati poslužitelju *fromClient* znakovni niz formata br. 2, poslužitelj će provjeriti aktivaciju, utvrditi da postoji takav serijski broj, i da je aktiviran, a usporedba sažetaka će također proći, jer je upisani serijski broj ispravan. Čak će i usporedba računalnih kôdova proći, jer je MAC adresa ispravna, dakle pokušaj instalacije došao je s odgovarajućeg računala. Međutim, registracijski ključ dohvaćen na poslužitelju bit će različit od onog koji će izračunati klijent koristeći lokalno upisani računalni kôd, pa će stvar konačno zapeti prilikom usporedbe računalnih kôdova na klijentu. Klijent ispisuje poruku o neuspjehu registracije.

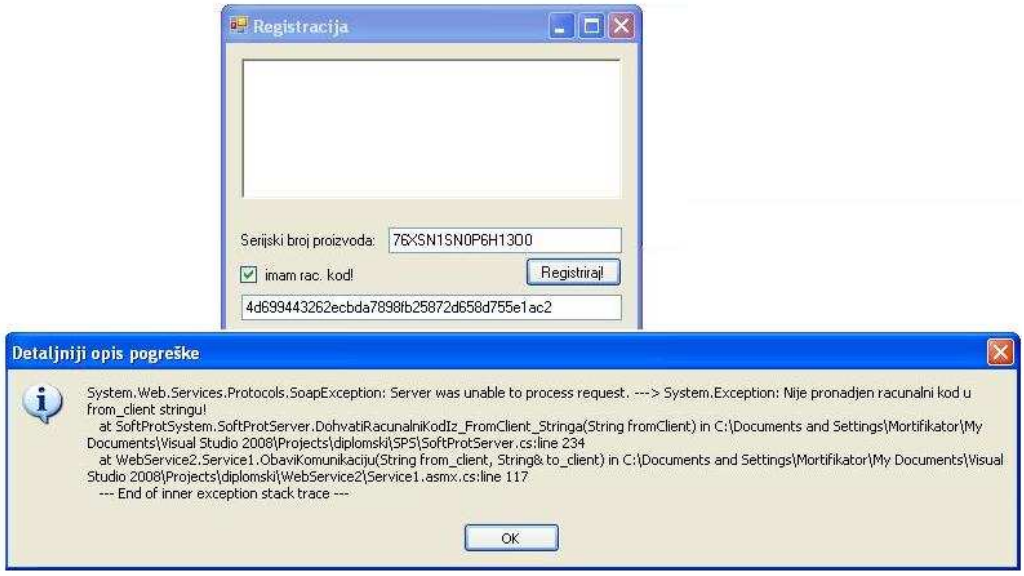
Kod ovog scenarija javlja se jedno pitanje – da li je uopće nužan upis računalnog kôda prilikom instalacije programa, ako je činjenica da je ovdje sve prošlo u redu na poslužiteljskoj strani, dakle utvrđeno je da su računalo i serijski broj odgovarajući u odnosu na prvu instalaciju, a i serijski broj je tajan, pa je dovoljan za verifikaciju? Odgovor je – ovo nije nužno, ali je u svakom slučaju potrebno, jer klijent mora odnekud izračunati registracijski ključ za usporedbu. Uostalom, upis računalnog kôda može se smatrati ekvivalentnim upisu samog registracijskog ključa prilikom instalacije, s tim da se ovako dodatno zaštićuje registracijski ključ, jer ostaje u potpunosti nepoznat.



Slika 25. Klijentska forma nakon neuspjeha registracije radi upisa krivog računalnog kôda

6) *Korisnik instalira program po prvi put, ali upisuje i računalni kôd, kao da je program već aktiviran*

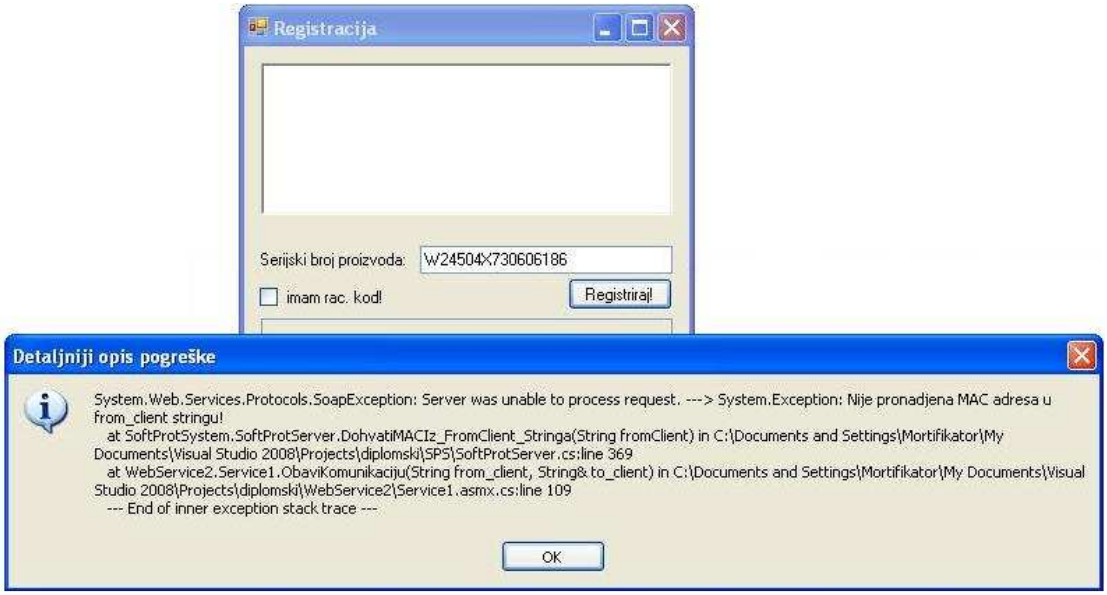
Ovakav scenarij teško da će se ikada desiti, ali je svejedno nešto što se testiranjem mora pokriti. Ovdje će glavni uzrok pogreške biti slanje krivog formata *fromClient* znakovnog niza. Na poslužitelju će proći testiranje aktivacije, dohvaćena vrijednost za aktivaciju bit će *false*. Testiranje sažetaka proći će uredno. Misleći da se radi o prvoj instalaciji programa, poslužitelj će pokušati pronaći u *fromClient* znakovnom nizu računalni kôd, no njega ondje nema, pa nastaje pogreška.



Slika 26. Klijentska forma nakon pogreške radi suvišnog upisa računalnog kôda

7) *Korisnik upisuje serijski broj proizvoda koji je već aktiviran, ali ne upisuje računalni kôd*

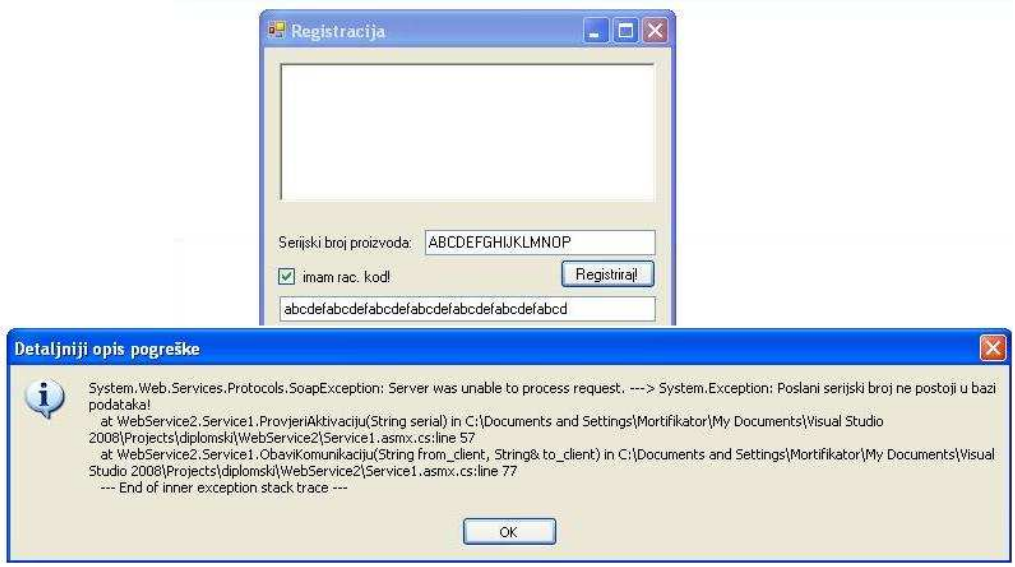
Ovo je vrlo vjerojatna situacija – korisnik jednostavno može zaboraviti upisati računalni kôd i pokušati instalirati program. Što će se desiti? Ovdje će nastati obrnuta situacija od prethodnog slučaja, dakle opet se šalje krivi *fromClient* znakovni niz, ali ovaj put prvi umjesto drugog. Na poslužitelju će uspješno proći i testiranje aktivacije, i provjera sažetaka. U bazi piše da je proizvod aktiviran, pa ovaj put poslužitelj treba dohvatiti iz *fromClient* znakovnog niza MAC adresu klijenta, no neće ju naći, pa će se generirati pogreška.



Slika 27. Klijentska forma nakon pogreške radi izostanka upisa računalnog kôda

8) Korisnik upisuje nepostojeći serijski broj i nepostojeći računalni kôd

U ovom slučaju, situacija će biti slična kao pod 3), samo što će se slati drugi format *fromClient* znakovnog niza. Neovisno o tome, poslužitelj će blokirati algoritam već pri provjeri aktivacije, jer serijski broj nije pronađen u bazi podataka.



Slika 28. Klijentska forma nakon pogreške radi upisa nepostojećeg serijskog broja i računalnog kôda

7.2. Analiza mogućih vrsta napada na ostvareni model zaštite

Zasigurno najbitniji dio testiranja sustava programske zaštite je testiranje otpornosti na razne vrste napada na sustav. Na kraju krajeva, programska zaštita i postoji upravo u tu svrhu. Ranije, u 3. poglavlju ovog rada, gdje sam rekao nešto osnovno o alatima i tehnikama probijanja programske zaštite, prvenstveno sam govorio o napadima lokalnim praćenjem izvođenja programa na asemblerskom nivou, te zaobilaženju ključnih grananja unutar programa. No klasično probijanje programske zaštite (eng. s*oftware cracking*) nije jedini način da se ugrozi na ovakav način ostvarena zaštita. S obzirom da se ovdje radi o modelu građenom na klijent-poslužitelj arhitekturi, koji se pri ostvarivanju zaštite koristi Internetom (koji je sam po sebi nesiguran medij), potrebno je osvrnuti se i na mogućnosti različitih uplitanja treće stranke u komunikaciju, što će uglavnom uključivati prisluškivanje i lažno predstavljanje.

Ovdje ću analizirati oba načina napada. Treći način napada mogao bi biti fizička krađa serijskog ili registracijskog broja, protiv čega se je nažalost nemoguće zaštititi programski. Potrebno je, dakle, da korisnik dobro čuva svoj serijski broj, kao i da fizički pristup poslužitelju, odnosno računalu na kojem se nalazi baza podataka, bude maksimalno ograničen.

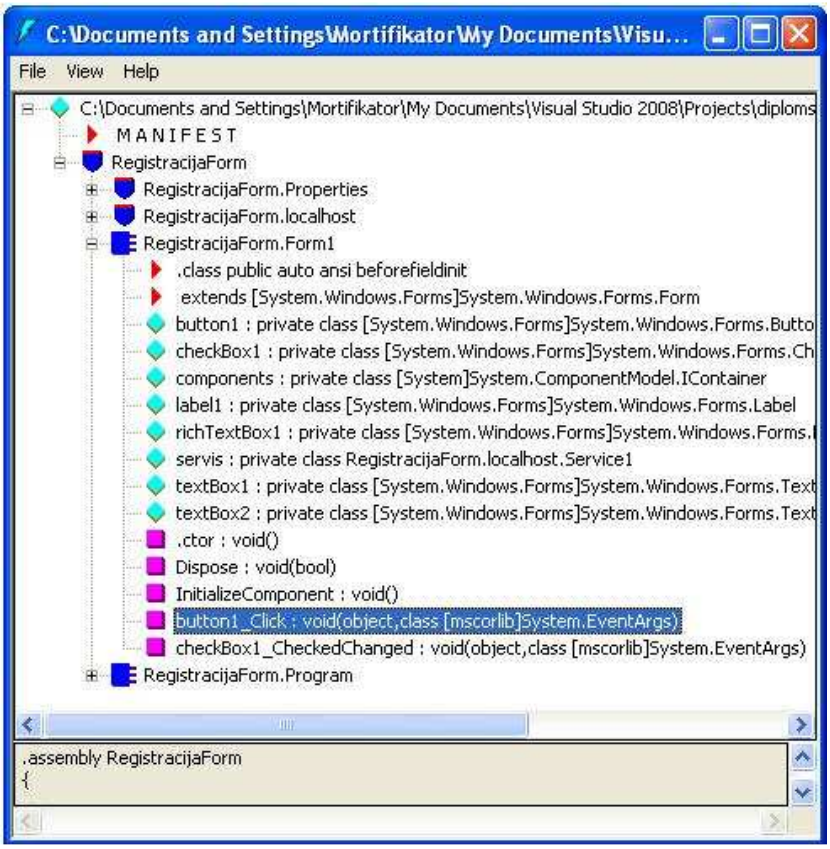
7.2.1. Lokalni napadi – pokušaji probijanja zaštite

Osnove probijanja zaštite pomoću programa za dohvat asemblerskog kôda opisao sam u ranijem tekstu. Osnovna ideja je da se ciljna EXE datoteka (izvršni kôd) otvori u takvom programu, kako bismo u izlazu dobili niz instrukcija asembleru, te pokrene aplikacija i prati njeno izvođenje korak po korak. Najčešće je za uklanjanje zaštite potrebno u dobivenom asemblerskom kôdu pronaći točno mjesto gdje se odlučuje o uspješnosti registracije, dakle nekakav izbor između pozitivne i negativne registracije i skok na odgovarajuće. Ovo smo sve

već vidjeli u primjeru u 3. poglavlju. Razlika je u odnosu na taj primjer što je u ovom slučaju aplikacija izgrađena u .NET okruženju, pa će pri prevođenju biti prevedena u MSIL (eng. ***M**icro**s**oft **I**ntermediate **L**anguage*), odnosno CIL (eng. ***C**ommon **I**ntermediate **L**anguage*) asemblerski jezik (koji sam opisao ranije), pa ćemo za dobivanje asemblerskog izlaza koristiti nekakav IL disasembler.

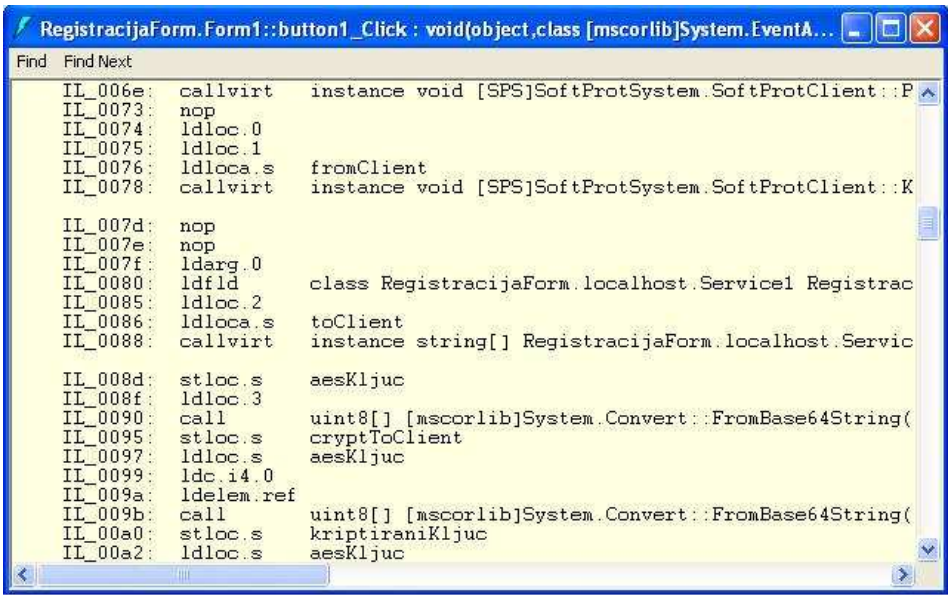
Lokalno probijanje zaštite, nažalost, za ovaj projekt bit će relativno lako. Nema jednostavnog načina za sprečavanje pronalaska odgovarajućeg kôda u asemblerskom izlazu koji se treba izmijeniti kako bi zaštita bila probijena. Iako je u mom projektu kôd vrlo jednostavan i izravan, koliko god mi zakomplicirali kôd, kad-tad će netko tko je dovoljno iskusan pronaći ono što traži. Istina je da, koliko god različitih implementacija zaštite postojalo, uvijek negdje postoji metoda koja verificira uspješnost registracije, dakle u glavnini slučajeva nekakvo grananje gdje se odlučuje da ako je neki uvjet ispunjen, program je registriran, inače nije. Jedino što u ovom slučaju možemo učiniti je integriranje tehnika prevencije probijanja (koje sam ranije opisao, u 4. poglavlju) u naš model programske zaštite. Od ovog ću nešto osnovno kasnije spomenuti, iako ne bih ulazio previše u detalje, jer, kao što sam rekao, sve sam to već opisao u ranijim poglavljima.

Pošto je CIL objektno orijentirani asemblerski jezik, struktura podataka unutar EXE datoteke čiji asemblerski kôd dohvaćamo biti će objektna, pa će IL disasembler prvo dohvatiti objekte pri analizi kôda. Zbog toga je prva stvar na koju treba paziti prilikom pisanja programske zaštite da metode koje eventualno imaju nekakve veze s registracijskim procesom nemaju očita imena, kako bi se netko tko pokušava pronaći mjesto verifikacije u kôdu što teže snašao. IL kôd klijentske EXE datoteke u našem primjeru izgleda ovako:



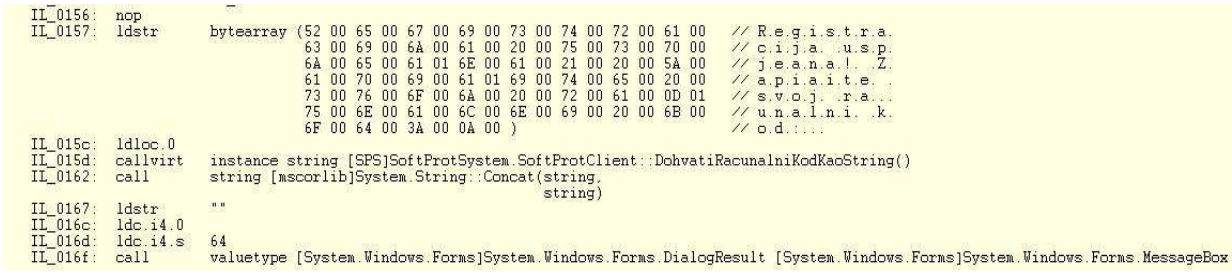
Slika 29. IL disassembler – dohvaćeni objekti klijentske aplikacije

Ovdje smo koristili IL disasembler uključen u standardni SDK (eng. ***S**ource **D**evelopment **K**it*) Microsoftovog Visual Studija. Ovdje na slici vidimo popis objekata u promatranoj izvršnoj datoteci. Ako pogledamo dublje u asemblerski kôd za metodu `button1_Click` (to je metoda koja će nas ovdje zanimati jer se unutar nje nalazi verifikacija registracijskog ključa), vidjet ćemo niz naredbi asemblera u CIL asemblerskom jeziku:



Slika 30. IL disassembler – IL disasemblij metode button1_Click klijentske aplikacije

Ovdje na slici pod adresom IL_0088 poziv je funkcije `ObaviKomunikaciju`. Ovaj se poziv mora zaobići ukoliko se želi probiti ova zaštita, jer se kroz taj poziv šalje serijski broj na poslužitelj, a osoba koja pokušava probiti ovu zaštitu očigledno ne posjeduje takav broj. Ukoliko pogledamo u referencu operacijskih kôdova za CIL, vidjet ćemo da naredba *callvirt* ima kôd 0x6F. Napravimo li sad istu stvar za cijeli blok koji okružuje ovu naredbu, otvorimo izvršnu datoteku u heksadecimalnom editoru, te potražimo odgovarajući uzorak, lako ćemo pronaći koji dio kôda treba zaobići/izmijeniti. Također, uz pronalazak ovog dijela koda potrebno je pronaći i dio kôda gdje se ispisuje obavijest o uspješnoj registraciji, a to je ovdje:

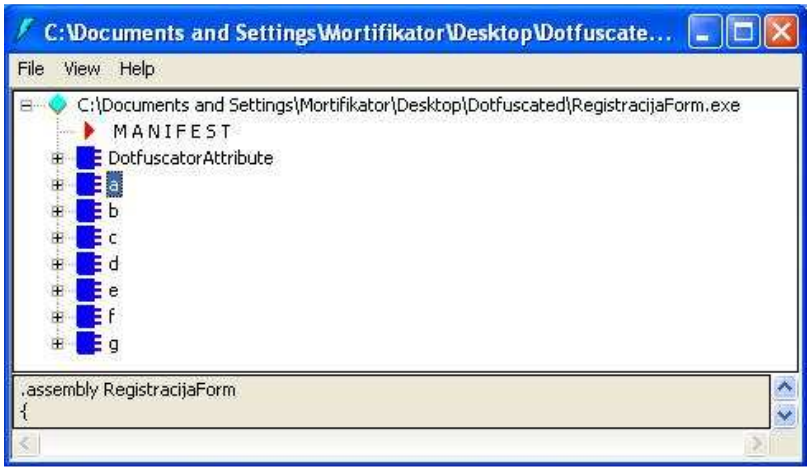


Slika 31. IL disassembler – dio IL asemblerskog kôda gdje se ispisuje poruka o uspješnoj registraciji

Na slici vidimo tekst „*Registracija je uspješna!*“, te poziv *MessageBox*-a. Ideja za izmjenu kôda koja se ovdje nameće je napraviti nekakav skok prije poziva funkcije `ObaviKomunikaciju` sve do trenutka kada je registracija uspješno obavljena. Naravno, sve to izgleda jednostavno ovako na papiru, ali, uz adekvatno znanje asemblerskog jezika CIL, ovdje je potreban i oprez pri umetanju/brisanju/zamjeni asemblerskih naredbi u kôdu, jer se mora paziti na poštivanje „try-catch“ blokova, sadržaj registara (posebno statusnih), sadržaj stôga u trenutku skoka itd., inače će se probijeni program rušiti. Ja sad neću pisati puni tijek probijanja ove zaštite, bitno je znati samo da je ovo neka glavna početna ideja. Zavisno je o iskustvu osobe koja pokušava probiti zaštitu koliko će to biti kompliciranо i koliko će dugo

trajati. Nažalost, kao što sam već nekoliko puta spomenuo, nijedna zaštita nije 100% neprobojna, samo je pitanje vremena, strpljenja i iskustva dok netko ne uspije u naumu probijanja.

Pošto su visoki jezici novog doba ekstremno podložni dekompilaciji i disasembliranju (kao što vidimo iz maloprijašnjeg primjera), prijeko je potrebno koristiti neke posebne metode za zaštitu samog izvornog kôda (koji je intelektualno vlasništvo programera, odnosno kompanije proizvođača). Uz već navedene tehnike prevencije pokušaja probijanja (vidi 4. poglavlje), veoma se često u .NET programiranju koristi tehnika namjernog kompliciranja kôda (eng. *code obfuscation*). Ideja je sljedeća: pošto je IL asemblerski međukod dovoljno jednostavan i izravan da dekompajler bez problema iz njega vraća gotovo 100% točan izvorni kôd u jednom od visokih jezika (.NET prevodilac koristi iscrpne metapodatke pri kreiranju IL kôda), ova tehnika komplicira kôd na taj način da on funkcionalno ostane isti, ali bude gotovo u potpunosti nečitljiv i čini dekompilaciju praktički nemogućom. Tehnika kompliciranja kôda sastoji se od jednostavnih izmjena tipa promjene imena metoda (npr. `ObaviKomunikaciju()`se prevodi u `a()`), do složenijih izmjena poput zamršenja kontrole toka (eng. *flow control obfuscation*; npr. zamjena jednostavnih naredbi s gomilom skokova u cilju otežavanja praćenja toka izvršavanja naredbi), enkripcije znakovnih nizova (eng. *string encryption*) i uvođenja preopterećenja (eng. *overload induction*; tehnika dodjeljivanja istog imena nekonfliktnim metodama, primjerice `DekriptirajSimetricniKljuc` i `Dekriptiraj_ToClient_String` mogu se obje preimenovati u `a()`, jer imaju različit broj argumenata, pa neće biti konflikta). Ovakve modifikacije kôda mogu imati različite posljedice. Primjerice, uklanjanje nepotrebnih metapodataka i nekorištenih članova, te korištenje manjeg broja imena u kôdu mogu ubrzati izvođenje algoritma i smanjiti veličinu izvršne datoteke. S druge strane, zamršenje kontrole toka može znatno usporiti izvođenje osjetljivijih algoritama. Stoga treba izabrati odgovarajući omjer modifikacija koje će se koristiti pri korištenju ove tehnike zaštite kôda. Kompliciranje kôda, naravno, ne radi se ručno, već su to automatske tehnike koje obavljaju danas komercijalni alati, među kojima su poznatiji (iako ih ima stvarno poprilično na tržištu): Dotfuscator (PreEmptive Solutions), .NET Reactor (Eziriz), Xenocode (Code Systems), {smartassembly} (Redgate Software). Cijene ovih programa zaista su varijabilne, iako ih većina košta između 500 i 1000 $. Od navedenih najjeftiniji je .NET Reactor (vjerojatno i najjeftiniji na tržištu) s cijenom od svega 169 $, a najskuplji Dotfuscator s cijenom od 1890 $. Besplatna verzija Dotfuscatora (pod imenom *Dotfuscator Community Edition*) nalazi se integrirana u instalaciji Visual Studija. Ako propustimo asemblij naše klijentske datoteke kroz Dotfuscator, on će ovako izgledati u IL Disassembleru:



Slika 32. IL disassembler – objekti IL kôda izvršne datoteke propuštene kroz Dotfuscator (obfuscated code)

Kao što vidimo, sva su imena (klase, objekti, metode) u asembliju preimenovana. Besplatna verzija Dotfuscatora radi samo preimenovanje, a ne podržava kontrolu toka i enkripciju znakovnih nizova.

7.2.2. Napadi u tijeku komunikacije

S druge strane, opasnost napada na otvorenoj mreži, nešto je sasvim drugo od lokalnog probijanja zaštite. Ovdje naglasak neće biti na zaobilaženju zaštite, odnosno pronalaženju verifikacijske metode u asemblerskom ispisu, već će napadači trud ulagati u pokušaje krađe tajnih informacija koje se prenose preko Interneta. U tom smislu, u računarskom žargonu, ove ćemo napadače zvati *hakerima* (eng. *hacker*), dok će oni napadači što probijaju zaštitu lokalno biti tzv. *krekeri* (eng. *cracker*). Pogledajmo nekoliko mogućih scenarija napada koji se mogu dogoditi.

Upad hakera u poslužiteljsko ili klijentsko računalo

Što se tiče upada u klijentsko računalo, pošto se ništa od tajnih podataka ne čuva na njemu, biti će to po našu programsku zaštitu bezopasno. Međutim, u općenitom smislu, hakerski upadi u računalo nipošto nisu bezopasni, pa je prijeko potrebno korištenje nekakve zaštite od takvih upada na svakom računalu, što se u današnje vrijeme uglavnom postiže pomoću zaštitnog zida (eng. *firewall*).

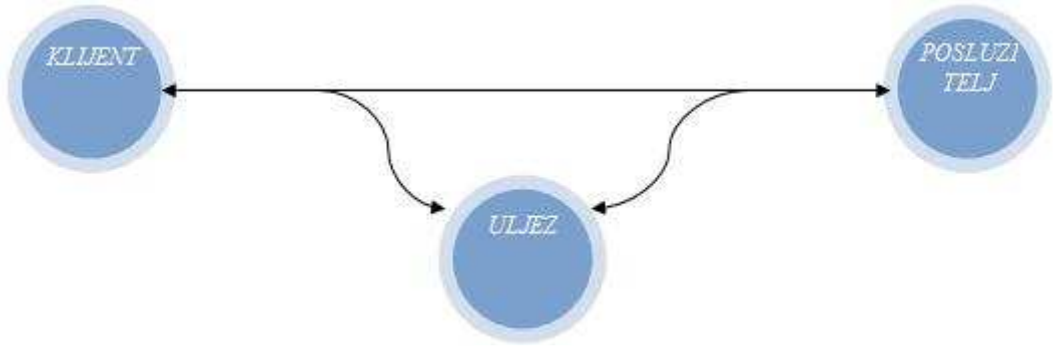
Upad u poslužiteljsko računalo (posebno u računalo na kojem se nalazi baza podataka; poslužitelj i baza podataka ne moraju se nalaziti na istom računalu), s druge strane, može imati katastrofalne posljedice. Iako bilo koji uvid u bazu podataka predstavlja opasnost, najgore bi bilo kad bi se haker domogao neaktiviranih serijskih brojeva, jer bi tada bez ikakvih problema mogao instalirati programski proizvod na bilo kojem računalu, dok god ti serijski brojevi ostaju neaktivirani. U slučaju krađe neaktiviranih serijskih brojeva, eventualno bi legitiman korisnik dobio jedan od tih brojeva, koji je u međuvremenu nelegitiman korisnik iskoristio, pa će se pri instalaciji legitimnom korisniku javljati pogreška. Već aktivirane brojeve bilo bi nešto teže koristiti, jer bi se u tom slučaju ipak moralo intervenirati u sam klijentski program, jer i uz ispravan serijski broj, u tekstu u komunikaciji prema poslužitelju šalje se i MAC adresa, prema kojoj bi poslužitelj u tom slučaju izračunao krivi računalni kôd.

Međutim, ako je haker uspio upasti u bazu i čitati iz nje, lako je moguće da može i pisati u nju, a onda bi stvarno mogao raditi bilo što – prema vlastitoj potrebi bi mogao unositi zapise o registriranim proizvodima, unositi vlastite serijske brojeve, ili izmjenjivati podatke o tuđim instalacijama u vlastitu korist.

Bilo kako bilo, očito je da je prijeko potrebno jakom zaštitom zaštititi poslužitelj s bazom podataka, jer bi u suprotnom posljedice bile katastrofalne. Ne bi bilo loše možda koristiti i enkripciju baze podataka, iako bi to znatno usporilo komunikaciju između poslužitelja i klijenta, odnosno cijeli proces registracije.

Napadi s uljezom u sredini (eng. *man-in-the-middle attack*)

Ova vrsta napada odnosi se na razne pokušaje prisluškivanja komunikacije, izmjenjivanja sadržaja poruke, ili krivotvorenja poruka, odnosno lažnog predstavljanja. Uljez u sredini je neka treća stranka u komunikaciji, i taj uljez se ubacuje u komunikaciju bez da su klijent i poslužitelj svjesni njegove prisutnosti.



Slika 33. Napad tijekom komunikacije s uljezom u sredini (man-in-the-middle attack)

Opasnost koju ovakva vrsta napada može prouzročiti leži prvenstveno u mogućnosti krađe znakovnih nizova koji se šalju. Pošto je *toClient* znakovni niz kriptiran, za njega se ne moramo brinuti, ali što ako se uljez dočepa *fromClient* znakovnog niza? Radi sigurnosnih razloga u *fromClient* znakovnom nizu klijent ne šalje cijeli serijski broj, već samo jedan njegov dio, dok se verifikacija točnosti serijskog broja provjerava na poslužitelju pomoću SHA-1 sažetka, koji se također šalje u istom znakovnom nizu. Odlika sažetka je da je on jedinstven za pojedini znakovni niz, dakle možemo biti sigurni da će sažetak jednog serijskog broja uvijek biti isti, i da niti jedan drugi znakovni niz neće imati isti sažetak. S druge strane, obrtanje procesa izrade sažetka je nemoguće, dakle nemoguće je iz sažetka dobiti original; matematičkim jezikom to znači da je funkcija sažetka takva da nema inverz. Međutim, da li je to dovoljna zaštita? Ako uljez presretne *fromClient* znakovni niz, i dalje će ga moći koristiti, ali bi ga morao uzeti cijelog, upisati vlastiti računalni kôd u njega i poslati ga kroz klijentsku aplikaciju poslužitelju sa svog računala. To bi, naravno, zahtijevalo manipulaciju klijentskom aplikacijom i izvršnim kôdom, što već zadire u lokalne metode probijanja zaštite, što sam analizirao u prethodnom odsječku. Međutim, vidimo da je, uz ovakav vid zaštite, i dalje moguće, iako puno teže, ukrasti i iskoristiti tuđi serijski broj.

Jedna mogućnost napada bez manipulacije kôdom bila bi da uljez presretne poruku od klijenta, jednostavno zamijeni računalni kôd vlastitim, te pusti dalje poruku prema poslužitelju i čeka njegov odgovor. Međutim, s obzirom da je *toClient* znakovni niz kriptiran, ovakav napad bi pao u vodu, jer je za dekripciju potreban privatni ključ koji se nalazi u klijentskoj aplikaciji.

Bilo kako bilo, zaštita izgrađena na ovaj način očito neće biti dovoljna. Bit će solidna, ali ako tražimo nešto što teži neprobojnosti, morat ćemo uvesti dodatne mehanizme zaštite. Jedna od prvih stvari na koju treba pripaziti prilikom izgradnje zaštite protiv *man-in-the-middle* napada je da klijent i poslužitelj budu sigurni da ne komuniciraju s nekim tko u tu komunikaciju ne pripada, prije uopće ikakvog slanja bitnih podataka. Ovo možemo postići uspostavljanjem sigurnog kanala komunikacije korištenjem tzv. TLS (odnosno bivšeg SSL) protokola (eng. *TLS=Transport Layer Security*; *SSL=Secure Sockets Layer*).

TLS i SSL protokoli koriste kriptografiju u svrhu obostrane autentikacije i komunikacijske povjerljivosti. Klasičan TLS primjenjuje jednostranu autentikaciju, primjerice kod Internet preglednika, gdje se autenticira samo poslužitelj na koji se web klijent spaja. Međutim, za potpunu zaštitu preporučuje se obostrana autentikacija, gdje su obje strane sigurne s kim komuniciraju. TLS i SSL kriptiraju segmente transportnog sloja OSI komunikacijskog modela. TLS se odvija u 3 faze: prva je faza tzv. faza rukovanja, gdje se dogovaraju kriptografski algoritmi, ključevi i sl.; druga je faza razmjena ključeva gdje se koriste algoritmi s javnim ključem, a u trećoj fazi se kriptira poruka simetričnim algoritmima i autenticira pomoću sažetaka. TLS koristi 1024 i 2048-bitnu RSA kriptografiju, koju je gotovo nemoguće probiti s današnjom tehnologijom (i u razumnom vremenu), tako da je uz korištenje ovog protokola dovoljno sigurno uklonjen problem uljeza.

8. Zaključak

Vidjeli smo, kroz dosad provedene analize i testiranja programske zaštite ostvarene u sklopu ovog diplomskog rada, da ona, kao i svaka druga programska zaštita, ima svoje prednosti i nedostatke. U svakom slučaju, potrebno je ostvareno programsko rješenje zaštite promatrati uz određenu rezervu, jer je ono nastalo kao primjer kako bi jedna zaštita ovakvog tipa trebala izgledati i funkcionirati. Među ostalim, napravljeno programsko rješenje nudi jedan okvir, okruženje u kojem se može graditi razne varijante modela zaštite na klijent-poslužitelj arhitekturi. Ostvareni okvir zaštite kao takav funkcionalan je za uporabu u svrhu registracije legitimnih kopija programa. Koliko je ta zaštita otporna na razne napade i pokušaje probijanja, ovisit će o implementaciji funkcionalnosti samog klijenta, odnosno poslužitelja, te mehanizmima zaštite koji se koriste. Kad se govori o sprečavanju manipulacije izvršnim ôodom, kao i napada uplitanjem u komunikaciju klijenta i poslužitelja, to je područje dovoljno pokriveno već gotovim komercijalnim i inim rješenjima, od raznih trikova koji se koriste za sprečavanje probijanja programske zaštite, do danas vrlo razvijenih i moćnih protokola mrežne sigurnosti. Pravilnim korištenjem i kombiniranjem tih metoda i protokola, može se napraviti poprilično jaka, gotovo neprobojna zaštita. Unatoč tome, svaka zaštita se, s vremenom i trudom, može probiti, i to se mora uzeti u obzir prilikom izgradnje sustava zaštite. Pritom programer mora imati predodžbu kolike gubitke očekuje u slučaju probijanja zaštite, te koliko je jakost zaštite zaista potrebna, odnosno koliko dugo je proizvođač odlučan odgoditi probijanje zaštite i početak odljeva zarade. U zaštiti ne treba pretjerivati, jer ponekad prekomplicirana rješenja rade probleme u normalnom radu s aplikacijom, a i usporavaju izvođenje.

Projekt `SoftProtSystem` u svojoj trenutnoj izvedbi primjenjiv je za zaštitu nekih manjih i jeftinijih aplikacija, jer je kao amaterski primjer ostvarenja programske zaštite poprilično robustan, sirov i, jednostavno rečeno, osnovan. Usto, funkcija izgradnje računalnog kôda iz MAC adrese i funkcija izgradnje registracijskog ključa iz računalnog kôda prejednostavne su za neku ozbiljnu zaštitu. Logično je da ova zaštita ima dosta nedostataka, kao i da se može ojačati i dopuniti.

Za kraj navest ću nekoliko ideja kako bi se ova zaštita mogla poboljšati:

1) Prvo što bi definitivno trebalo promijeniti su funkcije za računanje računalnog kôda, odnosno registracijskog ključa. Bilo bi uputno koristiti ne samo jednu, već niz slučajnih permutacija, XOR operacija i sličnog. Također bi trebalo kao početni argument za izgradnju računalnog kôda koristiti više parametara računala domaćina, a ne samo MAC adresu. Među potencijalnim kandidatima su serijski brojevi tvrdog diska, matične ploče, BIOS čipa, radne memorije i sl. Također bi se pri generiranju računalnog kôda mogla koristit GUID (odnosno UUID) struktura (koju sam opisao pri opisu metoda klijentske i poslužiteljske klase u 6.1.2. poglavlju), jer ima odlike globalno jedinstvenog identifikatora ovisnog o računalu domaćinu.

2) Postojeće rješenje ne pokriva mogućnost postojanja više mrežnih kartica u sustavu. Ako je takav slučaj, treba uvesti nešto po čemu će se birati jedna od MAC adresa u sustavu koja će se koristiti pri generiranju računalnog kôda; najlogičnije bi rješenje bilo da se od više kartica izabere ona preko koje se računalo spaja na Internet, odnosno preko koje će se slati podaci prema poslužitelju.

3) Baza podataka bi se trebala zaštititi i proširiti različitim dozvolama, kako bi joj se ograničio pristup. Također bi se umjesto pravih ključeva u bazu trebali upisivati

sažeci; na taj se način netko tko bi i uspio upasti u bazu podataka ne bi mogao domoći pravih ključeva.

4) Informacije o registraciji mogle bi se čuvati negdje razasute po operacijskom sustavu i sistemskim direktorijima, pa bi se mogle koristiti pri slučajnim provjerama tijekom rada zaštićenog programa.

5) Iako bi se izgubilo na automatiziranosti, možda bi bilo bolje da se podaci ne razmjenjuju između klijenta i poslužitelja na ovakav način, već da korisnik svoj računalni kôd upisuje na nekakvoj Internetskoj stranici, a ActiveX kontrola izračunava registracijski ključ, koji se zatim nekim sigurnim putem, primjerice e-poštom, šalje korisniku. Registracija bi se mogla vršiti i preko telefona, ali to je ponešto staromodno rješenje, i zahtijeva dogovor s telekomunikacijskim kompanijama.

6) Pošto određene .Net funkcije i API pozivi rade različito (ili ne rade) na različitim OS platformama, bilo bi dobro ispitivati na početku klijentskog programa koji operativni sustav se koristi na klijentu, te prilagoditi program tome. Također, i sam rad (ponašanje) aplikacije trebao bi testirati na više operativnih sustava.

7) Obavezno treba koristiti SSL enkripciju (HTTPS) i obostranu autentikaciju pri komunikaciji klijenta i poslužitelja. Moguće je čak uvesti kao rješenje i upotrebu digitalnih certifikata.

8) Pošto su trenutno svi ključevi u heksadecimalnom obliku, bilo bi dobro radi lakše čitljivosti, koristiti nekakvu vrstu alfanumeričke konverzije, dakle da dobijemo normalan oblik ključa sa znakovima „a-z“, odnosno „1-9“. Također, čitljivosti pomaže i da se ovi ključevi prikazuju korisniku u blokovima od po 5-10 znakova.

9) Pošto .Net nudi među svojim ugrađenim bibliotekama i objekte/strukture za čuvanje tajnih informacija, moglo bi ih se koristiti za spremanje serijskog broja, registracijskog ključa, pa i računalnog kôda tijekom aktivnog rada aplikacije (prilikom rada algoritma zaštite), jer su tajne informacije ugrožene dok god se nalaze u radnoj memoriji računala.

10) Moglo bi se u modelu zaštite predvidjeti korištenje više različitih funkcija za izračunavanje računalnog kôda, odnosno registracijskog ključa, pa da za svakog korisnika klijentska aplikacija pseudoslučajno bira koju funkciju će koristiti. Ovo bi zbunilo eventualne pokušaje izrade automatskih generatora registracijskog ključa.

9. Literatura

Knjige:

[1] P. Cerven: „***Crackproof Your Software***“ (original: „Cracking a jak se proti nemm brinit“), 2001., Češka
[2] E. Butow, T. Ryan: „***C#; Your Visual Blueprint For Building .NET Applications***“, 2002., SAD
[3] H. Henrickson, S. Hofmann: „***IIS 6: The Complete Reference***“, McGraw-Hill/Osborne, 2003., SAD
[4] B. Woody: „***Administrator's Guide To SQL Server 2005***“, Addison Wesley Professional, 2006., SAD
[5] Standard ECMA-335, 4th edition, „***Common Language Infrastructure (CLI), Partitions I to VI***“, 2006., ECMA International, Švicarska

Članci:

[1] Mitchell, Scott: „***Protecting Code, Persisting Data, And More***“, MSDN Magazine, Microsoft Corporation and CMP Media, LLC, 2006.

Internet URL (dostupnost linkova provjerena 7.10.2009.):

[1] Software Protection, Wikipedia, http://en.wikipedia.org/wiki/Copy_protection
[2] Software Cracking, Wikipedia, http://en.wikipedia.org/wiki/Software_cracking
[3] Disassembler, Wikipedia, http://en.wikipedia.org/wiki/Disassembler
[4] Decompiler, Wikipedia, http://en.wikipedia.org/wiki/Decompiler
[5] Debugger, Wikipedia, http://en.wikipedia.org/wiki/Debugger
[6] Hex Editor, Wikipedia, http://en.wikipedia.org/wiki/Hex_editor
[7] Microsoft Macro Assembler, Wikipedia, http://en.wikipedia.org/wiki/Microsoft_Macro_Assembler
[8] EU Copyright Directive, Wikipedia, http://en.wikipedia.org/wiki/EU_Copyright_Directive
[9] Digital Millenium Copyright Act, Wikipedia, http://en.wikipedia.org/wiki/Digital_Millenium_Copyright_Act
[10] Windows Driver Model, Wikipedia, http://en.wikipedia.org/wiki/Windows_Driver_Model
[11] Pseudocode, Wikipedia, http://en.wikipedia.org/wiki/Pseudocode
[12] CIL, Wikipedia, http://en.wikipedia.org/wiki/CIL
[13] Reverse Engineering, Wikipedia, http://en.wikipedia.org/wiki/Reverse_engineering
[14] Obfuscated Code, Wikipedia, http://en.wikipedia.org/wiki/Obfuscated_code
[15] Planet Source Code, http://www.planetsourcecode.com
[16] CodeProject, Free source code and programming help, http://www.codeproject.com
[17] ASP.NET Forums, http://forums.asp.net/
[18] MSDN Forums, SQL Server Category, http://social.msdn.microsoft.com/Forums/en-US/category/sqlserver/
[19] MSDN Forums, .NET Development Category, http://social.msdn.microsoft.com/Forums/en-US/category/netdevelopment
[20] Report On Warez, http://www.mindspring.com/%7Ewin32ch/Warez.htm
[21] Report On Cracks, http://www.mindspring.com/%7Ewin32ch/Crackit.htm
[22] MASM32 Homepage, http://masm32.com/
[23] Eziriz NET Reactor, http://www.eziriz.com

[24] Dotfuscator, http://www.preemptive.com/dotfuscator.html
[25] Xenocode, http://www.xenocode.com
[26] .NET obfuscator {smartassembly}, http://www.smartassembly.com
[27] An introduction in retrieving WMI in C# (C# Programming Tutorial), http://www.geekpedia.com/tutorial73_An-introduction-in-retrieving-WMI-in-Csharp.html
[28] WMI .NET Overview, http://msdn.microsoft.com/en-us/library/ms257340(VS.80).aspx
[29] A History Of Software Protection, Edge Online, http://www.edge-online.com/features/a-history-copy-protection
[30] What's Wrong with Copy Protection, http://www.toad.com/gnu/whatswrong.html
[31] Windows Anti-debug reference, http://www.securityfocus.com
[32] ASPack Software, http://www.aspack.com
[33] Silicon Realms Toolworks, Armadillo Software Protection System, http://www.siliconrealms.com/armadillo
[34] DataRescue, IDA Pro, http://www.datarescue.com
[35] MSDN Library, IL Disassembler, http://msdn.microsoft.com/en-us/library/f7dy01k1(VS.80).aspx
[36] Compuware, SoftICE, http://www.compuware.com
[37] Olly Debugger, http://www.ollydbg.de/
[38] Gpch Software, VB Decompiler, http://www.gpchsoft.com/
[39] Dis# .NET Decompiler, http://netdecompiler.com
[40] DJ Java Decompiler, http://members.fortunecity.com/neshkov/dj.html
[41] Hiew hex editor, http://www.hiew.ru/
[42] XVI32 hex editor, http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm
[43] 010 hex editor, http://www.sweetscape.com/010editor/
[44] Breakpoint Software Inc., Hex Workshop, http://www.bpsoft.com/
[45] Yuri Software, HEdit hex editor, http://www.yurisw.com/HEdit.htm
[46] EU Copyright Directive, http://www.euro-copyrights.org
[47] The Digital Millenium Copyright Act, http://thomas.loc.gov/cgi-bin/query/z?c105:H.R.2281:

Ostvarenje programske zaštite od neovlaštene uporabe

Sažetak

Ključne riječi:

U ovom diplomskom radu proučene su različite metode programske zaštite od neovlaštene i ilegalne uporabe. Opisani su različiti načini ostvarivanja programske zaštite, kao i mogućnosti ugrožavanja i probijanja te zaštite. Rad sadrži i listu i generalni opis programskih alata koji se koriste za probijanje, kao i alata za automatsko generiranje programske zaštite, te trikove za sprečavanje pokušaja dohvaćanja i praćenja izvornog kôda s ciljem probijanja zaštite (sprečavanje obrnutog inženjerstva). U sklopu rada ostvarena je kombinirana programska zaštita temeljena na klijent-poslužitelj arhitekturi.

Title: Realization of software protection against unauthorized use

Summary

Keywords:

This thesis studies various methods of software protection against unauthorized and illegal use. It describes means of realization of software protection, as well as threatening and cracking possibilities. The thesis includes a list and a general description of software tools used for cracking, as well as tools for automatic software protection generation, and anti-disassembling and anti-debugging tricks (reverse engineering prevention). Within this thesis is an implementation of a combined software protection based on a client-server architecture.