Reversing .NET

by LaFarge / RLTeam

Introduction

Iako smo do sada svi verovatno pisali ponešto o reversingu *native code*, tj. IA-32 kompajliranih programa, došlo je vreme da napišem jedan opširniji tekst o .NET-u i reversingu .NET programa. Ako se sećate mog tutorijala o ACCOUNT Knjigovodstvu, u kojem sam opisao kako reversovati program uz pomoć dekompajlera, i topic-a na starom RLabs forumu u kojem sam spomenuo da sam se malo ozbiljnije „bacio“ u .NET vode, tj .NET programming i reversing i da ću možda napisati jedan *allaround* tekst o .NET-u. Evo, rešio sam da ispunim dato obećanje, i objavim ovaj tekst u našem dragom pHearless eZine-u.

Do sada smo uglavnom raspravljali IA-32 platformu, vreme je da krenemo u raspravu o novim razvojnim platformama koje bi mogle biti veoma popularne u budućnosti. Postoji mnogo takvih platformi. Mogao bih sada pisati o drugim operativnim sistemima koje rade pod IA-32 platformom kao npr. Linux, ili o potpuno drugačijim platformama koje koriste poptuno drugačiji operativni sistem i arhitekturu procesora, kao npr. Apple Macintosh. Pored operativnih sistema i procesorskih arhitektura, takođe postoje i high-level platforme koje koriste svoj poseban asemblerski jezik i koje mogu da rade na svim platformama. To su *virtual-machine-based* platforme kao što su Java i .NET.

Iako je Java postala moćan i popularan razvojni jezik, ovaj tekst ćemo da fokusiramo na .NET. Odlučio sam se za taj korak iz nekoliko razloga. Jedan od njih je i taj da je Java je prisutna na sceni duže od .NET-a i reversing Jave je dosta dobro pokrivena tema.

U ovom tekstu ćemo opisati neke od osnovnih tehnika reversinga .NET programa. Da bi ovo postigli, moraćemo da se upoznamo sa osnovnim pravilima .NET platforme kao i sa MSIL jezikom. Koristiću jednostavan MSIL kod kao i u mom predhodnom tekstu *Decompiling It!*. Na kraju, upoznaću vas sa nekim alatima specijalizovanim za .NET (i za ostale *bytecode* platforme) kao što su obfuskatori i dekompajleri.

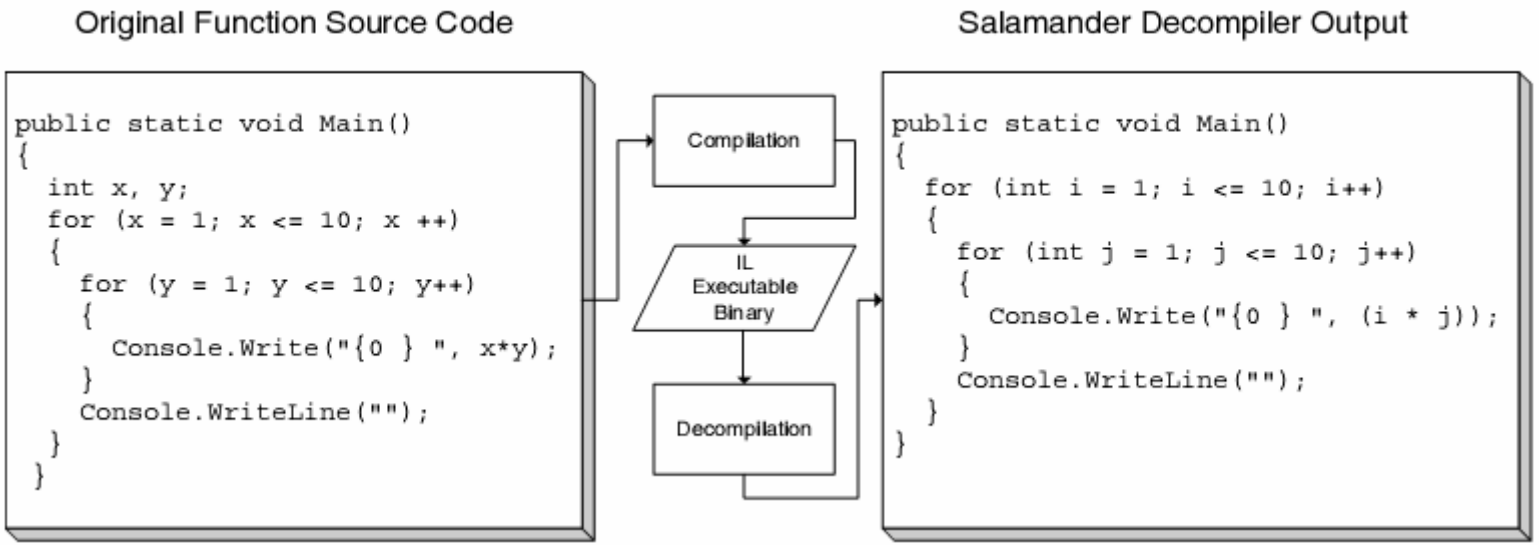
Nadam se da ste se sada već navikli na moj stil pisanja, malo engleski, malo srpski (kao i na RLabs forumu :D), tako da se nadam da nećete imati problema u praćenju ovog teksta. E pa, srećan vam .NET reversing. :-p

Osnovna Pravila

Da razjasnimo jednu stvar! Reversing .NET aplikacije je potpuno drugačiji proces od reversinga običnih IA-32 aplikacija. U osnovi, reversing .NET aplikacije je neverovatno prost proces. .NET programi su kompajlirani u *intermediate* jezik (ili *bytecode*) koji se zove MSIL (Microsoft Intermediate Language). MSIL je veoma detaljna reprezentacija koda i ima daleko više informacija o originalnom programu od IA-32 kompajliranih programa. Ti detalji se odnose na punu definiciju svake data strukture u programu, zajedno sa imenima skoro svakog simbola korišćenog u programu. Imena svakog objekta, data member-a i member funkcija su uključeni u svakoj .NET aplikaciji. Tako .NET *runtime* (CLR) pronalazi te objekte *at runtime*.

Ovo ne samo da pojednostavljuje proces reversinga programa jednostavnim čitanjem MSIL koda, ali i otvara dosta reversing načina i prilaza. Postoje dekompajleri koji mogu da nam daju tačne source kodove za svaku .NET aplikaciju. Izlazni kod je veoma čitljiv, zbog toga što su originalna imena simbola sačuvana u programu, kao i zbog detaljnih informacija o programu koje se nalaze u kompajliranom .EXE-u. Ove informacije mogu da iskoriste dekompajleri i da rekonstruišu *flow & logic* programa i detaljnu informaciju o objektima i data tipovima. Sledi grafikon koji prikazuje kako izgleda kod C# aplikacije dekompajliran Salamander dekompajlerom. Obratite pažnju kako je skoro svaki važan detalj source koda sadržan u dekompajliranom kodu (imena lokalnih varijabli su izgubljene ali ih Salamander „pametno“ zamenjuje sa *i* i *j*).

Zbog high-level transparentnosti .NET programa, koncept obfuskacije .NET programa je postao uobičajen i više nego što je slučaj sa IA-32 programima. Čak šta više, Microsoft isporučuje obfuskator sa svojom .NET platformom, *Visual Studio .NET*. Kao što vidimo iz sledećeg grafikona, ako isporučujete vašu .NET aplikaciju bez prethodne obfuskacije iste, bolje da ste poslali i source kod. :D



Grafikon br. 1
Izgled source-a i dekompajliranog koda
(Salamander Decompiler)

Reversing .NET by LaFarge / Team ICU © 2006

3

Osnove .NET-a

Za razliku od nativnih mašinskih programa, .NET programi se moraju izvršavati u posebnom okruženju. Ovo okruženje, koje se zove *.NET Framework*, radi ako posrednik između .NET programa i „ostatka sveta“. .NET Framework je u suštini *Software Execution Environment* u kojem rade .NET programi, i sastoji se iz 2 komponente:

- *Common Language Runtime (CLR)*
- *Class Library*

CLR je okruženje koje učitava i proverava .NET programe i u suštini predstavlja virtuelnu mašinu unutar koje se .NET programi izvršavaju.

Class Library je ono što .NET programi koriste za komunikaciju sa „ostalim svetom“. To je hijerarhija klasa koje nude različite usluge, kao npr. *user-interface* servise, networking, file I/O, string management, itd...

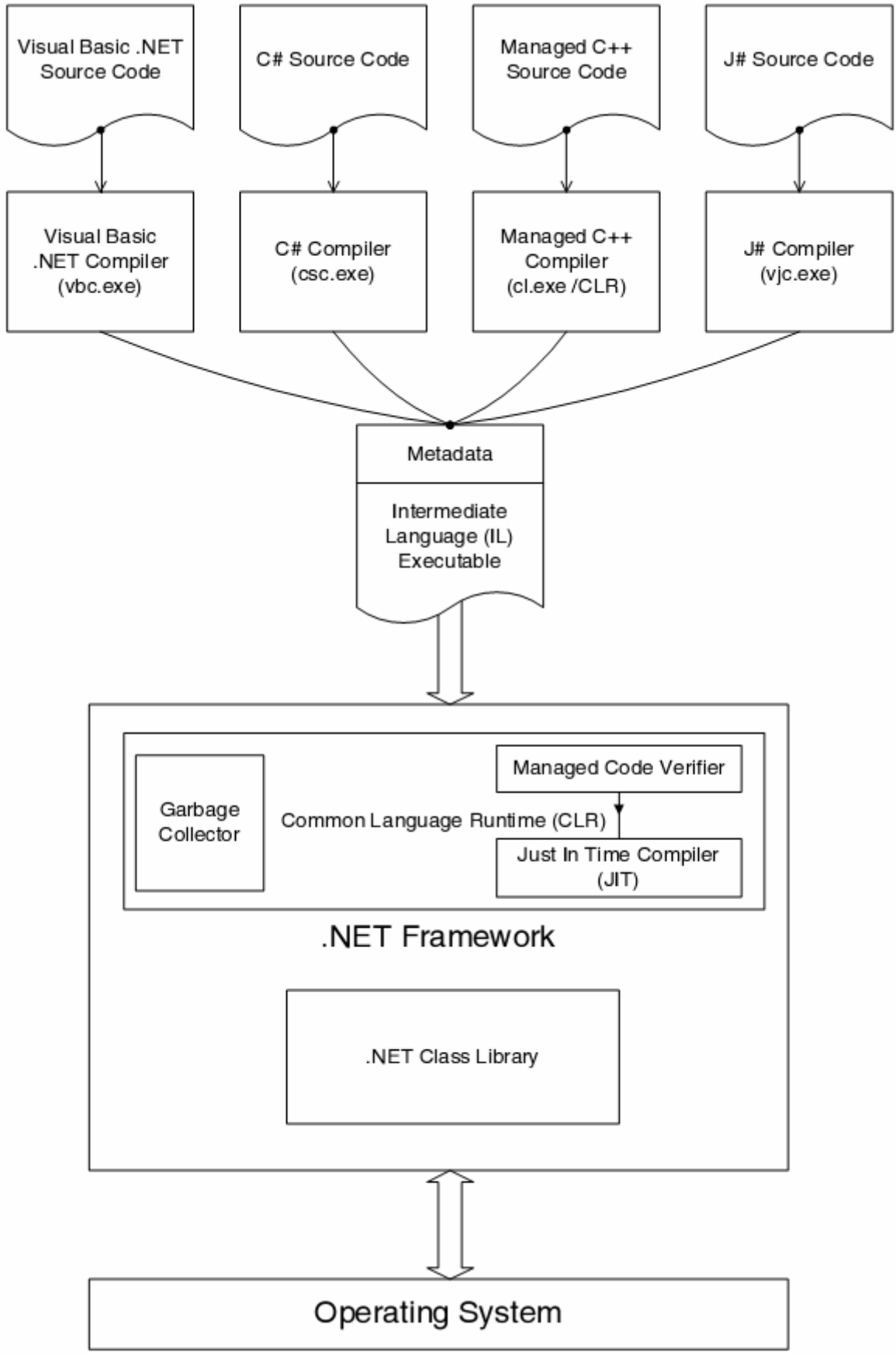
Idući grafikon pokazuje povezanost između raznih komponenti koje zajedno čine .NET platformu.

Za .NET binarni modul se često kaže *assembly*. Asembli sadrži kombinaciju IL koda i *metadata*-e. Metadata je specijalni data blok koji sadrži data type informacije o raznim objektima korišćenim u asembliju kao i tačne definicije svih objekata u programu (uključujući i lokalne varijable, parametre metoda...). Asembli se pokreće od strane CLR-a koji učitava metadatu u memoriju i kompajlira IL kod u *native-code* uz pomoć just-in-time (JIT) kompajlera.

Managed Code

Managed Code je svaki kod koji je verifikovan od strane CLR-a za *security, type-safety i memory usage*. Managed Code se sastoji od dva osnovna .NET elementa: MSIL koda i metadata-e. Kombinacija ta dva elementa omogućava CLR-u da uopšte pokrene managed kod. U svakom trenutku CLR je „svestan“ sa kakvim podacima radi program. Na primer, u običnim IA-32 kompajlerima kao što su C i C++, strukturama podataka se pristupa učitavanjem pointera u memoriju i izračunavanjem specifičnog offseta u strukturi kojem se pristupa. Procesor nema predstavu šta ta struktura predstavlja i da li je adresa kojoj se pristupa validna ili ne.

Kada CLR pokrene neki program, u svakom momentu je svestan o skoro svakom data tipu u programu. Metadata sadrži informacije o definicijama klasa, metodama i njihovim parametrima i o tipovima svake lokalne varijable u svakoj metodi. Te informacije omogućavaju CLR-u da proveri operacije koje radi IL kod i da proveri da li su uopšte ispravne. Na primer, kada asembli koji sadrži managed kod želi da pristupi nekom članu polja (array), CLR može lako da proveri veličinu polja i da triggeruje exception ako je član van granica polja.



Grafikon br. 2
Veze između CLR-a i različitih .NET programskih jezika

.NET Programski Jezici

.NET nije vezan ni za jedan jezik (osim IL-a), i postoje .NET kompajleri koji podržavaju brojne programske jezike. Ovo su najpopularniji programski jezici koji se koriste u .NET okruženju:

C#
C Sharp je jezik koji je dizajniran sa ciljem da bude osnovni .NET jezik. Sintaksa mu je slična C++ ali je funkcionalno više sličnija Javi nego C++-u. I Java i C# su objektno orijentisani, kao i *type-safe*, što znači da ne dozvoljavaju pogrešno korišćenje tipova. Naravno, oba jezika rade sa *garbage collector*-om.

Managed C++
Managed C++ predstavlja ekstenziju Microsoft-ovog C / C++ kompajlera, koja može da proizvodi *managed* IL programe iz C++ koda.

Visual Basic .NET
Microsoft je kreirao VB .NET i time eliminisao VB Virtual Machine (VBVM) komponentu tj. MSVBVM60.DLL koji je omogućavao pogretanje VB programa. Sada VB programi rade pod CLR-om, što znači da su sada VB programi isti kao i C# i Managed C++ programi. Svi se sastoje od managed IL koda i metadata-e.

J#
J Sharp je jednostavno implementacija Java-e za .NET. Microsoft je obezbedio Java-compatible kompajler za .NET koji proizvodi IL programe umesto Java *bytecode*-a. Ideja je očigledno portovanje Java programa za .NET

Common Type System (CTS)

Common Type System upravlja organizacijom data tipova u .NET programima. Postoje dva osnovna data tipa: vrednosti i reference. Vrednosti su tipovi koji predstavljaju stvarnu vrednost podatka, dok su reference upućuju na stvarne vrednosti, nešto kao pointeri. Vrednosti se tipično nalaze na stack-u ili unutar nekog drugog objekta, dok se sa referencama stvarni objekti uglavnom nalaze u heap bloku koji se oslobađa automatski od strane *garbage collector*-a (ovo je pojednostavljen opis, ali će poslužiti za sada).

Tipično upotreba vrednosti je za *built-in* tipove poput integer-a, ali programer može da definiše i sopstvene tipove vrednosti. Ovo je dobro samo kod manjih data tipova, jer se mora napraviti duplikat data-e kada se prosleđuje drugim metodama itd… Veći data tipovi koriste reference jer se tada samo prave duplikati referenci, a ne vrednosti.

Takođe, za razliku od vrednosti, reference su *self-describing*, što znači da sama referenca sadrži informaciju o tipu objekta koji referencira. Ovo je suprotno od tipa vrednosti (koji ne nosi informaciju o tipu).

Zanimljiva stvar kod CTS-a je koncept nazvan *boxing* i *unboxing*. Boxing je proces konvertovanja strukture tipa vrednosti u objekat tipa reference. Interno, ovo radi tako što se napravi duplikat objekta koji se referencira i napravi referenca ka objektu. Ideja je ta da se *boxed* objekat može proslediti bilo

kojoj metodi koja očekuje *generic* referencu objekta kao input. Setite se da tip reference nosi *type* informaciju sa sobom, tako da ako uzme referencu ka objektu kao input, metod može da odredi tip objekta *at runtime*. Ovo nije moguce sa tipom vrednosti. Unboxing je obrnut proces.

Intermediate Language (IL)

Obični .NET programi se distribuiraju u među-formi koja se zove *Common Intermediate Language (CIL)* ili *Microsoft Intermediate Language (MSIL)*, ali mi ćemo je zvati IL. .NET programi imaju dve faze pri kompajliranju. Prvo se program iz originalnog source koda kompajlira u IL kod, a potom, kada se pokrene, kompajlira u *native* kod od strane JIT kompajlera. Sledeće sekcije govore o nekim osnovnim low-level .NET konceptima kao što su *Evaluation Stack* i *Activation Record*, kao i upoznavanje sa IL-om i najvažnijim instrukcijama. Na kraju, pokazaću par IL primera i analiziraću ih.

Evaluation Stack (ES)

Razgovarajući sa SenseiX-om i sort-om na IRC-u, inače diplomiranim .NET programerima, došli smo i do jedne rasprave o *Evaluation Stack*-u. Citiraću šta su mi rekli:

„The evaluation stack is used for managing state information in .NET programs!“.

IL koristi evaluation stack na sličan način kao što IA-32 instrukcije koriste registre – za čuvanje privremenih podataka kao što su input i output podaci za instrukcije. Verovatno najvažnija stvar koju treba razumeti je da *evaluation stack u stvari ne postoji*! Zato što se IL kod nikad ne interpretira *at runtime* i zato što se uvek prvo prevede u native code pre nego što se startuje, evaluation stack jedino postoji u JIT fazi.

Za razliku od IA-32 stack-a, na koji smo se navikli, ES se ne sastoji od 32bit-nih vrednosti ili bilo kojih drugih veličina. Jedna vrednost u stack-u može da sadrži bilo koji data tip, uključujući i kompletne strukture. Mnoge instrukcije u IL setu su polimorfne, tako da mogu da rade sa različitim data tipovima. Ovo znači da, na primer, aritmetičke instrukcije mogu da ispravno rade i sa *floating-point* i sa integer operandima. Nema potrebe da se instrukciji otvoreno definiše tip podatka koji očekuje – JIT će da obavi neophodnu *data-flow* analizu i utvrditi data tipove za svaki operand koji se prosleđuje instrukciji.

Da bi u potpunosti razumeli IL „filozofiju“, morate da se naviknete na ideju da je CLR „*stack machine*“, što znači da IL instrukcije koriste ES kao što IA-32 ASM instrukcije koriste registre. Praktično svaka instrukcija ili POP-uje vrednost iz stack-a ili PUSH-uje neku vrednost u stack – tako IL instrukcije pristupaju svojim operandima.

Activation Records (AR)

Isto tako, SenseiX i sort su mi definisali *Activation Records*:

„Activation Records are data elements that represent the currently running function, much like a stack frame in native programs.“

Dakle, AR sadrži parametre prosleđene tekućoj funkciji, kao i sve lokalne varijable u toj funkciji. Za svaki poziv funkciji, alocira se i inicijalizuje novi AR. U većini slučajeva CLR alocira AR na stack-u, tako da znači da je AR isto što i stack frame sa kojim se srećemo u nativnim ASM programima. IL set instrukcija ima specijalne instrukcije za pristup trenutnom AR-u parametara funkcije, kao i za lokalne varijable. AR se automatski alociraju sa IL instrukcijom *CALL*.

IL Instrukcije

Sada ćemo da vidimo neke najinteresantnije IL instrukcije, čisto da dobijete sliku o jeziku i kako izgleda. Sledeća tabela sadrži neke od „najpopularnijih“ instrukcija u IL setu. Kompletan IL set sadrži preko 200 instrukcija.

INSTRUCTION NAME	DESCRIPTION
`ldloc`—Load local variable onto the stack `stloc`—Pop value from stack to local variable	Load and store local variables to and from the evaluation stack. Since no other instructions deal with local variables directly, these instructions are needed for transferring values between the stack and local variables. `ldloc` loads a local variable onto the stack, while `stloc` pops the value currently at the top of the stack and loads it into the specified variable. These instructions take a local variable index that indicates which local variable should be accessed.
`ldarg`—Load argument onto the stack `starg`—Store a value in an argument slot	Load and store arguments to and from the evaluation stack. These instructions provide access to the argument region in the current activation record. Notice that `starg` allows a method to write back into an argument slot, which is a somewhat unusual operation. Both instructions take an index to the argument requested.
`ldfld`—Load field of an object `stfld`—Store into a field of an object	Field access instructions. These instructions access data fields (members) in classes and load or store values from them. `ldfld` reads a value from the object currently referenced at the top of the stack. The output value is of course pushed to the top of the stack. `stfld` writes the value from the second position on the stack into a field in the object referenced at the top of the stack.
`ldc`—Load numeric constant	Load a constant into the evaluation stack. This is how constants are used in IL—`ldc` loads the constant into the stack where it can be accessed by any instruction.
`call`—Call a method `ret`—Return from a method	These instructions call and return from a method. `call` takes arguments from the evaluation stack, passes them to the called routine and calls the specified routine. The return value is placed at the top of the stack when the method completes and `ret` returns to the caller, while leaving the return value in the evaluation stack.

`br` – Unconditional branch	Unconditionally branch into the specified instruction. This instruction uses the short format `br.s`, where the jump offset is 1 byte long. Otherwise, the jump offset is 4 bytes long.
`box`—Convert value type to object reference `unbox`—Convert boxed value type to its raw form	These two instructions convert a value type to an object reference that contains type identification information. Essentially `box` constructs an object of the specified type that contains a copy of the value type that was passed through the evaluation stack. `unbox` destroys the object and copies its contents back to a value type.
`add`—Add numeric values `sub`—Subtract numeric values `mul`—Multiply values `div`—Divide values	Basic arithmetic instructions for adding, subtracting, multiplying, and dividing numbers. These instructions use the first two values in the evaluation stack as operands and can transparently deal with *any* supported numeric type, integer or floating point. All of these instructions pop their arguments from the stack and then push the result in.
`beq`—Branch on equal `bne`—Branch on not equal `bge`—Branch on greater/equal `bgt`—Branch on greater `ble`—Branch on less/equal `blt`—Branch on less than	Conditional branch instructions. Unlike IA-32 instructions, which require one instruction for the comparison and another for the conditional branch, these instructions perform the comparison operation on the two top items on the stack and branch based on the result of the comparison and the specific conditional code specified.
`switch`—Table switch on value	Table switch instruction. Takes an `int32` describing how many case blocks are present, followed by a list of relative addresses pointing to the various case blocks. The first address points to case 0, the second to case 1, etc. The value that the case block values are compared against is popped from the top of the stack.
`newarr`—Create a zero-based, one-dimensional array. `newobj`—Create a new object	Memory allocation instruction. `newarr` allocates a one-dimensional array of the specified type and pushes the resulting reference (essentially a pointer) into the evaluation stack. `newobj` allocates an instance of the specified object type and calls the object's constructor. This instruction can receive a variable number of parameters that get passed to the constructor routine. It should be noted that neither of these instructions has a matching "free" instruction. That's because of the garbage collector, which tracks the object references generated by these instructions and frees the objects once the relevant references are no longer in use.

Listing br.1
Neke od osnovnih IL instrukcija

Primeri IL Koda

Sada ćemo da vidimo par prostih primera sekvenci IL koda, da dobijete „osećaj“ za jezik. Imajte u vidu da se retko ukazuje potreba za analizom *raw*, non-obfuscated IL koda na ovaj način – dekompajler će da proizvede dosta čitljiviji kod. Jedini slučaj kada ćete da alaizirate ovakav kod je kada je IL kod obfuscated i kada se ne može dekompajlirati kako treba.

Counting Items Primer

Sledeća rutina je proizvedena od strane ILdasm-a, IL disasemblera koji standardno ide uz *.NET Framework SDK*. Originalan kod je pisan u C#, mada to i nema nekakav značaj. Ostali .NET programski jezici bi obično proizveli identičan ili veoma sličan kod:

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  .maxstack  2
  .locals init (int32 V_0)
  IL_0000:   ldc.i4.1

  IL_0001:   stloc.0
  IL_0002:   br.s       IL_000e

  IL_0004:   ldloc.0
  IL_0005:   call       void [mscorlib]System.Console::WriteLine(int32)
  IL_000a:   ldloc.0
  IL_000b:   ldc.i4.1
  IL_000c:   add
  IL_000d:   stloc.0
  IL_000e:   ldloc.0
  IL_000f:   ldc.i4.s   10
  IL_0011:   ble.s      IL_0004

  IL_0013:   ret
} // end of method App::Main
```

Listing br. 2
Primer IL koda, generisan sa ILdasm-om

Listing br. 2 počinje sa par definicija koje se odnose na metodu. Metod je specifikovan kao *.entrypoint*, što znači da je to prvi kod koji se izvrši kada se program pokrene. *.maxstack* izraz određuje maksimalan broj stavki koje ova rutina učitava u ES. Vredi napomenuti da specifična veličina stavke ovde ne igra nikakvu ulogu – ne predpostavljajte da su 32bit-ne niti bilo kakve; izraz predstavlja broj individualnih stavki, bez obzira na njihovu veličinu. Iduća linija definiše lokalne varijable. Ova funkcija ima samo jednu lokalnu varijablu tipa *int32* i imena *V_0*. Imena varijabli su stvari koje se uglavnom uklanjaju od strane kompajlera.

Rutina startuje sa *ldc* instrukcijom koja učitava vrednost 1 u ES. Sledeća instrukcija je *stloc.0*, koja POP-uje vrednost sa vrha stack-a u lokalnu varijablu broj 0 (tj. *V_0*), koja je prva i jedina varijabla u programu. Znači, efektivno smo učitali vrednost 1 u varijablu *V_0*. Primećujete kako je ova sekvenca duža od identične sekvence u ASM kodu? Potrebne su nam 2 instrukcije da bi učitali konstantu u lokalnu varijablu. CLR je „*stack machine*“, sve prolazi kroz ES.

Procedura dalje bezuslovno „skače“ na adresu *IL_000e*. Ciljna instrukcija je određena korišćenjem relativne adrese od kraja trenutne instrukcije. Instrukcija grananja koja se ovde koristi je *br.s*, koja predstavlja kratku verziju, što znači da je relativna adresa predstavljena jednim bajtom. Ako je udaljenost između trenutne i ciljne intrukcije veća od 255 bajtova, kompajler bi koristio *br* instrukciju koja koristi *int32* parametar za definisanje relativne adrese „skoka“. Kratka verzija instrukcije se koristi da bi se kod učinio što manjim.

Kod na adresi *IL_000e* počinje učitavanjem dve vrednosti u ES, vrednost lokalne varijable br. 0, koja je maločas inicijalizovana sa vrednošću 1, i konstante 10. Onda se te dve vrednosti porede sa instrukcijom *ble.s*. Ovo je *branch if lower or equal* instrukcija koja radi i poređenje i jump, za razliku od IA-32 koda kojem je potrebno 2 instrukcije, jedna za poređenje i jedna za grananje. CLR poredi drugu vrednost po redu na stack-u sa onom na vrhu, dakle *lower or equal* grananje će se desiti ako je vrednost lokalne varijable br. 0 (koja je prethodno inicijalizovana sa vrednošću 1) manja ili jednaka konstanti 10. Pošto znamo da je vrednost lokalne varijable br. 0 = 1, grananje će se sigurno desiti, bar kada se kod po prvi put izvrši. Na kraju, da bi *ble.s* instrukcija bila u mogućnosti da uporedi argumente koji su joj prosleđeni, oni moraju biti POP-ovani iz stack-a. Ovo je tačno za skoro svaku IL instrukciju koja prima argumente preko ES-a – ti argumenti se više neće nalaziti na stack-u kada se instrukcija završi.

Pod pretpostavkom da se desilo grananje, kod se nastavlja na adresi *IL_0004*, gde rutina poziva *WriteLine* funkciju koja je deo *class library*-ja. *WriteLine* prikazuje liniju teksta u prozoru *console-mode* aplikacije. Funkcija prima jedan parametar, koji je vrednost naše lokalne varijable. Kao što predpostavljate, vrednost je prosleđenja korišćenjem ES-a. Jedna stvar vredna pomena je i ta da funkcija prima integer vrednost a štampa tekst?. Ako pogledamo liniju sa koje je izvršen poziv videćemo ovo:

void [mscorlib]System.Console::WriteLine(int32)

Ovo je prototip specifične funkcije koja je pozvana. Parametar koji prima funkcija je *int32*, a ne *string* kao što očekujemo. Kao i mnoge druge funkcije u *class library*-ju, *WriteLine* je *overloaded* funkcija i ima različite verzije koje primaju stringove, integere, floating-point itd. U ovom slučaju se radi o verziji koja prima *int32* kao parametar – kao i u C++, automatski odabir odgovarajuće verzije vrši kompajler.

Posle pozivanja *WriteLine* funkcije, rutina ponovo učitava dve vrednosti u stack: lokalnu varijablu i konstantu 1. Ova operacija je praćena *add* instrukcijom koja sabira dve vrednosti iz ES-a i rezultat upisuje ponovo u ES. Dakle, kod dodaje vrednost 1 lokalnoj varijabli i rezultat smešta u stack (u liniji *IL_000d*). Ovo nas sada vraća na *IL_000e* gde smo se nalazili pre nego što smo ušli u ovaj loop.

Ovo je veoma jednostavna rutina. Sve što radi je loop-ing između *IL_0004* i *IL_0011* i štampa trenutnu vrednost brojača. Loop prestaje kada je vrednost brojača veća od 10. Nije nešto posebno, ali demonstrira kako IL radi.

Linked List Primer

Pre nego što pređemo na analizu obfuscated IL koda, pogledajmo na još jedan, malo složeniji, primer. Ovaj (kao i skoro svaki drugi .NET program na koji ćete naići) koristi par objekata tako da predstavlja malo realniji primer kako bi realan program izgledao. Počinjemo sa disasemblovanjem *Main* entrypoint-a programa, prikazanog na sledećem listingu:

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  .maxstack  2
  .locals init (class LinkedList V_0,
           int32 V_1,
           class StringItem V_2)
  IL_0000:  newobj     instance void LinkedList::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldc.i4.1
  IL_0007:  stloc.1
  IL_0008:  br.s       IL_002b

  IL_000a:  ldstr      "item"
  IL_000f:  ldloc.1
  IL_0010:  box        [mscorlib]System.Int32
  IL_0015:  call       string [mscorlib]System.String::Concat(
                                              object, object)
  IL_001a:  newobj     instance void StringItem::.ctor(string)
  IL_001f:  stloc.2
  IL_0020:  ldloc.0
  IL_0021:  ldloc.2
  IL_0022:  callvirt   instance void LinkedList::AddItem(class ListItem)
  IL_0027:  ldloc.1
  IL_0028:  ldc.i4.1
  IL_0029:  add
  IL_002a:  stloc.1
  IL_002b:  ldloc.1
  IL_002c:  ldc.i4.s   10
  IL_002e:  ble.s      IL_000a

  IL_0030:  ldloc.0
  IL_0031:  callvirt   instance void LinkedList::Dump()
  IL_0036:  ret
} // end of method App::Main
```

Listing br. 3
Malo složeniji IL kod koji koristi i objekte

Kao što ste i očekivali, i ova rutina počinje sa definisanjem lokalnih varijabli. Ovde su tri varijable, jedna tipa integera i dve tipa objekta (*LinkedList* i *StringItem*). Prva stvar koji program uradi je pravljenje instance *LinkedList* objekta i pozivanje njenog konstruktora sa *newobj* instrukcijom (obratite pažnju na to da je *.ctor* rezervisano ime metode za konstruktore). Posle toga učitava referencu ka novostvorenom objektu u prvu lokalnu varijablu, *V_0*, koja je naravno definisana kao *LinkedList* objekat. Ovo je odličan primer funkcionalnosti *managed* koda. Pošto smo eksplicitno definisali tip varijable i pošto je CLR uvek „svestan“ koji su data tipovi elemenata na stack-u, CLR može uvek da proveri da li je vrednost dodeljena varijabli kompatibilna vrednost. Ako postoji nekompatibilnost, CLR će okinuti exception.

Sledeća sekvenca koda na liniji *IL_0006* učitava 1 u *V_1* (koji je definisan kao *integer*) preko ES-a i nastavlja ka jump-u na *IL_002b*. Sada metoda učitava dve vrednosti u ES, 10 i vrednost varijable *V_1* i jump-uje nazad na *IL_000a*. Ovaj loop je veoma sličan loop-u iz predhodnog listinga, i predstavlja *posttested* loop. Očigledno, *V_1* je brojač i može da ima vrednost od 1-10. Kada bude imao vrednost veću od 10, loop se prekida.

Sekvenca na *IL_000a* je početak loop bloka. Tu metoda učitava string „*item*“ u ES, a potom i vrednost varijable *V_1*. Onda se vrednost varijable *V_1* „boxira“ , što znači da CLR kreira objekat koji ima kopiju *V_1* varijable i PUSH-uje referencu ka tom objektu u ES. Objekat ima tu prednost da nosi tačnu type informaciju, tako da metod koji će da primi taj objekat moći lako da utvrdi tip objekta. Ta „identifikacija“ se može lako izvršiti upotrebom IL instrukcije *isinst*.

Posle boxing-a varijable *V_1*, ostajemo sa dve vrednosti u ES-u, sa stringom „*item*“ i sa referencom ka boxed kopiji varijable *V_1*. Te dve vrednosti su tada prosleđene metodi iz *class library*-ja:

string [mscorlib]System.String::Concat(object, object)

koja uzima dva objekta i spaja ih u jednu string type vrednost. Ako su oba objekta stringovi, onda će metoda da ih samo „zalepi“ jedan za drugi. U suprotnom, metoda će da konvertuje oba objekta u string (pod pretpostavkom da ni jedan objekat nije string) i onda primeniti spajanje. U ovom slučaju se radi o *string* i *int32* objektima. Dakle, metoda će konvertovati *int32* u string i spojiti je sa prvim stringom. Rezultat je string, koji će se naći na vrhu stack-a, i koji bi trebao da ima formu „*itemX*“, gde je *X* vrednost varijable *V_1*.

Posle konstrukcije stringa, metod pravi *instance* objekta *StringItem* i poziva njegov konstruktor (ovo sve radi *newobj* instrukcija). Ako pogledamo prototip *StringItem* konstruktora, vidimo da prima jedan parametar tipa *string*. Pošto je rezultat *Concat* metode smešten na vrh ES-a, nema potrebe za zezanjem sa dodatnim kodom – string je već na ES-u i biće prosleđen konstruktoru. Po povratku iz konstruktora, *newobj* instrukcija će da stavi referencu ka novom objektu na vrh ES-a. Sledeća linija koda POP-uje referencu u *V_2* varijablu, koja je definisana kao *StringItem*.

Sledeća sekvenca učitava vrednosti varijabli *V_0* i *V_2* u ES i poziva:

LinkedList::AddItem(class ListItem)

Upotreba *callvirt* instrukcije pokazuje da je ovo virtuelni metod, što znači da će specifični metod biti određen *at runtime*, u zavisnosti od objekta nad kojim je inicijalizovan metod. Prvi parametar prosleđen funkciji je *V_2*, tj *StringItem* varijabla. To je instanca objekta za metod koji će biti pozvan. Drugi parametar je *V_0*, *ListItem* parametar koji metod uzima kao input. Prosleđivanje instance objekta kao prvi parametar kada pozivamo *class member* je standardna praksa u object-oriented jezicima. Ako se pitate o implementaciji *AddItem* member-a, to će mo raspraviti malo kasnije, ali prvi da završimo sa trenutnom metodom.

Sekvenvcu na *IL_0027* ste već videli. Povećava vrednost *V_1* varijable i smešta rezultat ponovi u *V_1*. Posle toga dolazimo do kraja loop-a. Kada se jump ne izvrši (kada je *V_1* veći od 10) kod poziva *LinkedList::Dump()* nad našim *LinkedList* objektom iz *V_0*.

Da zaključimo šta ste do sada videli, pre nego što pređem na analizu individualnih objekata i metoda. Videli smo program koji pravi instancu *LinkedList* objekta i loop-uje 10 puta kroz sekvencu koja pravi string *ItemX* gde je X vrednost brojača. Onda se string prosleđuje *LinkedList* objektu koristeći *AddItem* member. Kada se završi loop, poziva se *Dump* metod.

ListItem klasa

Sada možete da bacite pogled i na ostale objekte koji su definisani u programu i da analizirate njihove implementacije. Počnimo sa *ListItem* klasom:

```
.class private auto ansi beforefieldinit ListItem
       extends [mscorlib]System.Object
{
  .field public class ListItem Prev
  .field public class ListItem Next
  .method public hidebysig newslot virtual
          instance void  Dump() cil managed
  {
    .maxstack  0
    IL_0000:  ret
  } // end of method ListItem::Dump

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method ListItem::.ctor

} // end of class ListItem
```

Listing br. 4
Deklaracija ListItem klase

Nema šta puno da se vidi. Ima dva polja, *Prev* i *Next*, i oba su definisana kao *ListItem* reference. Ovo je klasična linked-list struktura. Pored dva polja, klasa i nema nešto puno koda. Imamo *Dump* virtuelni metod, koji je *prazan* i standardni konstruktor *.ctor* koji je kreiran od strane kompajlera.

LinkedList klasa

Sada prelazimo na analizu *LinkedList* klase koja je i ključ menadžmenta linked liste:

```
.class private auto ansi beforefieldinit LinkedList
       extends [mscorlib]System.Object
{
  .field private class ListItem ListHead
  .method public hidebysig instance void
          AddItem(class ListItem NewItem) cil managed
  {
    .maxstack  2
    IL_0000:  ldarg.1

    IL_0001:  ldarg.0
    IL_0002:  ldfld      class ListItem LinkedList::ListHead
    IL_0007:  stfld      class ListItem ListItem::Next
    IL_000c:  ldarg.0
    IL_000d:  ldfld      class ListItem LinkedList::ListHead
    IL_0012:  brfalse.s  IL_0020

    IL_0014:  ldarg.0
    IL_0015:  ldfld      class ListItem LinkedList::ListHead
    IL_001a:  ldarg.1
    IL_001b:  stfld      class ListItem ListItem::Prev
    IL_0020:  ldarg.0
    IL_0021:  ldarg.1
    IL_0022:  stfld      class ListItem LinkedList::ListHead
    IL_0027:  ret
  } // end of method LinkedList::AddItem

  .method public hidebysig instance void
          Dump() cil managed
  {
    .maxstack  1
    .locals init (class ListItem V_0)
    IL_0000:  ldarg.0
    IL_0001:  ldfld      class ListItem LinkedList::ListHead
    IL_0006:  stloc.0
    IL_0007:  br.s       IL_0016

    IL_0009:  ldloc.0
    IL_000a:  callvirt   instance void ListItem::Dump()
    IL_000f:  ldloc.0
    IL_0010:  ldfld      class ListItem ListItem::Next
    IL_0015:  stloc.0
    IL_0016:  ldloc.0
    IL_0017:  brtrue.s   IL_0009

    IL_0019:  ret
  } // end of method LinkedList::Dump

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method LinkedList::.ctor

} // end of class LinkedList
```

Listing br. 5
Deklaracija LinkedList klase

LinkedList objekat sadrži član *ListHead* tipa *ListItem*, i dve metode: *AddItem* i *Dump*. Počnimo sa *AddItem*. Ova metoda počinje sa interesantnom sekvencom koda koja PUSH-uje *NewItem* parametar na ES, praćen prvim parametrom koji je *this* referenca za *LinkedList* objekat. Sledeća linija koristi *ldfld* instrukciju za čitanje iz polja u *LinkedList* data strukturi (specifična instanca koja se čita je ona čija je referenca na vrhu stack-a – *this* objekat). Polje kome se pristupa je *ListHead*; sadržaj se nalazi na vrhu stack-a (kao i obično, *LinkedList* referenca se POP-uje po završetku instrukcije).

Nastavljamo na *IL_0007*, gde se poziva *stfld* koji upisuje u polje *ListItem* instance, čija je referenca druga po redu na stack-u (*NewItem* PUSH-ovan na *IL_0000*). Polje kojem se pristupa je *Next* i vrednost koja se upisuje je ona koja je trenutno na vrhu stack-a, tj vrednost koja je upravo pročitana iz *ListHead*-a. Nastavljamo na *IL_000c* gde je *ListHead* ponovo učitan u stack i proveren da li nosi pravu vrednost. Ovo je postignuto sa *brfalse* instrukcijom koja vrši grananje ako je vrednost na vrhu stack-a *null* ili *false*.

Pod pretpostavkom da do grananja nije došlo, kod nastavlja dalje ka *IL_000c* gde se *stfld* ponovo koristi, ali ovog puta da inicijalizuje *Prev* polje sa vrednošću *NewItem* parametra. Ideja je jasna, PUSH-uje se item koji je trenutno na početku liste i postaviti *NewItem* na početak liste. Ovo su klasične linked list sekvence. Finalna operacija ove metode je inicijalizovanje *ListHead* polja sa vrednošću *NewItem* parametra. Ovo se radi na *IL_0020*, adresi na koji *brfalse* jump-uje ako je *ListHead* jednako *null*. Ponovo, klasična linked list-adding sekvenca. Novi itemi se smeštaju na početak liste.

Sledeći metod koji gledamo je *Dump*, koji se nalazi odmah ispod *AddItem* metode. Metoda počinje učitavanjem trenutne vrednosti *ListHead*-a u *V_0* lokalnu varijablu koja je definisana kao *ListItem*. Onda sledi bezuslovni *branch* ka *IL_0016* (ovo ste videli više puta, uglavnom predstavlja početak *posttested* loop konstruktora). Kod na *IL_0016* koristi *brtrue* instrukciju koja proverava da li je *V_0* non-null i jump-uje na početak loop-a dok god je to slučaj.

Loop blok je veoma jednostavan. Poziva *Dump* virtuelnu metodu za svaki *ListItem* i učitava *Next* polje iz trenutnog *V_0* nazad u *V_0*. Možete da pretpostavite da je original kod sličan ovome:

CurrentItem = CurrentItem.Next

U principu, ovde se prelazi preko svakog člana liste i vrši „dampovanje“ svakog člana posebno. Trenutno ne znate šta znači „dampovanje“, pošto je *Dump* metoda u *ListItem*-u deklarisana kao virtualna metoda, šta će metoda uraditi zavisi od tipa objekta koji joj je prosleđen.

StringItem klasa

Zaključimo ovaj primer sa kratkim pregledom listinga br. 6. tj *StringItem* klase nastale iz *ListItem* klase:

```
.class private auto ansi beforefieldinit StringItem
       extends ListItem
{
  .field private string ItemData
  .method public hidebysig specialname rtspecialname
          instance void  .ctor(string InitializeString) cil managed
  {
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void ListItem::.ctor()
    IL_0006:  ldarg.0
    IL_0007:  ldarg.1
    IL_0008:  stfld      string StringItem::ItemData
    IL_000d:  ret
  } // end of method StringItem::.ctor

  .method public hidebysig virtual instance void
          Dump() cil managed
  {
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  ldfld      string StringItem::ItemData
    IL_0006:  call       void [mscorlib]System.Console::Write(string)
    IL_000b:  ret
  } // end of method StringItem::Dump

} // end of class StringItem
```

Listing br. 6
StringItem klasa

StringItem klasa je ekstenzija *ListItem* klase i sadrži jedno polje: *ItemData* definisano kao *string*. Konstruktor za ovu klasu prima jedan *string* parametar i smešta ga u *ItemData* polje. *Dump* metod samo prikazuje sadržaj *ItemData* polja pozivom na *System.Console::Write*. Teoretski, možete imati više klasa iz *ListItem* klase, svaku sa svojom *Dump* metodom namenjenu za „dampovanje“ vrednosti određenog tipa.

Dekompajleri

Kao što ste upravo videli, reversing IL koda je mnogo lakši od nativnog IA-32 ASM koda. IL kod ima daleko manji broj redundantnih detalja kao što su flagovi i registri, i mnogo više definicija klasa, lokalnih varijabli i tačnih data type informacija. Ovo znači da bi bilo neizmerno lakše dekompajlirati IL kod u high-level kod. U principu, retko ima potrebe da sedimo i čitamo IL kod kao što smo to maločas činili, osim ako je kod obfuskovan i ako dekompajler ne može da pravilno prevede kod.

Pokušajmo da dekompajliramo IL metodu i da vidimo kakav ćemo output dobiti. Sećate se *AddItem* metode sa listinga br. 5 ? Dekompajliraćemo je sa Spices.Net (9Rays.Net, www.9rays.net) i dobijamo ovaj kod:

```
public virtual void AddItem(ListItem NewItem)
{
  NewItem.Next = ListHead;
  if (ListHead != null)
  {
    ListHead.Prev = NewItem;
  }
  ListHead = NewItem;
}
```

Listing br. 7
Dekompajlirani AddItem metod

Ovaj listing je daleko čitljiviji od IL koda sa listinga br. 5. Objekti i njihova polja su adekvatno rešena, i uslovni izraz je predstavljen kako treba. Naravno, reference ka *this* objektu su eliminisane – nisu potrebne za adekvatno dekompajliranje ove rutine. U većini slučajeva ne znate u kojem je jeziku pisan program. Većina dekompajlera, kao Spices.Net, omogućavaju da dekompajlirate u kojem jeziku želite – nema veze u kojem je jeziku pisan program.

Visok kvalitet dekompajliranog neobfuskovanog koda znači da se reversovanje takvih .NET programa svodi na čitanje dekompajliranog source koda i pokušaja da se shvati šta program pokušava da uradi. Ovaj proces se uglavnom naziva *program comprehension* i može da varira od prostog do veoma kompleksnog, u zavisnosti od veličine programa i količine informacija koje su ekstraktovane iz njega.

Obfuskatori

Pošto .NET programi imaju tu „urođenu ranjivost“, koncept obfuskacije takvih programa u cilju onemogućavanja dekompilacije je veoma česta pojava danas. Ovo se veoma razlikuje od nativnih IA-32 kompajliranih programa, jer arhitektura IA-32 procesora (ma koliko to glupo zvučalo) nudi određenu zaštitu jer je nekome teško da čita ASM kod. IL kod je veoma detaljan kod i lako se dekompajlira u high-level source kod. Pre nego što pređemo na analizu pojedinih obfuskatora, pređimo preko osnovnih strategija za obfuskaciju .NET programa.

Symbol Renaming

Pošto .NET programi sadrže čiljtiva imena parametara metoda, imena klasa, imena polja i imena metoda, ti simboli moraju biti izbrisani iz programa ako želimo da otežamo reversing. Pravo uklanjanje tih stringova nije moguće jer su neophodni za identifikaciju elemenata unutar programa. Umesto toga, ovim simbolima se daju beznačajna, kriptovana imena umesto originalnih. Na primer, *ListItem* može da postane *d* ili *xcf1h59m5dfo6j*. Ovo ne može nikoga da zaustavi u reversingu, ali ga može otežati.

Control-Flow Obfuskacija

Kod .NET programa, *control-flow* obfuskacija se vrši da bi se sprečio dekompajler i da bi se onemogućilo stvaranje koda koji bi mogao da ima značaj za reversera. Ovo može da se uradi veoma lako, jer dekompajleri očekuju osetljive CFG-ove (o CFG-ovima pročitajte u *Decompiling It!)* koji se mogu lako prevesti u high-level *control-flow* konstruktore kao što su loop-ovi i uslvni izrazi.

Sprečavanje Dekompajliranja & Disasemblovanja

Jedna od mogućnosti kojukoristi veliki broj obfuskatora, uključujući Dotfuscator, XenoCode i Spices.Net, je ta da se pokuša kompletno sprečiti dekompajliranje ili disasemblovanje. U zavisnosti koji program pokušava da otvori obfuskovan .exe, može doći do rušenja programa ili do prikazivanja error poruke, kao što to prikaže ILdasm:

Postoji nekoliko različitih strategija za zaštitu od disasemblovanja i dekompajliranja .NET asemblija. Ako mislimo na ILdasm, postoje nedokumentovani metadata unosi koje proverava ILdasm prilikom učitavanja programa. Ti unosi su modifikovani od strane obfuskatora što na kraju proizvodi gornju poruku.

Drugi način bi bio oštećivanje metadata-e tako da CLR i dalje može da pokrene program, ali da zbuni programe koji učitavaju obfuskovani program u memoriju i skeniraju njegov metadata. To se može učiniti ubacivanjem lažnih referenci ka nepostojećim stringovima, poljima ili metodama. Neki programi ne handluju takve *„broken“* linkove i jednostavno se ruše prilikom učitavanja. Ovo je loš pristup obfuskaciji i generalno se ne preporučuje, pogotovo ako uzmemo o ubzir koliko je lako autorima dekompajlera da zaobiđu ovakve trikove.

Reversing Obfuskovanog Koda

Ova sekcija predstavlja nekoliko efekata koje proizvode popularni .NET obfuskatori, i ocenu koliki učinak imaju. Za one koji traže tačnu meru u kojoj obfuskacija otežava proces reversinga, ovde nema mesta, jer takva mera ne postoji. Sada čemo da analiziramo uticaj nekih popularnih obfuskatora na linked-list primer i pokušaćemo da utvrdimo koliko su efektivni ti obfuskatori protiv dekompajliranja i „ručne“ analize IL koda.

XenoCode Obfuscator

Kao prvi test primer sam uzeo linked-list primer koji smo ranije analizirali i provukao sam ga kroz XenoCode 2005 (XenoCode Corporation, www.xenocode.com) obfuscator, sa uključenim *string renaming* i *control flow* opcijama. Opcija „Suppress Microsot IL Disassembler“ je takođe uključena i sprečila je Ildasm da disassembluje kod, ali je i dalje bilo moguće disassemblovati sa nekim drugim alatom poput Decompiler.NET (Jungle Creatures Inc., www.junglecreature.com) ili Spices.Net. Takođe, oba alata podržavaju i disassemblovanje i dekompilaciju. Listing br. 8 prikazuje disasemblovani kod *AddItem* funkcije koji je napravio Spices.Net:

```
instance void x5921718e79c67372(class xcc70d25cd5aa3d56
                                xc1f1238cfa10db08) cil managed
{
  // Code size: 46 bytes
  .maxstack 8
  IL_0000: ldarg.1
  IL_0001: ldarg.0
  IL_0002: ldfld       class xcc70d25cd5aa3d56
                         x5fc7cea805f4af85::xb19b6eb1af8dda00
  IL_0007: br.s        IL_0017
  IL_0009: ldarg.0
  IL_000a: ldfld       class xcc70d25cd5aa3d56
                         x5fc7cea805f4af85::xb19b6eb1af8dda00
  IL_000f: ldarg.1
  IL_0010: stfld       class xcc70d25cd5aa3d56
                         xcc70d25cd5aa3d56::xd3669c4cce512327
  IL_0015: br.s        IL_0026
  IL_0017: stfld       class xcc70d25cd5aa3d56
                         xcc70d25cd5aa3d56::xbc13914359462815
  IL_001c: ldarg.0
  IL_001d: ldfld       class xcc70d25cd5aa3d56
                         x5fc7cea805f4af85::xb19b6eb1af8dda00
  IL_0022: brfalse.s   IL_0026
  IL_0024: br.s        IL_0009
  IL_0026: ldarg.0
  IL_0027: ldarg.1
  IL_0028: stfld       class xcc70d25cd5aa3d56
                         x5fc7cea805f4af85::xb19b6eb1af8dda00
  IL_002d: ret
}//end of method x5fc7cea805f4af85::x5921718e79c67372
```

Listing br. 8
Disasemblovani obfuskovani kod AddItem funkcije
(Spices.Net)

Prva stvar koju primetimo na ovom listingu je da su simboli preimenovani. Umesto gomile „lepih“ imena klasa, metoda i polja, sada vidimo dugačke, *random* kombinacije brojeva i slova. Ovo je veoma iritirajuće, i možda bi imalo smisla da reverser preimenuje ove dugačke simbole u kraće, kao na primer *a, b,* itd. I dalje neće imati nekakav značaj, ali će se lakše uočiti veze sa ostalim simbolima.

Pored kriptovanih imena simbola, control flow izrazi u metodi su obfuskovani takođe. Ovo znači da su segmenti koda ispomerani tamo-amo upotrebom bezuslovnih jump-ova. Na primer, *unconditional branch* na *IL_0007* je jednostavno originalni *if* izraz, samo što je relociran na „kasnije“ mesto u funkciji. Kod koji ide posle te instrukcije (do kojeg se dolazi sa *IL_0024*) je ustvari blok tog *if* izraza. Problem kod ovakvih transformacija je taj da ne predstavljaju nekakvu zaštitu protiv reversera koji radi na IL nivou. U stvari su mnogo efektivnije protiv dekompajlera koji se time mogu lako zbuniti i navesti da ih konvertuju u *goto* izraze. Ovo se dešava kada kompajler ne uspe da napravi ispravan CFG metode. Za više informacija o dekompajliranju pročitatje moj *Decompiling It!* tekst.

Da vidimo šta se dešava kada ubacimo gornji obfuskovani kod u Spices.Net dekompajler. Sledeći listing predstavlja dekompajlirani gornji kod obfuskovane C# metode:

```
public virtual void x5921718e79c67372(xcc70d25cd5aa3d56
                                      xc1f1238cfa10db08)
{
  xc1f1238cfa10db08.xbc13914359462815 = xb19b6eb1af8dda00;
  if (xb19b6eb1af8dda00 != null)
  {
    xb19b6eb1af8dda00.xd3669c4cce512327 = xc1f1238cfa10db08;
  }
  xb19b6eb1af8dda00 = xc1f1238cfa10db08;
}
```

Listing br. 9
Dekompajlirani obfuskovani kod AddItem metode (Spices.Net)

Heh, izgleda da je Spices.Net kako treba rešio control-flow obfuskaciju. Naravno, preimenovani simboli i dalje smetaju pri analizi, ali je i dalje moguća. Jedina dobra stvar kod svega ovoga su dugačka random imena simbola koje je ubacio XenoCode. Taj metod obfuskacije je veoma effektivan jer iziskuje mnogo truda da se pronađu *cross-reference*. Nije baš lak posao prelaziti preko tih stringova i pronalaziti razlike.

DotFuscator by Preemptive Solutions

DotFuskator (Preemptive Solutions, www.preemptive.com) je još jedan obfuskator koji nudi sličnu funkcionalnost kao i XenoCode. Podržava *symbol renaming*, control flow obfuskaciju i može da spreči određene alate da dampuju i disasembluju obfuskovani .NET program. DotFuscator podržava agresivno menjanje imena simbola koje eliminiše *namespaces* i koristi *overloaded* metode kako bi dodao još više zabune (Overload-Induction opcija). Recimo da imamo klasu koja ima tri različite metode: jednu koj ane prima parametre, jednu koja prima *integer* i jednu koja prima *boolean*. Stvaro kod Overload-Induction-a je da će sve tri metode verovatno primiti isto ime a da će se specifičan metod izabrati po broju i tipu parametara koji su joj prosleđeni. Ovo je veoma zbunjujuće jer je teško razabrati razliku između dve metode. Listing br. 10 predstavlja *LinkedList::Dump* metod:

```
instance void a() cil managed
{
  // Code size: 36 bytes
  .maxstack 1
  .locals init(class d V_0)

  IL_0000: ldarg.0
  IL_0001: ldfld       class d b::a
  IL_0006: stloc.0
  IL_0007: br.s        IL_0009
  IL_0009: ldloc.0
  IL_000a: brtrue.s    IL_0011
  IL_000c: br          IL_0023
  IL_0011: ldloc.0
  IL_0012: callvirt    instance void d::a()

  IL_0017: ldloc.0
  IL_0018: ldfld       class d d::b
  IL_001d: stloc.0
  IL_001e: br          IL_0009
  IL_0023: ret
}//end of method b::a
```

Listing br. 10
Disasemblovan obfuskovani kod Dump metode (DotFuscator)

Prva uočljiva funkcija DotFuscator-a su ona kratka, single-letter imena za simbole. Ovo može srašno da iznervira jer svaka klasa ima bar jedan metod *a*. Ako pokušate da pratite control-flow sa listinga br. 10, videćete da uopšte ne podseća na *LinkedList::Dump* – DotFuscator može da primeni veoma agresivne mere control-flow obfuskacije, u zavisnosti od želje programera.

Prvo, uslov loop-a je pomeren na početak loop-a, i bezuslovni sa početka loop-a je prebačen na kraj (na *IL_001e)*. Ova struktura ne predstavlja ništa više od *pretested* loop-a, ali postoje i dodatni elementi koji su obačeni da bi se zbunio dekompajler. Ako pogledate na uslov loop-a, videćete da je preuređen na krajnje neobičan način: ako je *brtrue* instrukcija „zadovoljena“, preskače bezuslovni jump i jump-uje u loop blok. A ako nije, sledeća instrukcija je bezuslovni skok koji nas vodi na kraj metode.

Pre uslova loop-a se nalazi neobična sekvenca koda na *IL_0007* koja koristi *unconditional branch* instrukciju da preskoči sledeću instrukciju na *IL_0009. IL_0009* je prva instrukcija u loop-u i bezuslvni branch na kraju loop-a se vraća na tu instrukciju. Izgleda da je ideja sa bezuslovnim branch-om na *IL_0007* ta da se zakomplikuje CFG i da se stvore da bezuslovna branch-a koja vode na istu lokaciju, što na kraju služi tome da se zbune control-flow algoritmi u nekim dekompajlerima.
Sada ćemo da ubacimo ovaj obfuskovani kod u dekompajler i pokušaćemo da utvrdimo kakav uticaj ove agresivne control-flow obfuskacione tehnike imajuna finalni kod. Ovaj kod je proizveo Spices.Net za listing br. 10:

```
public virtual void a()
{
  d d = a;
  d.a();
  d = d.b;
  while (d == null)
  {
    return;
  }
}
```

Listing br. 11
Dekompajlirani kod obfuskovan DotFuscator-om
(Spices.Net)

Spices.Net je totalno zbunjen neobičnim control-flow konstruktorima ove rutine i generiše netačan kod. Ne uspeva da ispravno identifikuje loop blok i tačna mesta *return* izraza unutar loop-a, iako je izvršen *posle* loop-a, *d.a()* i *d = d.b* izrazi su postavljeni pre loop-a iako su sastavni deo loop bloka. Na kraju, uslov loop-a je suprotan: loop treba da se ponavlja ukoliko je *d = notnull* a ne obrnuto.Različiti dekompajleri koriste različite control-flow algoritme i generalno gledajući, različito reaguju na ovakve tipove control-flow obfuskacije. Hajde da ubacimo DotFuscated kod u neki drugi dekompajler, recimo Decompiler.Net i proverimo kako reaguje na DotFuscator-ovu obfuskaciju:

```
public void a ()
{
      for (d theD = this.a; (theD != null); theD = theD.b)
      {
            theD.a ();
      }
}
```

Listing br. 12
Dekompajlirani kod obfuskovan DotFuscator-om
(Decompiler.Net)

Ovde nema problema – Decompiler.Net je korektno odradio posao i obfuskovana control-flow struktura izgleda nema uticaj na finalni kod. Činjenica je da control-flow obfuskacija imaju određenu prirodu „mačke i miša“ – autori dekompajlera uvek imaju mogućnost dodavanja specijalnih heuristika koje se mogu baviti sa različitim control-flow obfuskacionim strukturama. Važno je to uvek imati u vidu i ne podceniti uticaj koji ove tehnike imaju na sveukupnu čitljivost programa. Skoro je uvek moguće dekompajlirati obfuskovani control-flow kod – na kraju kod mora da zadrži originalno značenje da bi program radio kako treba.

Ako se vratimo na temu preimenovanja simbola, primećujete kako je zbunjujuća ova jednostavna *alphabetical naming* šema? Metod *a* pripada klasi *b* i postoje dve refence ka *a:* jedna *this.a* referenca i *theD.a* metod *call*. Jedna je polje u klasi *b* a druga je metod u klasi *d*. Ovo je odličan primer kako preimenovanje simbola može da iznervira reversera.
E, da, već dok se bavimo preimenovanjem simbola, DotFuscator ima još jednu opciju koja može da dovede do dodatnih problema za reversera. Moguće je zameniti imena simbola sa invalid karakterima koji se ne mogu prikazati pravilno. Ovo znači da u zavisnosti koji se alat koristi za pregled koda, ponekad nemoguće razlikovati jedan simbol od drugog i u nekim slučajevima ovi karakteri mogu da spreče određeni alat u otvaranju asemblija. Sledeći listing je *AddItem* metod obfuskovan DotFuscator-ovom opcijom *„Unprintable Symbol Names“*. Sledeći kod je proizveo Decompiler.Net:

```
public void áœ¤ (áœf A_0)
{
     A_0.áœ_ = this.áœ¤;
     if (this.áœ¤ != null)
     {
          this.áœ¤.áœ¤ = A_0;
     }
     this.áœ¤ = A_0;
}
```

Listing br. 13
Dekompajlirani kod obfuskovan DotFuscator-ovom
Unprintable Symbol Names opcijom
(Decompiler.Net)

Kao što vidite, ovu funkciju je skoro nemoguće odgonetnuti – veoma je teško uočiti razliku između simbola. Ipak, dekompajleru ne bi trebalo biti teško da prevaziđe ovaj problem – trebalo bi samo da identifikuje takve simbole i da ih preimenuje u čitljiviju formu. Sledeći listing predstavlja isti kod dekompajliran sa Spices.Net (koji gornji problem rešava automatski):

```
public virtual void \u1700(\u1703 A_0)
{
  A_0.\u1701 = \u1700;
  if (\u1700 != null)
  {
    \u1700.\u1700 = A_0;
  }
  \u1700 = A_0;
}
```

Listing br. 14
Dekompajlirani kod obfuskovan DotFuscator-ovom
Unprintable Symbol Names opcijom
(Spices.Net)

Pošto je Spices.Net automatski preimenovao simbole, ovaj metod postaje puno čitljiviji. Ovo je tačno i za ostale, manje agresivne tehnike preimenovanja. Dekompajler može uvek u toku dekompajliranja da promeni ime svakom simbolu i učini kod dosta čitljivijim. Na primer, učestala upotreba *a*, *b* i *c* (spominjani ranije) bi mogla biti zamenjena sa unikatnim imenima. Zaključak je taj da mnoge od transformacija koje uradi obfuskator mogu biti delimično uklonjene sa odgovarajućim automatizovanim alatima. Sve dok je god moguće delimično ili u potpunosti ukloniti efekat transformacija, obfuskatori postaju beskorisni. Izazov za developere obfuskatora je da iskodiraju nereverzibilnu transformaciju.

Remotesoft Obfuscator i Linker

Remotesoft Obfuscator (Remotesoft, www.remotesoft.com) je proizvod baziran na konceptima sličnim onima kod drugih obfuskatora koje sam spominjao. Jedina razlika je ta da ovaj obfuskator ima i Linker komponentu koja dodaje još jedan „sloj“ zaštite. Ova mogućnost obfuskatora je veoma korisna opcija u mnogim sličajevima, ali je isto tako zanimljiva sa reversing strane, jer obezbeđuje dodatni sloj (layer) zaštite protiv reversinga.

Kao što sam demonstrirao u *Decompiling It!* tekstu, u situacijama kada je vaoma malo informacija o delu koda dostupno, sistemski pozivi mogu mnogo da pomognu. Problem kod .NET programa je taj da bez obzira koliko dobro je program obfuskovan, on će i dalje imati dobro definisane i dostupne pozive ka *System namespace*-u, koji nam opet mogu mnogo pomoći pri analizi koda. Rešenje je obfuskovanje *class library*-ja i distribucija iste uz program. Na ovaj način, kada se referencira objekat *System*, naziv poziva je „mangled“ tj. zbrkan, tako da postaje veoma teško da se utvrdi koji je stvarni poziv korišćen.

Jedan pristup koji može da odgonetne takve sistemske klase, čak i ako su preimenovane, koristi hijerarhijski *call graph view* koji prikazuje kako različite metode vrše interakciju. Pošto *System* klasa sadrži veliku količinu koda izolovanu od glavnog programa (koji nikada ne poziva kod u glavnom programu), postaje veoma lako da se identifikuju sistemske grane i da se bar zna da je određena klasa deo *System* namespace-a. Postoji nekoliko alata koji mogu da naprave *call* grafikone za .NET asemblije, uključujući i nama dobro poznati IDA Pro.

Remotesoft Protector

Remotesoft Protector je još jedan obfuskator koji koristi drugačiji pristup zaštiti od reversovanja. Protektor ima dva moda u kojima radi. Postoji *platform-dependent* mod u kojem se ceo IL kod prekompajlira u IA-32 kod i time kompletno eliminiše IL kod iz programa. Ovo ima veliku prednost jer već znamo da je reversing običnog IA-32 ASM koda daleko teži od IL koda. Loša strana ovog moda je da program može pokrenuti samo na IA-32 platformi.

Protektor takođe poseduje i *platform-independent* mod koji umesto eliminacije vrši enkripciju koda. U ovom modu protektor enkriptuje IL instrukcije i „skriva“ ih unutar programa. Ovo je veoma slično nekim pakerima i DRM (Digital Rights Management) proizvodima za nativne IA-32 programe. Rezultat takve transformacije koda je nemogućnost direktnog učitavanja ovakvih asemblija u dekompajler ili disasembler, jer je IL kod nedostupan i enkriptovan unutar asemblija.

U sledeće dve sekcije opisaću ove dve tehnike i pokušaću da odredim nivo zaštite koje pružaju.

Prekompajlirane Asemblije

Ako želite da žrtvujete portabilnost aplikacije, prekompajliranje je neosporivo najbolji način da zaštitite vašu aplikaciju. Nativni kod je drastično manje čitljiv od IL koda i koliko znam nema ni jedan *working* dekompajler (osim DeDe-a, ali on je za Delphi). Čak i ako ih ima, sumnjam da proizvode source kod imalo sličan onom koje proizvodi IL dekompajler.

Pre nego što neko odmah pohita da prekompajlira svoj program, imajte ovo u vidu. Prekompajlirane asemblije zadržavaju metadata-u – metadata je neophodan da bi CLR mogao da pokrene program. Ovo znači da bi teoretski bilo moguće za nekakav custom native dekompajler da pročita metadatu i da nekako poveća kvalitet izlaznog koda. Ako se pojavi takav dekompajler, biće u mogućnosti da proizvede veoma čitljiv kod.

Ostavimo ovaj „advanced decompiler“ po strani i sagledajmo drugu stranu medalje. Čak i ako ste prekompajlirali .NET program i IA-32 nativni program, imajte na umu da reversing IA-32 programa nije *BAŠ* toliko težak posao :D Sve što je potrebno je vreme i odlučnost! To mi koji se bavimo tim poslom najbolje znamo. Zaključak je, ako imate veliku količinu koda koju želite da zaštitie, prekompajling će odraditi posao. Ako imate samo nekakv mali algoritam koji želite da zaštitite, čak ni prekompilacija neće odvratiti odlučnog reversera da je nađe.

Enkriptovane Asemblije

Za one koji ne žele da žrtvuju portabilnost nad sigurnošću, Protector nudi još jednu opciju koja zadržava platformsku nezavisnost .NET-a. Ovaj mod enkriptuje IL kod i smešta ga unutar asemblije. Da bi zaštićene asemblije radile u platformski nezavisnoj formi, Protector uključuje i DLL koji dekriptuje IL kod i nalaže JIT-u da kompajlira dekriptovane metode *at runtime*. Ovo znači da enkriptovani programi nisu 100% platformski nezavisni – i dalje vam treba dekripcioni DLL za svaku platformu.

Ovaj pristup ekripcije IL koda je efektivan protiv standardne dekompilacije, ali i ništa više od toga. Ključ za enkripciju IL koda se pravi hash-ovanjem određenih sekcija asemblija sa MD5 algoritmom. Kod se onda enkriptuje sa RC4 algoritmom sa MD5 ključem.

Ove imamo stari problem. Enkripcioni algoritmi, bez obzira koliko moćni bili, nemaju veliki značaj kada se ključ prosledi i legalnom kupcu i reverseru. Pošto dekripcioni ključ mora biti u asembliju, sve što reverser treba da uradi da dekriptuje IL kod je da pronađe ključ.

Dok se na prvi pogled čini da obfuskator pruža manju zaštitu od zaštite baziranoj na enkripciji, to stvarno i nije slučaj. Mnoge obfuskacione transformacije su ireverzibilne, tako da je obfuskovani kod nemoguće odgonetnuti, a reverser nikada neće biti u mogućnosti da povrati asembliju u originalno stanje.

Da bi reversovali asembliju zaštićenu sa Protector-om, morali bi nekako da dekriptujemo IL kod unutar asemblije i da ga potom dekompajliramo sa nekim standardnim dekompajlerom. Na nesreću, ovaj proces je veoma jednostavan ako imamo u vidu da je encryption / decryption ključ u samom asembliju. Ovo je tipično ograničenje svake *code encryption* tehnike: *Dekripcioni ključ mora biti dostupan krajnjem korisniku da bi mogli da pokrenu program, i može se koristiti za dekripciju kriptovanog koda.*

U malom eksperimentu koji sam proveo na test asembliju obfuskovan sa Remotesoft Obfuscator-om i enkriptovan sa Remotesoft Protector-om (u Version-Independent modu), bio sam u mogućnosti da veoma lako pronađem dekripcionu rutinu u DLL-u i da pronađem tačnu lokaciju enkripcionog ključa u asembliju. Stepovanjem kroz dekripcioni kod, bio sam u mogućnosti da pronađem lokaciju i raspored enkriptovanog koda. Pošto sam saznao sve potrebne informacije, mogao sam da napišem unpacker koji bi dekriptovao i dampovao enkriptovani kod. Ne bi bilo previše teško ni ubaciti dekriptovani kod u neki dekompajler i time dobiti razumljiv kod.

Ovo je razlog zašto uvek prvo treba propustiti asembliju kroz obfuskator pa tek onda kroz protektor. Čak i ako reverser uspe da dekriptuje kod, želite da se uverite da je kod dobro obfuskovan. U suprotnom biće previše lako dobiti source kod vašeg programa.

Zaključak

Sigurno je to da je .NET kod dosta podložniji reversingu od IA-32 koda ili bilo kog drugog nativnog koda za druge arhitekture. Kombinacija IL koda i metadata-e omogućava dekompajliranje IL metoda u neverovatno čitljiv source kod. Obfuskatori pokušavaju da otežaju ovaj proces što je više moguće, upotrebom brojnih tehnika, ali imaju ograničen efekat i samo mogu da uspore odlučnog reversera.

Postoje dve potencijalne strategije za kreiranje moćnijih obfuskatora koje bi mogle da imaju veliki uticaj u sigurnosti .NET programa. Jedna je ta da se poboljša koncept enkripcije koju koristi Remotesoft Protector i da se koriste različiti ključevi za različite delove programa. Dekripcija bi se trebala odvijati sa polimorfnim IL kodom koji bi se razlikovao od programa do programa (da bi se sprečilo kreiranje generičkih unpakera), i da se koriste ključevi koji proizilaze od raznih mesta (regiona metadata-e, konstanti unutar koda, parametara koji se prosleđuju metodama…).

Još jedan pristup je ulaganje u još napredniju obfuskacionu transformaciju. To su transformacije koje značajno menjaju strukturu koda što rezultuje težim razumevanjem koda. Takve transformacije možda neće biti dovoljne da se spreči dekompajliranje, ali je cilj taj da se što više smanji čitljivost izlaznog koda, do onog stepena kada je neupotrebljiv za reversera. Verzija 3.0 DotFuscator-a bi trebala da krene tim putem, i očekujem da će tako da postupe i ostali developeri obfuskatora.

Pa Čoveče, Umukni Već Jednom!

Da, da ,da… Očekivao sam takav komentar na kraju (ko je pročitao sve ovo il' je lud il' blesav).

E pa dragi moji, stvarno sam se više smorio pišući ovaj tekst. Šta bre hoćete? 28 strana evo sa ovom zadnjom. Nadam se da ste uživali čitajući ovaj poveći tekst. Nešto ste naučili, malo ste se smarali (kao i uvek kada sam ja u pitanju).

I kako to već ide sada malo greetz i thanks i fuckoffs itd...

Greetz to (bez nekog posebnog redosleda):

deroko, ap0x, Vrane, Kaca, Orthodox, Reverser, i svima ostalima na RLabs forumu...

Thanks to (bez nekog posebnog redosleda):

deroko [ARTeam], ap0x [RLabs], Gabri3l [ARTeam], Shub-Nigurath [ARTeam], spothorse, LOQUILLO, SenseiX, sort, Dcoder, McCool [MP2k], diablo2oo2 [MP2k], dila, Whiterat [ICU], PolishOX [ICU], gammexane [ICU], Knetus [ICU], Baxxter [TSRh], ByTESCRK [TSRh], SuperCracker [SnD], Ziggy [SnD], melatonin [ICU], aer0smith, loco, moom, KaGra, _pusher_, Tanatos, ados, BlaCkEye, CoaXCable [CPH], Crudd [RET], DappA [ICU], irokos [TiTAN], Whoopee [TiTAN], glAsh [CRD], Int3rpol [TiTAN], tHeBiGmAn, NeoXQuick [XMA0D], p0int [CRD], QuEST [CPH], AliEN [ICU], Eg0iste [TSRh], SvenZZon [TiTAN]…

FuckOff's:

Znate ko ste, ne bih da se ponavljam...!

Kontakt:

Web: http://ap0x.jezgra.org/forum/
Email: lafaman@gmail.com (I ne pokušavajte da me spamujete)
IRC: #SCForce, #ARTeam, #Cracking4Newbies @ EFnet

Са Вером У Бога
LaFarge / RLTeam / Team ICU

Četvrtak, Jun 01, 2006