Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Sledovanie volaní v .NET aplikáciach

Bakalárska práca

2016
Martin Ivančík

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Sledovanie volaní v .NET aplikáciach

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Richard Ostertág, PhD.
Konzultant:	Mgr. Peter Košinár

Bratislava, 2016
Martin Ivančík

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Martin Ivančík
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Sledovanie volaní v .NET aplikáciach
.NET Call Tracer

Cieľ: Cieľom práce je ucelene zdokumentovať postupy a časti .NET platformy potrebné pre implementáciu nástroja, pomocou ktorého bude možné zaznamenať sledované činnosti vykonané počas behu zvolenej .NET aplikácie. Medzi sledované činnosti budú patriť najmä:
- volania metód,
- ich argumenty,
- návratové hodnoty.

Súčasťou práce bude aj implementácia jednoduchej aplikácie využívajúcej zdokumentované postupy.

Vedúci: RNDr. Richard Ostertág, PhD.
Konzultant: Mgr. Peter Košinár
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 26.10.2015

Dátum schválenia: 27.10.2015 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Chcel by som sa poďakovať môjmu školiteľovi RNDr. Richardovi Ostertágovi, PhD. a konzultantovi Mgr. Petrovi Košinárovi za rady a pomoc pri práci. Moje poďakovanie patrí aj Jacquelinovi Potierovi za poskytnuté materiály a odpovede na všetky moje otázky, ktoré vznikli pri implementácii profilovacieho nástroja.

Abstrakt

V tejto práci stručne predstavíme platformu .NET. Popíšeme štruktúru metadát v spustiteľnom súbore .NET aplikácie a vysvetlíme, akým spôsobom možno sledovať volania funkcií a získať hodnoty ich argumentov a návratových hodnôt.

Kľúčové slová: metadáta, signatúry, profilovanie, COM objekt, hook, common language runtime

Abstract

In this thesis, we introduce .NET platform and describe structure of metadata in .NET executable file. We then explain how to trace called functions and retrieve values of its arguments and return values.

Keywords: metadata, signatures, profiling, COM object, hook, common language runtime

Obsah

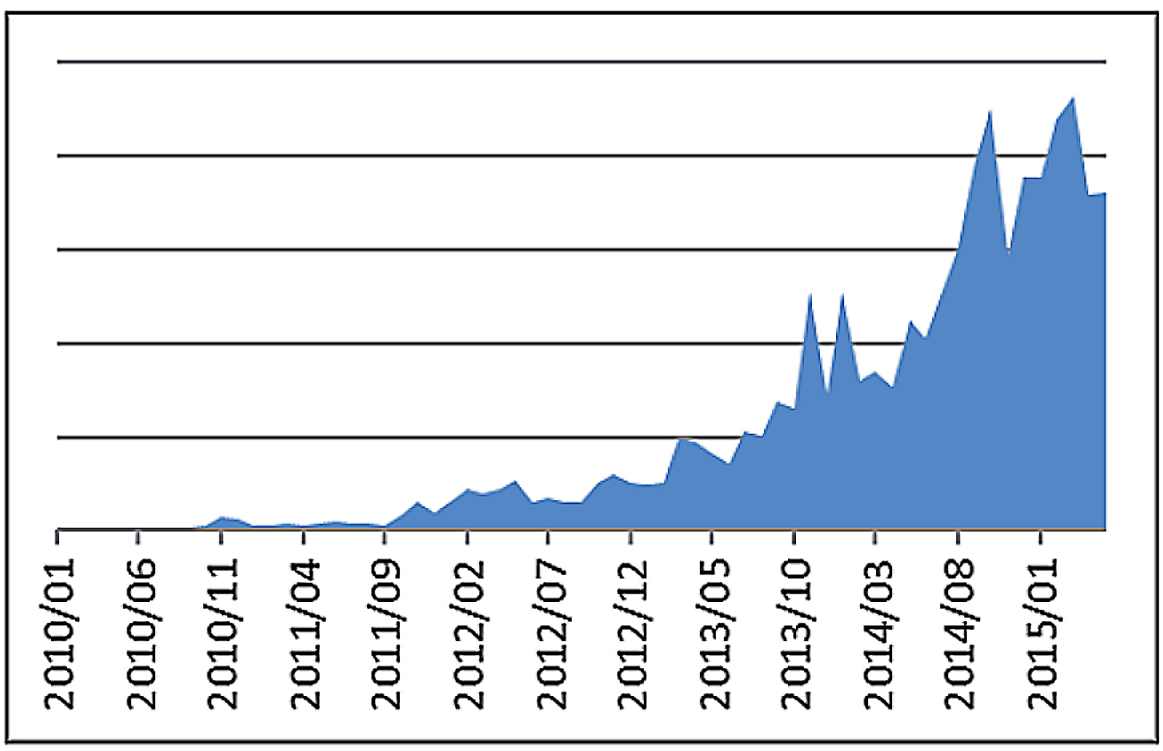
Zoznam obrázkov

Úvod

S rozvojom informačných technológií za posledných desať rokov dochádza k rozšíreniu škodlivého softvéru, určeného pre najrozšírenejšie platformy ako napríklad Windows. Škodlivý softvér (malvér) predstavuje aplikáciu, ktorej účelom je spôsobiť škodu v systéme, prípadne odcudziť dáta. Autori malvéru motivovaní peniazmi a ziskom citlivých údajov vytvárajú rozličné techniky, ako obísť ochranné mechanizmy a spôsobiť čo najväčšiu ujmu. Škodlivý kód už nie je jednoduchý ako v minulosti. Aktívne sa bráni proti odhaleniu, či odstráneniu a vzhľadom na postavenie informačno-komunikačných technológií v dnešnom svete, môže spôsobiť fatálne škody, dokonca straty na životoch. Vzniká teda potreba vedieť sa brániť pred takýmto druhom softvéru. Aby to bolo možné, je potrebné vedieť, ako malvér funguje.

Analýza škodlivého kódu je spôsob, ktorý umožňuje poznať štruktúru a funkcionalitu malvéru, následkom čoho je možné vytvoriť technológie umožňujúce detekciu a ochranu pred škodlivými aplikáciami. Autori škodlivého kódu si toto uvedomujú, a preto sa snažia o znemožnenie analýzy, respektíve o predĺženie času potrebného na analyzovanie kódu použitím rôznych obfuskačných techník ako zneprehľadnenie kódu, vkladanie nezmyselných inštrukcií, šifrovanie častí kódu, či pridávanie vetiev, do ktorých sa vykonávaný program nedostane.

Jedným zo spozorovaných javov počas uplynulých piatich rokov je nárast malvéru pre platformu .NET. Situáciu znázorňuje obrázok 1. Pre tvorcov malvéru je použitie vysoko úrovňových programovacích jazykov ako Visual Basic, či C# pravdepodobne jednoduchšie a pohodlnejšie. Analyzovať programový kód .NET aplikácie po aplikovaní techník pre sťaženie analýzy je veľmi náročné. Analytici škodlivého kódu používajú rozličné nástroje pri skúmaní softvéru. Platforma .NET je však určitým spôsobom špecifická. V tejto práci predstavíme možnosti, ako vhodne sledovať vybrané volania v bežiacej .NET aplikácii a ako získať argumenty a návratové hodnoty vyvolaných funkcií za účelom zjednodušenia analýzy správania programu.



Obr. 1: Nárast výskytu .NET malvéru [5].

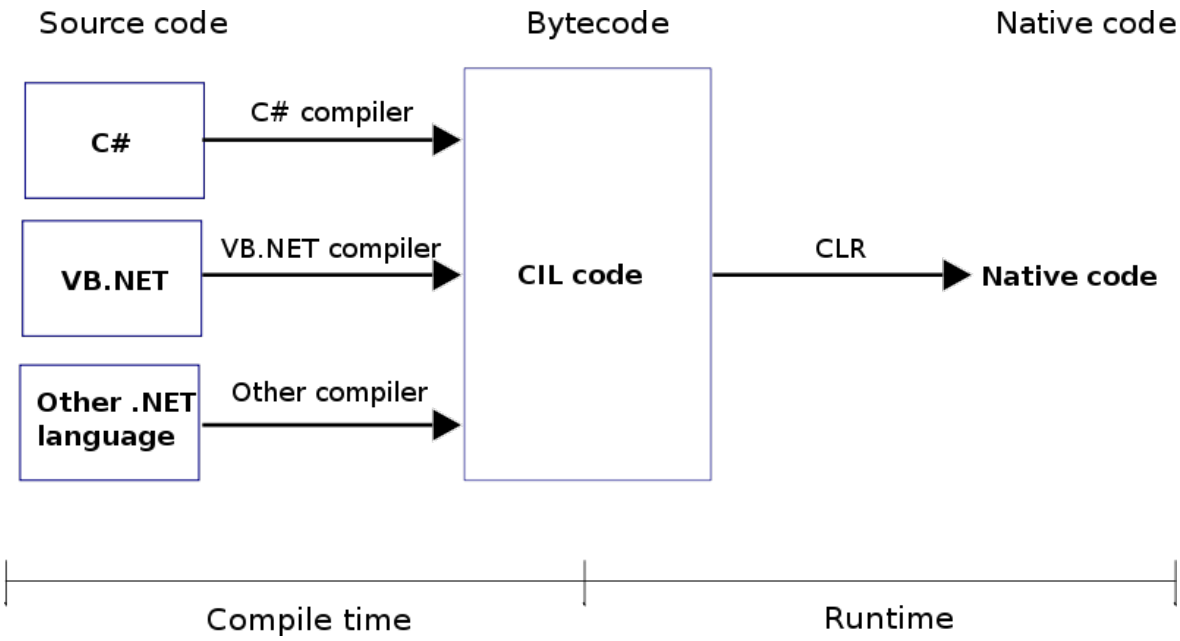
Kapitola 1

Platforma .NET

V tejto kapitole stručne predstavíme platformu .NET. Okrem základných pojmov popíšeme spôsob vykonávania .NET aplikácií za účelom pochopenia princípu sledovania vybraných volaní.

1.1 .NET

Microsoft .NET Framework je softvérová platforma od spoločnosti Microsoft. Pozostáva z dvoch častí: FCL a CLR. FCL (Framework Class Library) je skupina knižníc využívaných .NET aplikáciami. Vykonávanie .NET aplikácií prebehia prostredníctvom softvérového prostredia CLR (Common Language Runtime). Aplikácie teda nie sú vykonávané priamo procesorom. Výsledkom kompilácie .NET aplikácie nie je natívny kód, ale takzvaný „medzijazyk“ IL (Intermediate Language) alebo MSIL (Microsoft Intermediate Language). Spôsob vykonávania .NET programov je znázornený na obrázku 1.1. Zdrojový kód .NET jazyka je skompilovaný špecifickým kompilátorom do CIL kódu. Kompilácia prebieha pred spustením programu. Výsledkom kompilácie je spustiteľný súbor. Po spustení súboru sa o vykonávanie stará CLR, ktoré počas behu programu prekladá CIL inštrukcie do natívneho kódu určeného pre procesor.



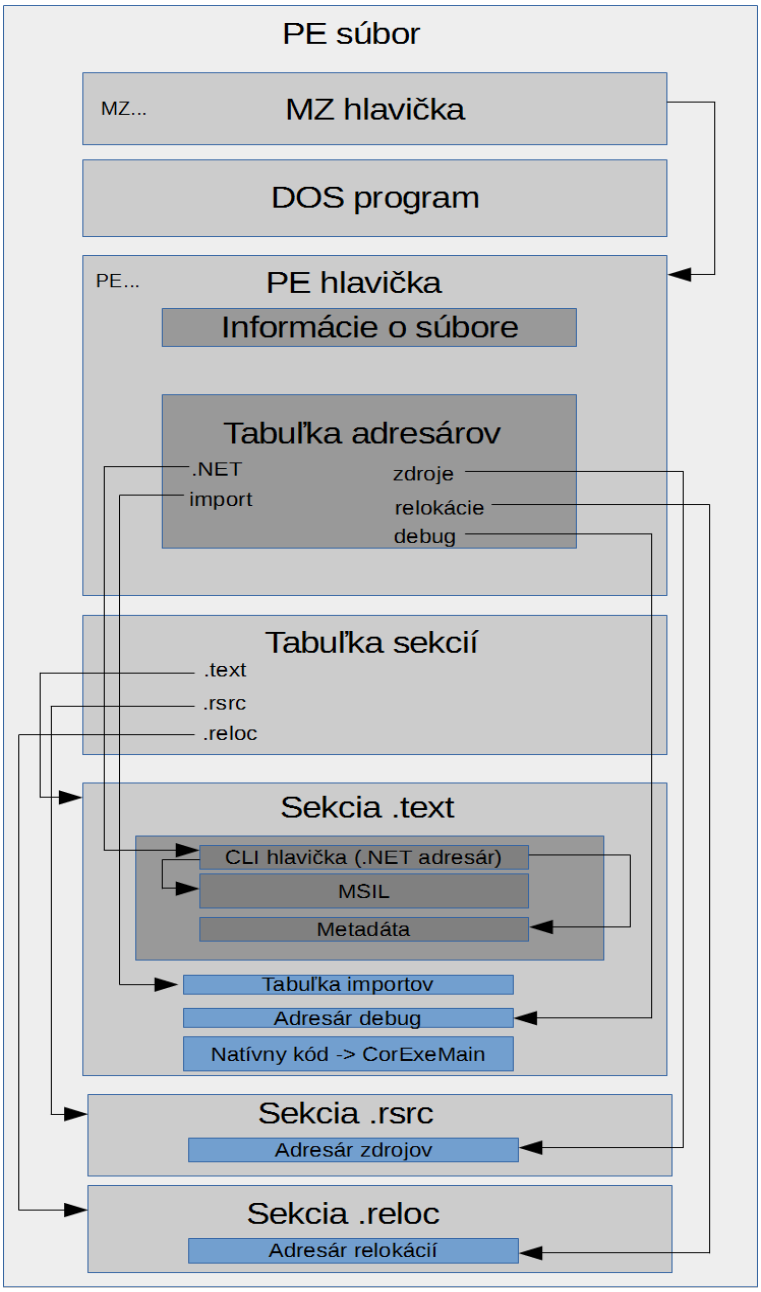
Obr. 1.1: Kompilácia .NET aplikácie [18].

Programový kód, vykonávaný priamo procesorom, sa nazýva nemanažovaný. Kód určený na vykonávanie v softvérovom prostredí ako CLR alebo Java Virtual Machine je manažovaný kód. Keďže prekladanie inštrukcií do natívneho kódu prebieha v čase vykonávania programu JIT (Just In Time) kompilácia, je možné program určitým spôsobom kontrolovať, čo má za následok niekoľko výhod pre programátora. Medzi najdôležitejšie patria:

- Garbage Collection: nepotrebné objekty netreba odstraňovať ručne, CLR to urobí automaticky.
- Bezpečnosť: CLR spravuje pamäť. Počas behu kontroluje hranice alokovanej pamäte, následkom čoho sa dá vyhnúť zásahom mimo povolenú pamäť, a teda chybám ako pretečenie buffera. Kontroluje sa aj typ objektov, s ktorými sa pracuje.
- Optimalizácia: CLR môže za behu program optimalizovať. Keďže kompilácia prebieha v čase behu programu, CLR môže inštrukcie nahradiť inými inštrukciami, prípadne inštrukciu nevykonať vôbec, a tým zefektívniť vykonávanie programu za predpokladu, že výsledný optimalizovaný kód bude ekvivalentný neoptimalizovanému.
- Portabilita: možno ju dosiahnuť implementovaním .NET frameworku pre konkrétne platformy.
- Jazyková nezávislosť: platforma .NET nevyžaduje použitie konkrétneho programovacieho jazyka.

1.2 Súborový formát Portable Executable

Výsledkom kompilácie .NET jazyka pre platformu Windows je binárny súbor (modul) vo forme dll knižnice alebo spustiteľného súboru. Tento binárny súbor neobsahuje len kód určený na vykonávanie, ale aj dodatočné informácie o programovom kóde. Je preto nutné, aby súbor zachovával určitú presne definovanú štruktúru. Na základe tejto štruktúry potom operačný systém vie správne interpretovať dáta v súbore, načítať programový kód do pamäte a spustiť ho. Tento formát súboru sa nazýva PE (Portable Executable) a jeho štruktúra je znázornená na obrázku 1.2. Teraz stručne popíšeme podstatné časti PE súboru.



Obr. 1.2: .NET Portable Executable formát.

PE súbor sa skladá z niekoľkých hlavičiek a sekcií. Hlavičky nesú informácie o ďalších hlavičkách alebo o sekciách. Sekcie obsahujú tabuľky s rôznymi údajmi o programovom kóde alebo samotný kód. PE súbor začína MZ hlavičkou. Názov je určený podľa prvých dvoch bajtov, ktoré sú ASCII znaky M a Z. MZ hlavička obsahuje odkaz na PE hlavičku. Za ňou nasleduje krátky kód, ktorý je vykonaný v prípade, že je program spustený pod operačným systémom MS-DOS a jeho úlohou je iba vypísať hlášku o tom, že program nemôže bežať v danom systéme. V PE hlavičke sú informácie ako entrypoint (adresa programového kódu, ktorým sa začína vykonávanie programu), adresa, na ktorú má byť program načítaný v pamäti (image base), verzia operačného systému, pre ktorý je modul určený a mnoho ďalších informácií potrebných pre operačný systém. PE hlavička ďalej obsahuje informácie o tabuľkách (Data directories) nachádzajúcich sa v jednotlivých sekciách. Sú to ich adresa, názov a sekcia, v ktorej sa nachádzajú. Podstatné sú tabuľka importov, zdrojov, relokácií a CLI hlavička. Za PE hlavičkou nasleduje tabuľka sekcií s informáciami o jednotlivých sekciách. Tie obsahujú rôzne typy dát. Najdôležitejšie sekcie sú .text, .rsrc a .reloc. Sekcia .text obsahuje programový kód určený na vykonávanie. Okrem MSIL kódu sa v .text sekcii nachádza aj natívny kód, ktorým začína vykonávanie .NET aplikácie. Ten obsahuje skok na jedinú importovanú funkciu _CorExeMain (exe súbor) prípadne _CorDllMain (dll knižnica) v module mscoree.dll. Funkcia _CorExeMain incializuje CLR, lokalizuje entrypoint MSIL kódu a spustí jeho vykonávanie. Informácie o funkciách, ktoré sa volajú z iných modulov, sa nachádzajú v tabuľke importov. Operačný systém tak vie, že má ešte načítať do pamäte ďalšie moduly, ktoré budú použité pri vykonávaní programu a pred spustením programu doplní adresy týchto funkcií do kódu podľa toho, na aké adresy boli potrebné moduly načítané. Sekcia .rsrc obsahuje dáta ako obrázky, ikony a ďalšie zdroje, s ktorými modul pri vykonávaní pracuje. Sekcia .reloc nesie informácie o vykonaných relokáciach. Ak program očakáva, že bude načítaný na konkrétnu adresu v pamäti a táto adresa nie je voľná, operačný systém načíta program na inú adresu a musí upraviť (relokovať) program tak, aby bol schopný pracovať aj na inej začiatočnej adrese, než akú predpokladal. Tabuľka relokácií obsahuje informácie o miestach v programe, ktoré v takom prípade treba upraviť.

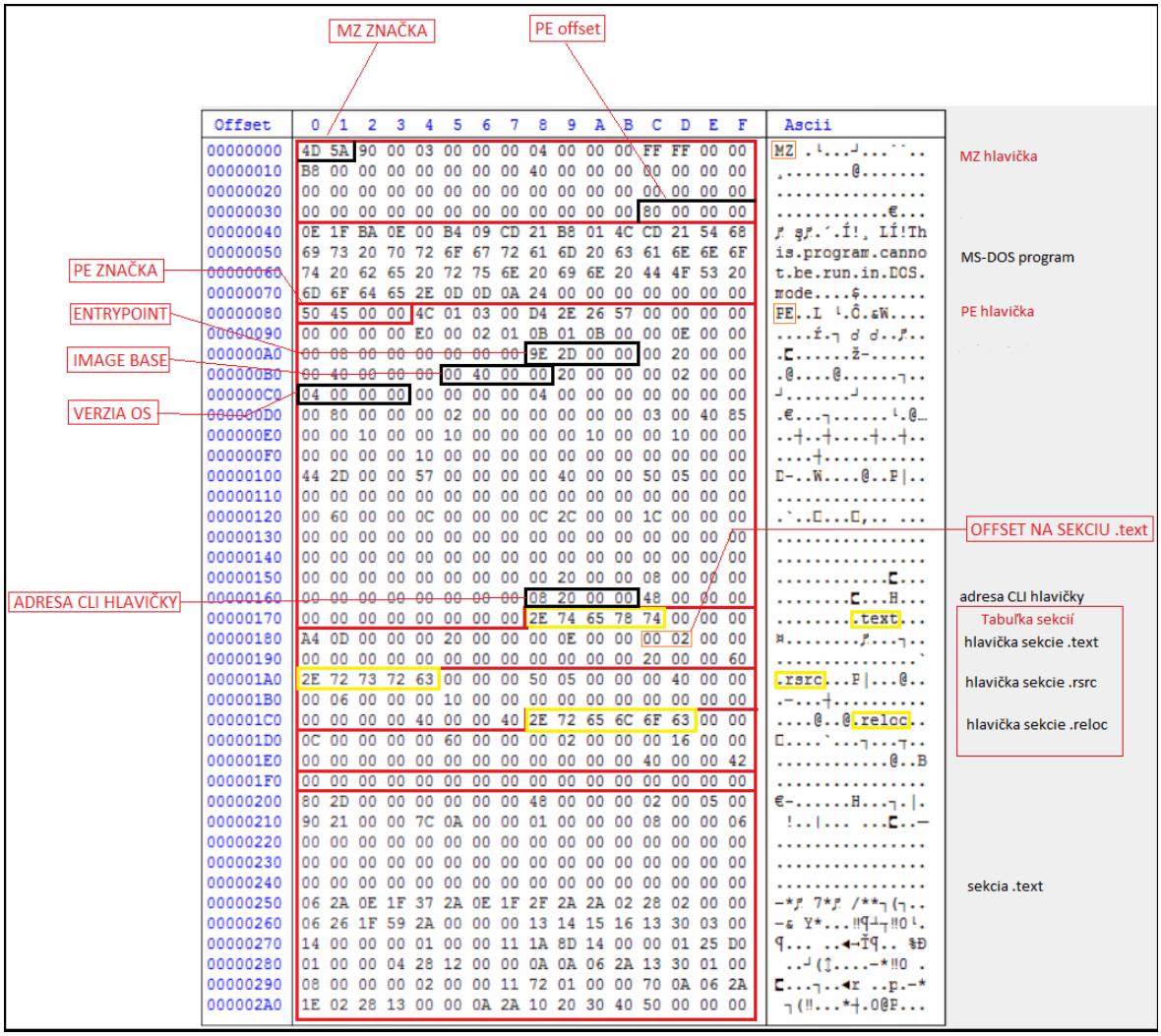
Na obrázku 1.3 je zdrojový kód programu HelloWorld. Časť skompilovaného kódu, ktorý zodpovedá programu HelloWorld je na obrázku 1.4 zobrazený pomocou hex editora. Obrázok obsahuje tri stĺpce. Prvý stĺpec obsahuje adresu od začiatku súboru. V druhom stĺpci sú dáta reprezentované ako čísla v šestnástkovej sústave. Dáta sú ukladané ako little-endian, to znamená, že menej významný bajt je umiestnený na nižšej adrese. Textová ASCII reprezentácia týchto dát je v treťom stĺpci. Na obrázku sú zvýraznené niektoré dôležité časti PE súboru.

```
using System;
using System.IO;
namespace HelloWorld {
  public class TriedaFero {

   public unsafe void funkcia(char* buffer, int* x, String str,
     byte[] dynamickepoleb,byte[] poleb, int[] poleint, char[] polechar){
       return; }
   public int voidfunkcia() { return 55; }  // žiadny argument
   public int stringfunkcia(String s) { return 47; } // argument string
   public char integerfunkcia(int x) { return 'Y'; }  // argument int
   public byte[] charfunkcia(char x) {
      byte[] polebajtov = { 0x13, 0x14, 0x15, 0x16 }; return polebajtov;
      }
   public String boolfunkcia(bool val) {
      String str = "Moj String";
      return str;
      }
   }

  public class Program {
      unsafe  public static void Main(string[] args) {
      triedaFero f = new triedaFero();
      int x = 55;
      char c = 'R';
      byte[] data={0x10,0x20,0x30,0x40,0x50};
      char[] field={'a','b','c','d'};
      int[] pole = { 22, 33, 44, 55, 66 };
      byte[] dynamic=new byte [9708];
      dynamic[0] = 9; dynamic[1] = 8; dynamic[2] = 7; dynamic[3] = 6;
      unsafe { f.funkcia(&c,&x, "Ahoj", dynamic,data ,pole, field);}
      f.stringfunkcia("Text");
      f.integerfunkcia(47);
      f.charfunkcia('A');
      f.boolfunkcia(true);
      Console.ReadLine();
      }
    }
  }
```

Obr. 1.3: Program Hello World



Obr. 1.4: HelloWorld.exe.

1.3 Metadáta

V časti 1.2 sme spomínali CLI hlavičku. Táto štruktúra obsahuje informácie o MSIL kóde potrebné pre CLR, napríklad entrypoint MSIL kódu a adresu, kde sa nachádzajú metadáta. Metadáta sú binárne informácie popisujúce programový kód MSIL, ktorý sa nachádza v súbore PE alebo v pamäti. Metadáta uchovávajú informácie ako názvy a typy premenných, tried a ich členov, metód a ich parametrov a návratových hodnôt, ktoré sú definované v danom module, alebo na ktoré modul odkazuje a ďalšie iné. Pri spustení .NET aplikácie sú metadáta načítané spolu s programovým kódom do pamäte a používané pri vykonávaní programu. Metadáta prinášajú niekoľko výhod. Keďže metadáta sú generované automaticky kompilátorom, odpadá potreba použitia hlavičkových súborov, alebo IDL (Interface Definition Language) súborov, ktoré definujú a popisujú rozhrania, prostredníctvom ktorých môžu komunikovať odlišné moduly [11]. Štruktúra metadát v PE súbore je nasledovná. Začína hlavičkou, ktorá obsahuje základné informácie o metadátach. Podstatné pre nás budú takzvané prúdy metadát nasledujúce za hlavičkou. Základných je päť, a každý prúd obsahuje pole dát určitého typu. Sú to tieto [3]:

1. #~ : Tento prúd je odlišný od nasledujúcich. Obsahuje tabuľky metadát. Každá tabuľka popisuje iný typ dát a obsahuje zoznam položiek a k nim dáta a indexy do polí v nasledujúcich prúdoch. Popis niektorých dôležitých tabuliek metadát je v tabuľke 1.1.

2. #**Strings** : Prúd obsahuje pole všetkých textových reťazcov v kódovaní UTF8 ukončené nulou. Ako vyzerá prúd #Strings v skompilovanom súbore, je možné vidieť na obrázku 1.5 (a).

3. #**US** : V prúde sa nachádza pole všetkých užívateľských textových reťazcov (User Strings odtiaľ názov) kódovaných ako 16 bitové Unikódové reťazce. Každý reťazec začína číslom určujúcim dĺžku reťazca v počte bajtov. Na konci reťazca je ešte jeden bajt navyše, ktorý je rovný 1 v prípade, že aspoň jeden UTF16 znak má nenulový horný bajt, alebo ak je dolný bajt jedna z nasledujúcich hodnôt: 0x01–0x08, 0x0E–0x1F, 0x27, 0x2D, 0x7F. Ináč je nastavený na 0. Hodnota 1 značí, že znaky musia byť spracované špeciálnym spôsobom, na rozdiel od štandardného 8-bitového kódovania. Príklad: Na obrázku 1.3 funkcia boolfunkcia vracala string inicializovaný na "Moj String". Tento textový reťazec bude uložený v prúde #US. Na obrázku 1.5 (b) je prúd #US v súbore HelloWorld.exe.

4. #**GUID** : Prúd uchováva všetky GUID čísla obsiahnuté v module. GUID objasníme v kapitole 2.

5. #**Blob** : Tento prúd obsahuje definície tried a ich členov, funkcií a ich parametrov a návratových hodnôt.

Tabuľka 1.1: Tabuľky metadát [1].

Tabuľka a jej index	**Typ dát, ktoré obsahuje**
Module (0x00)	informácie o danom module
TypeRef (0x01)	informácie o typoch referencovaných v iných moduloch
TypeDef (0x02)	informácie o typoch definovaných v danom module
MethodDef (0x06)	informácie o metódach (funkciách) definovaných v danom module
Param (0x08)	informácie o parametroch definovaných metód
AssemblyRef (0x23)	informácie o moduloch referencovaných z daného modulu
. . .	

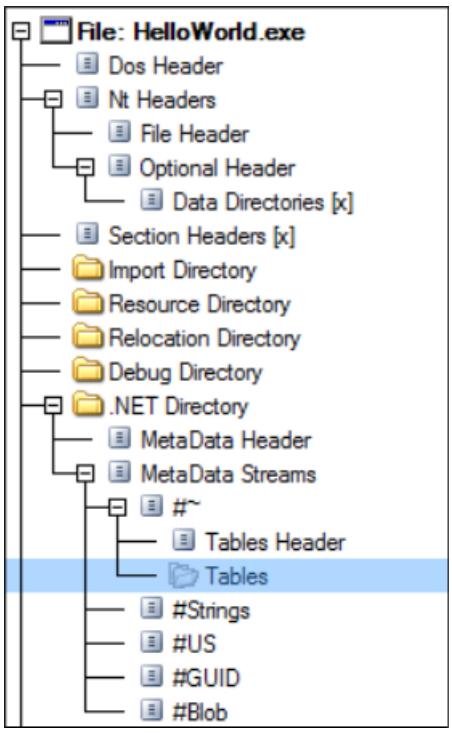
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000770													00	3C	4D	6F	.<Mo
00000780	64	75	6C	65	3E	00	48	65	6C	6C	6F	57	6F	72	6C	64	dule>.HelloWorld
00000790	2E	65	78	65	00	54	72	69	65	64	61	46	65	72	6F	00	.exe.TriedaFero.
000007A0	48	65	6C	6C	6F	57	6F	72	6C	64	00	50	72	6F	67	72	HelloWorld.Progr
000007B0	61	6D	00	6D	73	63	6F	72	6C	69	62	00	53	79	73	74	am.mscorlib.Syst
000007C0	65	6D	00	4F	62	6A	65	63	74	00	66	75	6E	6B	63	69	em.Object.funkci
000007D0	61	00	76	6F	69	64	66	75	6E	6B	63	69	61	00	73	74	a.voidfunkcia.st
000007E0	72	69	6E	67	66	75	6E	6B	63	69	61	00	69	6E	74	65	ringfunkcia.inte
000007F0	67	65	72	66	75	6E	6B	63	69	61	00	63	68	61	72	66	gerfunkcia.charf
00000800	75	6E	6B	63	69	61	00	62	6F	6F	6C	66	75	6E	6B	63	unkcia.boolfunkc
00000810	69	61	00	2E	63	74	6F	72	00	4D	61	69	6E	00	62	75	ia..ctor.Main.bu
00000820	66	66	65	72	00	78	00	73	74	72	00	64	79	6E	61	6D	ffer.x.str.dynam
00000830	69	63	6B	65	70	6F	6C	65	62	00	70	6F	6C	65	62	00	ickepoleb.poleb.

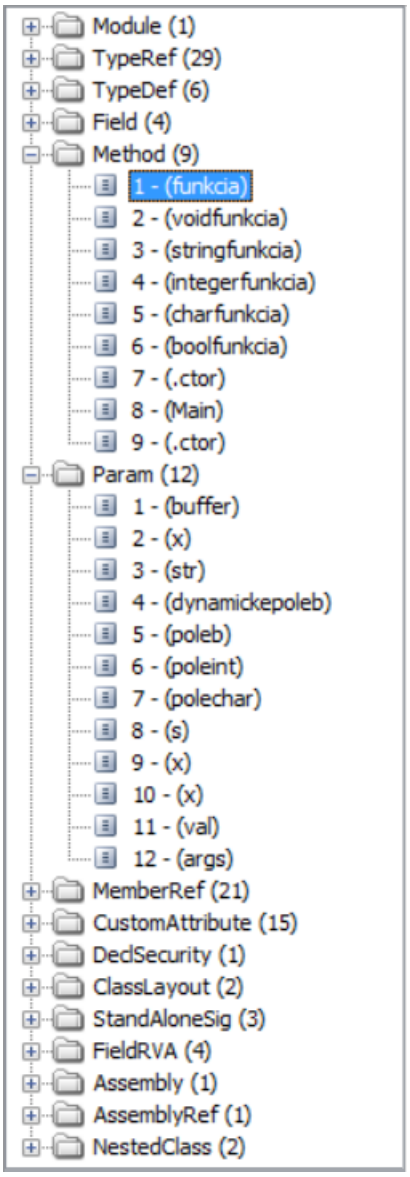
(a) prúd #Strings

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Ascii
00000C00													dĺžka	15	4D	00	↓M.
00000C10	6F	00	6A	00	20	00	53	00	74	00	72	00	69	00	6E	00	o.j...S.t.r.i.n.
00000C20	67	00	00	09	41	00	68	00	6F	00	6A	00	00	09	54	00	g..A.h.o.j...T.
00000C30	65	00	78	00	74	00	00	00									e.x.t...

(b) prúd #US

Obr. 1.5: Časti prúdov v súbore HelloWorld.exe.





Member	Offset	Size	Value	Meaning
RVA	00000574	Dword	00002050	
ImplFlags	00000578	Word	0000	Click here
Flags	0000057A	Word	0086	Click here
Name	0000057C	Word	004E	funkcia
Signature	0000057E	Word	000A	Blob Index
ParamList	00000580	Word	0001	Param Table Index 1

Obr. 1.6: HelloWorld.exe v programe CFF Explorer.

Tabuľka 1.2: Dáta v tabuľke MethodDef pre funkciu „funkcia“.

Položka	**Hodnota**	**Význam políčka**
RVA	0x2050	Adresa funkcie
ImplFlags	0x0	Typ kódu MSIL, manažovaný
Flags	0x86	Bližšia špecifikácia funkcie (Public)
Name	0x4e	Meno (index do prúdu #Strings)
Signature	0x0A	Signatúra (index do prúdu #Blob)
ParamList	1	Parametre funkcie (index do tabuľky Param)

Veľmi dobrý nástroj pre analýzu skompilovaného súboru je program CFF Explorer. Na obrázku 1.6 je zobrazený program HelloWorld.exe. Nástroj umožňuje prezeranie obsahu súboru a jednotlivých jeho štruktúr. Pre lepšie pochopenie ilustrujeme štruktúru metadát v PE súbore pre funkciu „funkcia“ z programu HelloWorld. Dáta v tabuľke MethodDef pre funkciu „funkcia“, ktorá je definovaná v module HelloWorld.exe, vyzerajú nasledovne. Na obrázku 1.5 prúd #Strings začína na adrese 0x77C a textový reťazec „funkcia“ je uložený na adrese 0x7CA. Môžno si všimnúť, že hodnota 0x4E skutočne zodpovedá danému textovému reťazcu, keďže 0x7CA-0x77C=0x4E. Dôležitá položka v tabuľke 1.2 je signatúra. Signatúry sú štruktúry, v ktorých sú uchovávané metadáta v prúde #Blob.

Pred popísaním signatúr ešte uvedieme jednu dôležitú vec. S narastaním programového kódu .NET aplikácie rastie aj počet metadát popisujúcich daný program, čo vedie k narastaniu veľkosti spustiteľných súborov. Štvorbajtové celé čísla často obsahujú malé hodnoty reprezentovateľné aj menším počtom bajtov. Aby nebolo zbytočne zaberané miesto v pamäti alebo súbore, kompilátor sa snaží o optimalizáciu. Komprimácia neznamienkových celých čísel je popísaná v tabuľke 1.3 a prebieha nasledovným spôsobom:

Číslo je podľa veľkosti kódované jedným, dvoma alebo štyrmi bajtami a na prvý bajt je aplikovaná maska ako logický súčin, na základe čoho je potom možné dáta dekomprimovať.

Tabuľka 1.3: Reprezentácia neznamienkových čísel [9].

Rozsah	**Počet bajtov**	**Maska**	**Binárny zápis**
[00000000h,0000007Fh]	1	80h	0BBBBBBB
[00000080h,00003FFFh]	2	C0h	10BBBBBB BBBBBBBB
[00004000h,1FFFFFFFh]	4	E0h	110BBBBB BBBBBBBB BBBBBBBB BBBBBBBB

- Rozsah udáva hodnoty (vrátane maxima a minima), ktoré môže číslo mať, aby bolo kódované daným spôsobom. Najväčšie číslo, ktoré možno skomprimovať je 2^{29}.

- Počet bajtov hovorí o tom, koľkými bajtami bude reprezentované skomprimované číslo.
- Maska je aplikovaná na prvý bajt komprimovaného čísla ako logický súčin s danou hodnotou.
- Binárny zápis čísla po komprimácii. Treba dodať, že čísla v tabuľke sú reprezentované ako big-endian, teda významnejší bajt je skôr.

Dekompresia čísla prebieha takto. Ak je prvý bajt tvaru $0bbbbbbb_2$, hodnota čísla je $bbbbbbb_2$. Ak je prvý bajt tvaru $10bbbbbb_2$ a druhý bajt je x, hodnota reprezentovaného čísla je ($bbbbbb_2$ « 8 + x). Ak prvý bajt je tvaru $110bbbbb_2$ a nasledujúce bajty sú x, y, z, výsledná hodnota nekomprimovaného čísla bude ($bbbbb_2$ « 24 + x « 16 + y « 8 + z) [3].

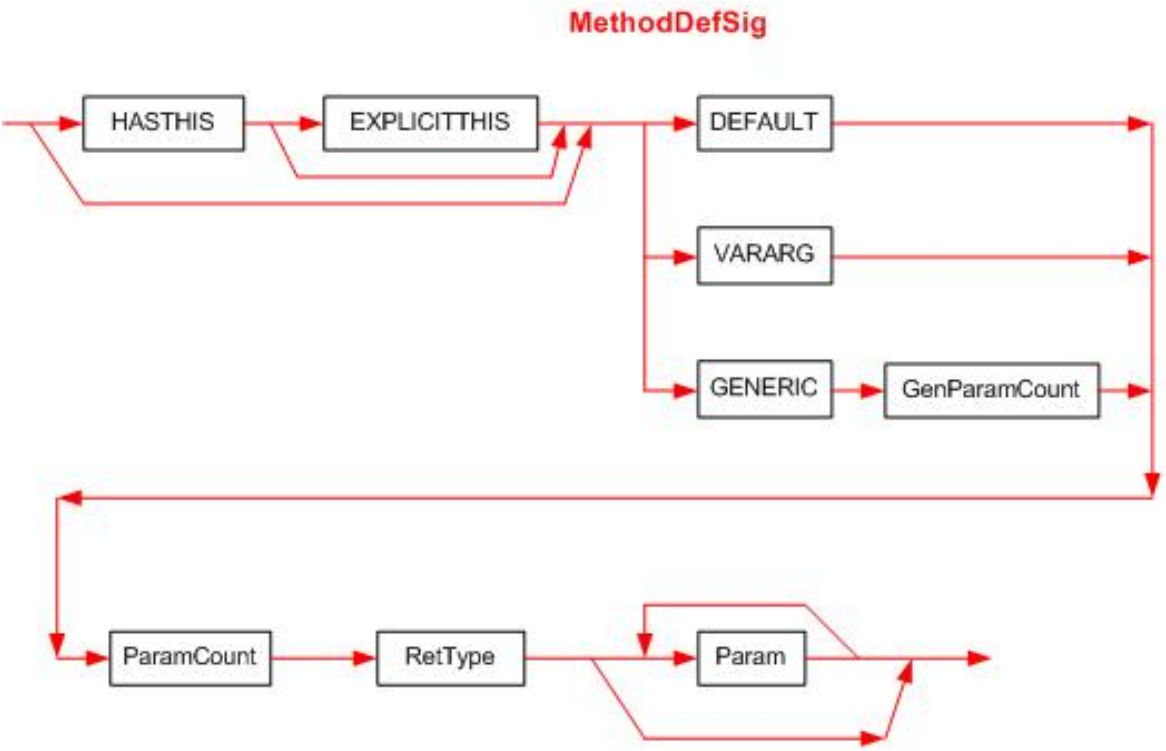
CorElementTypeEnum špecifikuje typy, ktoré môžu byť uložené v signatúre. Na obrázku 1.7 je niekoľko základných typov dát. Jednoduché typy ako boolean a char sú uložené v signatúre priamo hodnotou. V prípade zložitejších typov, ako jednorozmerné pole alebo smerník, nasleduje za danou hodnotou v signatúre ešte hodnota typu, ktorá sa viaže k zložitému typu alebo token odkazujúci na ďalšiu signatúru.

```
typedef enum CorElementType{
          ELEMENT_TYPE_VOID          = 0x1,  void
          ELEMENT_TYPE_BOOLEAN       = 0x2,  boolean
          ELEMENT_TYPE_CHAR          = 0x3,  char
          ELEMENT_TYPE_I1            = 0x4,  znamienkové celé číslo 8 bitov
          ELEMENT_TYPE_U1            = 0x5,  neznamienkové celé číslo 8 bitov
          ELEMENT_TYPE_I2            = 0x6,  znamienkové celé číslo 16 bitov
          ELEMENT_TYPE_U2            = 0x7,  neznamienkové celé číslo 16 bitov
          ELEMENT_TYPE_I4            = 0x8,  znamienkové celé číslo 32 bitov
          ELEMENT_TYPE_U4            = 0x9,  neznamienkové celé číslo 32 bitov
          ELEMENT_TYPE_I8            = 0xa,  znamienkové celé číslo 64 bitov
          ELEMENT_TYPE_U8            = 0xb,  neznamienkové celé číslo 32bitov
          ELEMENT_TYPE_R4            = 0xc,  float 32 bitov
          ELEMENT_TYPE_R8            = 0xd,  float 64 bitov
          ELEMENT_TYPE_STRING        = 0xe,  string
          ELEMENT_TYPE_PTR           = 0xf,  smerník
          ELEMENT_TYPE_BYREF         = 0x10, referencia
          ELEMENT_TYPE_VALUETYPE     = 0x11, struct / enum
          ELEMENT_TYPE_CLASS         = 0x12, trieda
          ELEMENT_TYPE_VAR           = 0x13, statická premenná
          ELEMENT_TYPE_ARRAY         = 0x14, viacrozmerné pole
          ELEMENT_TYPE_GENERICINST   = 0x15, generický typ
          ELEMENT_TYPE_FNPTR         = 0x1B, smerník na funkciu
          ELEMENT_TYPE_OBJECT        = 0x1C, typ System.Object
          ELEMENT_TYPE_SZARRAY       = 0x1D, jednorozmerné pole
          ELEMENT_TYPE_MAX           = 0x22, nesprávny typ
          .
          .
          .

} CorElementType;
```

Obr. 1.7: CorElementType [14].

Vráťme sa teraz k príkladu metadát pre funkciu „funkcia“. Štruktúru signatúry MethodDefSig, ktorá popisuje funkciu a jej parametre znázorňuje diagram na obrázku 1.8. Prvý bajt signatúry charakterizuje volaciu konvenciu a početnosť generických parametrov. Vznikne logickým súčtom konštánt, ktoré sú popísané v tabuľke 1.4. Ak bol nastavený flag GENERIC, tak nasledujúca hodnota v signatúre určuje počet generic-

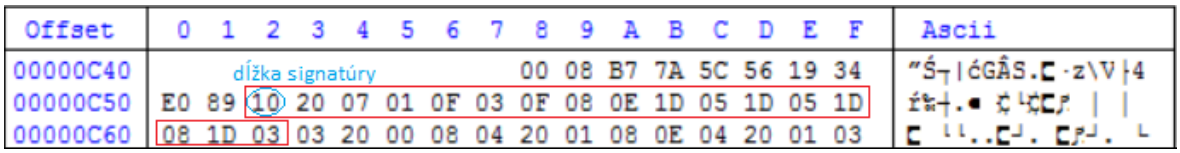


Obr. 1.8: Signatúra MethodDefSig [3].

Názov	**Hodnota**	**Význam**
HASTHIS	0x20	prvý argument funkcie je this
EXPLICITTHIS	0x40	signatúra obsahuje typ this parametra
DEFAULT	0x00	východzia volacia konvencia
VARARG	0x05	funkcia má premenlivý počet argumentov
GENERIC	0x08	metóda má generické parametre

Tabuľka 1.4: Flagy charakterizujúce signatúru MethodDefSig.

kých parametrov metódy. Ďalej nasleduje počet argumentov funkcie a typ návratovej hodnoty. V prípade, že bol nastavený flag EXPLICITTHIS, nasleduje typ this parametra. Na konci signatúry sú typy jednotlivých parametrov funkcie. Signatúra pre funkciu „funkcia“ v súbore HelloWorld.exe je zobrazená na obrázku 1.9. Signatúra začína na adrese 0xC52, čo je práve desiaty index (0xA) v prúde #Blob, rovnako ako uvádza tabuľka 1.2. Číslo 0x10 určuje dĺžku signatúry, takže nasledujúcich 16 bajtov bude opisovať funkciu „funkcia“ z programu HelloWorld na obrázku 1.3. Hodnota 0x20 znamená, že funkcia bude mať nultý argument, čo je smerník this. Keďže flag EXPLICITTHIS nie je nastavený, popis typu smerníka this v signatúre nie je. Počet argumentov funkcie je 7 a typ návratovej hodnoty (0x01) je podľa obrázka 1.7 void. Prvý argument je typu smerník (0x0f) na char (0x03). Druhý argument je typu smerník (0x0f) na int (0x08). Tretí argument je typu string (0x0e) a nasledujú štyri polia



Obr. 1.9: Signatúra MethodDefSig v prúde #blob.

(0x1d) typu byte (0x05), byte (0x05), int (0x08) a char (0x03). Je možné všimnúť si, že ani jedna hodnota v signatúre nebola väčšia ako 0x7F, teda nebolo nutné komprimovať žiadnu hodnotu. Pri zložitejších signatúrach s väčšou dĺžkou a zložitejšou štruktúrou budú čísla uložené v skomprimovanej forme.

Nakoniec tejto časti uvedieme ešte jeden dôležitý pojem. **Metadatatoken** je štvorbajtové číslo, ktoré jednoznačne identifikuje záznam v tabuľke metadát v prúde #~. Najvýznamnejší bajt určuje tabuľku metadát a ostatné tri bajty záznam. Príklad: Metadatatoken 0x06000001 reprezentuje tabuľku MethodDef (0x06) a číslo 0x01 prvý záznam, ktorý obsahuje informácie o funkcii „funkcia“.

Kapitola 2

Profilovanie aplikácií

Kapitola detailne opíše spôsob získania informácií o vyvolaných funkciách, ich argumentoch a návratových hodnotách v .NET aplikáciách.

2.1 Základné pojmy

Na začiatku uvedieme niekoľko pojmov potrebných pre pochopenie princípu sledovania vybraných volaní.

ABI: Application Binary Interface. Definuje rozhranie medzi dvoma programovými modulmi na úrovni strojového kódu. ABI kompatibilita zahŕňa kompatibilitu v nasledovných vlastnostiach:

- Veľkosť, reprezentácia a zarovnanie dátových typov v pamäti.
- Volacia konvencia. Argumenty volanej funkcie môžu byť uložené všetky buď na zásobníku, alebo niektoré v registroch. Dôležité je tiež samotné poradie argumentov. Návratová hodnota funkcie je uložená väčšinou v registri EAX. Po návrate z funkcie treba dáta zo zásobníka odstrániť. Za odstránenie môže byť zodpovedný volajúci alebo volaná funkcia. Všetky tieto vlastnosti sú určené použitím konkrétnej volacej konvencie. Preto ak napríklad jeden modul volá funkciu v inom module s hoci aj správnymi parametrami (typ, počet, hodnota), ale volacie konvencie určené pri kompilovaní sú odlišné, program nebude fungovať, pretože k dátam v pamäti bude pristupovať iným spôsobom a dôjde k chybe.

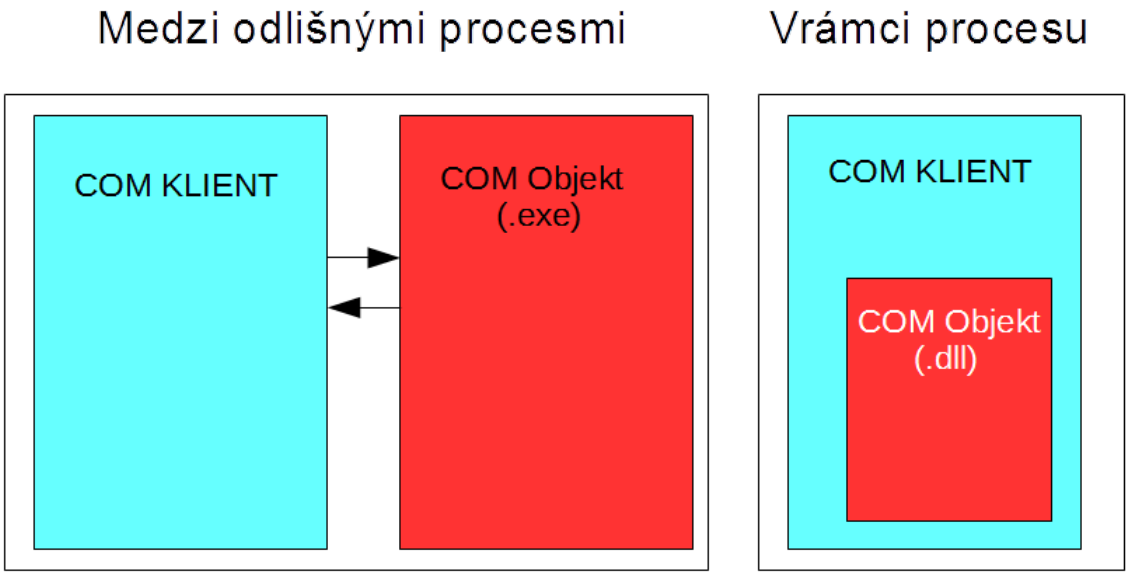
Operačný systém Windows umožňuje zdieľanie binárneho kódu prostredníctvom dynamicky pripájaných (DLL) knižníc. Tie obsahujú skompilovaný zdrojový kód napísaný v konkrétnom programovacom jazyku, napríklad Java. Ak by sme chceli používať tento kód z modulu, ktorý bol vytvorený použitím iného programovacieho jazyka, c++ alebo

niektorý z .NET jazykov, bolo by nutné, aby moduly boli ABI kompatibilné, čo však nie je možné, keďže boli použité odlišné programovacie jazyky.

2.1.1 COM

COM je štandard, ktorý problém ABI kompatibility rieši zavedením jednotného ABI rozhrania. Určuje, ako majú byť exe súbory a dll knižnice skompilované. Definuje presnú štruktúru a organizáciu modulu v pamäti. COM umožňuje použitie objektov bez ohľadu na použitý programovací jazyk a vnútornú implementáciu. Skompilovaný binárny súbor však nesmie závisieť od špecifickej črty programovacieho jazyka. Jediná požiadavka na programovací jazyk je, aby mal podporu volania funkcií prostredníctvom smerníkov na tieto funkcie.

Objekt predstavuje štruktúru obsahujúcu dáta a metódy. Prostredníctvom metód sa manipuluje s dátami. Vzájomná interakcia medzi objektami je zabezpečená pomocou rozhraní. Rozhrania definujú metódy objektu, ich parametre a návratovú hodnotu. Objekt je definovaný prostredníctvom triedy. COM trieda sa nazýva coclass a implementuje rozhranie alebo niekoľko rozhraní, ktoré dedia od rozhrania IUnknown popísané v tabuľke 2.1. Konkrétna inštancia triedy coclass sa nazýva COM objekt. Interakcia medzi COM objektami prebieha na úrovni klient-server a znázorňuje ju obrázok 2.1. COM server poskytuje klientovi služby prostredníctvom rozhrania. Komunikáciu medzi COM objektami znázorňuje obrázok. Tá môže byť v rámci procesu, vtedy je COM server implementovaný ako dynamická knižnica (COM DLL) alebo mimoprocesová.



Obr. 2.1: Komunikácia COM objektov [6].

Tabuľka 2.1: rozhranie IUnknown [8].

Metóda	Popis
AddRef	Inkrementuje počítadlo referencií na objekt
QueryInterface	Vráti smerník na tabuľku virtuálnych metód
Release	Dekrementuje počítadlo referencií na objekt

Metódy Addref a Release manipulujú s počítadlom referencií, ktoré existujú na COM objekt. Ak program požiada o konkrétny COM objekt a rozhranie, ktoré implementuje, zodpovedajúca knižnica je načítaná operačným systémom do pamäte, odovzdá sa referencia na daný objekt a inkrementuje sa počítadlo referencií na objekt a na knižnicu. Ak počet referencií klesne na 0, a teda objekt už viac nie je používaný, objekt je odstránený z pamäte. Metóda QueryInterface dostane ako parameter IID rozhrania, ktoré objekt implementuje a vráti smerník na tabuľku obsahujúcu smerníky na všetky implementované funkcie daného rozhrania.

COM DLL môže obsahovať viacero coclass tried a každá z týchto tried môže implementovať viacero rozhraní. Každé rozhranie a každá trieda má pridelené vlastné identifikačné číslo (GUID) označované ako IID (interface ID) alebo CLSID (class ID). Microsoft používa GUID na referencovanie tried a rozhraní objektov v systéme Windows. GUID (Globally Unique Identifier) je jedinečné 128-bitové číslo zobrazované ako 32 hexadecimálnych cifier oddelených pomlčkou.
Príklad: 21EC2020-3AEA-4069-A2DD-08002B30309D. Výhoda použitia GUID spočíva v tom, že pri odkazovaní sa na objekt nie je nutné poznať cestu k objektu, iba príslušné číslo. Pred použitím je však potrebné zaregistrovať COM DLL do Windows register databázy. COM DLL exportuje funkcie popísané v tabuľke 2.2.

Tabuľka 2.2: exporty COM DLL.

Metóda	Popis
DllRegisterServer	Pri registrácii objektu zapíše do registrov cestu k DLL.
DllUnregisterServer	Odstráni cestu ku knižnici z registrov.
DllGetClassObject	EntryPoint.
DllCanUnloadNow	Zistí, či môže byť knižnica uvoľnená z pamäte.

COM klient zavolá funkciu CoCreateInstance s CLSID coclass triedy a IID rozhrania, ktoré coclass implementuje. Táto funkcia vytvorí COM objekt a vráti smerník na požadované rozhranie. Interne sa udejú nasledovné veci [2]:

1. COM knižnica, ktorú klient používa nájde v registroch cestu ku COM DLL serveru na základe CLSID.

2. COM knižnica načíta do pamäte COM server a zavolá funkciu DllGetClassObject.

3. COM DLL server okrem toho, že obsahuje implementáciu jednej alebo niekoľko coclass tried, obsahuje ku každej definovanej coclass triede ešte jednu špeciálnu coclass triedu classfactory, ktorá dedí od rozhrania IClassFactory. Úlohou tejto triedy je vytvoriť inštanciu triedy, ku ktorej zodpovedá. DllGetClassObject vytvorí inštanciu classfactory a vráti na ňu smerník.

4. COM knižnica potom prostredníctvom classfactory vytvorí inštanciu COM objektu a vráti smerník na rozhranie, ktoré klient žiadal.

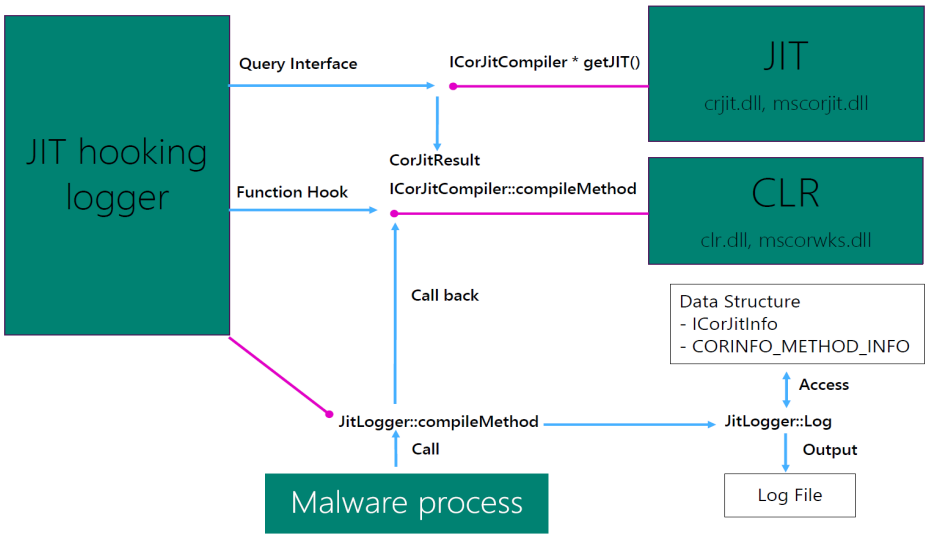
Klient odteraz môže interagovať so serverom prostredníctvom obdržaného rozhrania.

2.2 Profiler

V praxi je často potrebné kontrolovať beh aplikácie. Spôsob dynamickej analýzy bežiaceho programu, počas ktorej sa sleduje činnosť programu, časová a pamäťová zložitosť programu, čas strávený vykonávaním funkcií a frekvencia ich volania za účelom optimalizácie a analýzy programu, sa nazýva profilovanie. Profilovaním je možné odhaliť chybu alebo neefektívnu implementáciu. Aplikácia, prostredníctvom ktorej prebieha profilovanie, sa nazýva profiler. V nasledujúcom bude hlavný účel profilera sledovania vyvolaných funkcií, ich argumentov a návratových hodnôt v škodlivých aplikáciách. Bude nás teda zaujímať udalosť, keď program vstúpi do funkcie a stav tesne pred návratom z funkcie. Ďalej nás bude zaujímať udalosť načítania programového kódu do pamäte počas behu programu. Malvér zvykne programový kód uchovávať v zašifrovanej podobe. Počas behu dešifruje dáta a načíta ich do pamäte. Následne načítaný programový kód vykoná. V tomto prípade sa už nejedná o manažovaný kód, ale o natívny nemanažovaný kód, ktorý nie je možné sledovať použitím .NET profilera, keďže kód nie je vykonávaný prostredníctvom CLR. Týmto spôsobom je možné zistiť aspoň to, že program zmenil spôsob vykonávania. Teraz sa pozrieme na možnosti, ako sledovať vybrané volania:

- JIT Hook: obrázok 2.2 znázorňuje použitie tejto techniky. Ako bolo spomenuté v časti 1.1, MSIL kód je preložený do natívneho kódu JIT kompilátorom počas behu programu. Prekladanie prebieha takým spôsobom, že CLR zavolá funkciu compileMethod s argumentami, ktoré nesú informácie o prekladanej funkcii. Hooknutím tejto funkcie je možné získať prístup ku štruktúre CORINFO_METHOD_INFO a rozhraniu ICorJitInfo a následne obdržať informácie a vykonanej metóde. Tento spôsob sledovania volaní má oproti nasledujúcemu niekoľko nevýhod, a preto sa

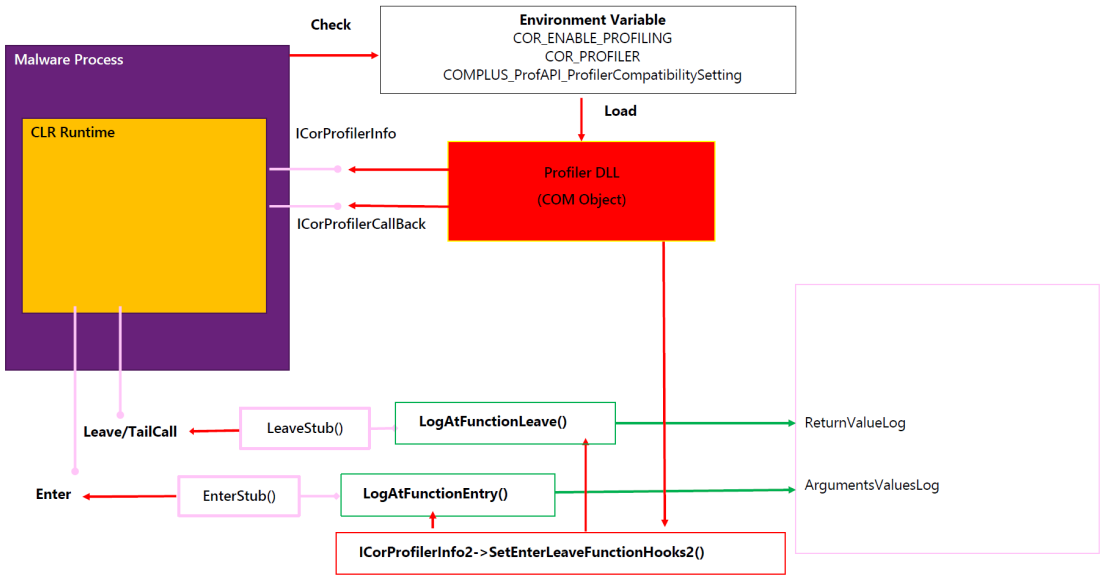
mu detailnejšie venovať nebudeme. Nevýhody spočívajú v tom, že hooknutie funkcie compileMethod umožňuje získať informácie len vtedy, ak dôjde ku kompilácii. V prípade použitia vopred skompilovaného kódu nebude možné zaznamenať žiadne volania. Platforma .NET sa navyše mení od verzie k verzii a to znamená, že pre každú verziu .NET platformy by bolo potrebné implementovať hooknutie iným spôsobom.



Obr. 2.2: JIT Hook [12].

- Autori .NET platformy umožňujú vývojárom vytvoriť vlastný profiler, ktorým budú môcť sledovať vykonávanú .NET aplikáciu, poskytnutím softvérového rozhrania na komunikáciu medzi profilerom a CLR. Platforma poskytuje API, prostredníctvom ktorého je možná notifikácia o týchto udalostiach. Narozdiel od predchádzajúceho spôsobu, umožňuje sledovať viacero rôznych udalostí a poskytuje možnosť špecifikovať, o ktorých udalostiach má CLR profiler notifikovať. Profiler pre platformu .NET je COM DLL server. S CLR komunikuje prostredníctvom niekoľkých rozhraní. Sú to rozhrania ICorProfilerCallback, ICorProfilerCallback2 a ICorProfilerCallback3.[1] Tieto rozhrania definujú callbacky, ktoré musí profiler implementovať, aby mohol získavať informácie o vybraných udalostiach. Callback predstavuje funkciu, časť programového kódu, ktorá sa nevolá priamo, ale je určená ako argument pre inú funkciu. Táto funkcia ju pri určitej udalosti zavolá. Na riadenie monitorovania slúži rozhranie ICorProfilerInfo, ICorProfilerInfo2 a ICorProfilerInfo3. Návratové hodnoty funkcií implementujúcich tieto rozhrania

[1]Názov rozhrania s vyšším číslom rozširuje predchádzajúce rozhranie o nové funkcie. Profiler, ktorý je súčasťou tejto práce implementuje rozhrania verzie 2, keďže tie rozširujú rozhrania verzie 1. Preto budeme používať ďalej v texte rozhrania s označením 2.



Obr. 2.3: API pre profilovanie .NET aplikácií [12].

sú typu HRESULT. Tie možu nadobúdať hodnoty, ktoré znázorňuje tabuľka 2.3

Tabuľka 2.3: Hodnoty HRESULT [7].

Hodnota	**Meno**	**Popis**
0x00000000	S_OK	Operácia úspešná
0x80004001	E_NOTIMPL	Neimplementované
0x80004002	E_NOINTERFACE	Rozhranie nepodporované
0x80004003	E_POINTER	Smerník nie je validný
0x80004004	E_ABORT	Operácia zrušená
0x80004005	E_FAIL	Nešpecifikovaná chyba
0x8000FFFF	E_UNEXPECTED	Neočakávaná chyba
0x80070005	E_ACCESSDENIED	Prístup zamietnutý
0x80070006	E_HANDLE	Handle nie je validné
0x8007000E	E_OUTOFMEMORY	Alokácia pamäte zlyhala
0x80070057	E_INVALIDARG	Jeden, alebo viac argumentov nie sú validné

2.2.1 Nastavenie prostredia

Pred samotným spustením .NET programu CLR kontroluje dve premenné prostredia spúšťanej aplikácie. Konkrétne sú to:

1. COR_ENABLE_PROFILIN: Táto premenná môže nadobúdať hodnoty 1 alebo 0.

Neexistencia tejto premennej alebo hodnota 0 znamená, že aplikácia nebude profilovaná a prejde sa ihneď k samotnému vykonávaniu programu. Hodnota 1 indikuje prítomnosť nástroja na profilovanie aplikácie a pokračuje sa kontrolovaním ďalšej premennej.

2. COR_PROFILER: Premenná určuje, ktorý profiler sa má použiť pri profilovaní aplikácie. Obsahuje GUID coclass triedy profilera.

3. COMPLUS_ProfAPI_ProfilerCompatibilitySetting:
.NET Framework verzie 4 štandardne načíta a spustí profiler určený pre verziu 4 (implementuje rozhranie ICorProfilerCallback3). Ak chceme použiť profiler verzie 2.0, je potrebné nastaviť túto premennú. Možné hodnoty znázorňuje tabuľka 2.4.

Tabuľka 2.4: premenná Complus [13].

Hodnota	**Činnosť CLR**
EnableV2Profiler	Načíta a spustí profiler verzie 2.0
DisableV2Profiler	Zakáže použitie profilera verzie 2.0 (východzie)
PreventLoad	Zakáže všetky profilovacie nástroje bez ohľadu na verziu.

Treba podotknúť, že ak chceme profilovať jednu konkrétnu aplikáciu, treba nastaviť tieto premenné iba danej aplikácii. Ak budú premenné nastavené globálne, profilovaná bude každá spustená .NET aplikácia.

CLR skontroluje premenné a v prípade úspešnej kontroly vytvorí inštanciu profilera a načíta ju do adresného priestoru profilovanej aplikácie, spustí aplikáciu a podľa zvolených udalostí, o ktorých má byť profiler informovaný, volá príslušné callbacky a odovzdáva informácie o vykonaných udalostiach profileru.

Po načítaní profilovanej aplikácie do pamäte dostane CLR referenciu na profiler a zavolá funkciu ICorProfilerCallback2::Initialize. Táto funkcia slúži na inicializáciu profilera. Ako parameter dostane smerník na rozhranie IUnknown, od ktorého profiler metódou QueryInterface získa smerník na rozhranie ICorProfilerInfo2. Prostredníctvom tohoto rozhrania je možné riadiť notifikáciu od CLR. Profiler môže dostávať informácie o rôznych udalostiach. Konkrétne udalosti, o ktorých bude profiler notifikovaný, sú určené použitím masky. Na základe nastavenej masky potom budú volané príslušné callbacky, ktoré profiler implementuje. Maska vznikne logickým súčtom konštánt, definujúcich sledovanie konkrétnej vlastnosti. Najpodstatnejšie sú tieto:

- COR_PRF_MONITOR_ENTERLEAVE - notifikuje profiler o udalosti, keď program zavolá funkciu, alebo sa z nej vráti.

- COR_PRF_MONITOR_ASSEMBLY_LOADS - notifikuje o udalosti, keď program načíta do pamäte programový kód.

- COR_PRF_ENABLE_FUNCTION_ARGS - profiler dostane informáciu o argumentoch volanej funkcie.
- COR_PRF_ENABLE_FUNCTION_RETVAL - profiler dostane informáciu o návratovej hodnote volanej funkcie.
- COR_PRF_DISABLE_INLINING - štandarne sa CLR pri vykonávaní programu snaží o optimalizáciu. Volanie funkcie je drahá operácia. Treba dať argumenty na zásobník, prípadne do registrov, uložiť návratovú hodnotu a vykonať funkciu. Volaná funkcia navyše obsahuje prológ a epilóg, kde sa pracuje so zásobníkom. Celá táto administrácia stojí určitý čas, a preto CLR môže program pri JIT kompilácii optimalizovať takým spôsobom, že namiesto generovania kódu, ktorý volá ďalšiu funkciu, rozbalí kód volanej funkcie do tela volajúcej funkcie. Táto optimalizácia však nemusí byť žiadaná, nakoľko chceme sledovať volania funkcií a po aplikovaní optimalizácie k žiadnemu volaniu funkcie nedôjde. Tým pádom strácame určité informácie o vykonávanom programe. Ak by do volanej funkcie vstupovali argumenty, ktorých hodnota by bola pre nás „zaujímavá“, túto hodnotu by sme nemohli získať. Na druhej strane, v kontexte škodlivých aplikácií, keď sa autori malvéru snažia o sťaženie analýzy programu generovaním množstva zbytočného kódu, kde si funkcie argumenty len „prehadzujú“, môže byť výhodné ak CLR optimalizáciu urobí.

Zavolaním funkcie ICorProfilerInfo2::SetEventMask nastavíme notifikačnú masku. Prostredníctvom funkcie ICorProfilerInfo2::SetEnterLeaveFunctionHooks2 ďalej určíme, ktoré callback funkcie budú volané pri vstúpení do funkcie, pri návrate z funkcie a vtedy, ak posledná inštrukcia funkcie je skok do inej funkcie (TailCall). Tieto callback funkcie sú však definované ako holé. To znamená, že kompilátor pri kompilácií nevytvorí funkcii prológ a epilóg. Preto je potrebné v implementácii callbacku použiť inline assembler a napísať vlastný prológ a epilóg. Špecifikovaním týchto udalostí končí inicializácia a CLR spustí aplikáciu určenú na profilovanie.

2.2.2 Informácie o vykonaných udalostiach

Teraz detailne opíšeme postupnosť krokov pre získanie informácií o vykonávanej .NET aplikácii. Funkcie uvedené nižšie sú popísané v dokumentácii Microsoftu [15] a zdrojových kódoch štandardu CLI opisujúceho .NET framework [3]. Keď program vstúpi do funkcie, CLR zavolá callback FunctionEnter2, ktorý bol špecifikovaný pri inicializácii profilera, a ktorý profiler implementuje. Funkcia má nasledovné argumenty:

```
void __stdcall FunctionEnter2 (
[in]  FunctionID                    funcId,
[in]  UINT_PTR                      clientData,
```

```
[in]  COR_PRF_FRAME_INFO                func,
[in]  COR_PRF_FUNCTION_ARGUMENT_INFO   *argumentInfo
);
```

Podstatné parametre sú funcId a *argumentInfo. FuncId je jednoznačné identifikačné číslo, ktoré CLR pridelilo funkcii. ArgumentInfo je smerník na štruktúru obsahujúcu argumenty funkcie. Túto štruktúru popíšeme neskôr. V časti 1.3 sme rozoberali metadáta nesúce všetky informácie o programovom kóde.
Prostredníctvom funkcie ICorProfilerInfo2::GetTokenAndMetaDataFromFunction

```
HRESULT GetTokenAndMetaDataFromFunction(
[in]  FunctionID functionId,
[in]  REFIID     riid,
[out] IUnknown   **ppImport,
[out] mdToken    *pToken);
```

je možné získať metadatatoken pre funkciu s identifikátorom functionId a smerník na rozhranie IMetadataImport. Parameter riid je identifikačné číslo rozhrania, o ktoré máme záujem. V tomto prípade to je id rozhrania IMetadataImport. Toto rozhranie umožňuje manipulovať s metadátami a získavať informácie o funkciách.

Funkciou IMetaDataImport::GetMethodProps získame názov volanej funkcie, smerník na signatúru popisujúcu funkciu a smerník na metadatatoken pre triedu, v ktorej je funkcia definovaná a zavolaním funkcie IMetaDataImport::GetTypeDefProps získame z tokenu názov tejto triedy. Postupnosťou týchto krokov sme obdržali názov zavolanej funkcie a triedy, v ktorej je funkcia definovaná a prístup ku signatúre popisujúcej danú funkciu. Keďže dáta v signatúre sú uložené v komprimovanej podobe, treba dáta pred spracovaním rozbaliť. Na to slúži funkcia CorSigUncompressData. Prvý argument funkcie je smerník na dáta v signatúre, ktoré treba rozbaliť. Výstupný druhý argument je adresa, kam majú byť uložené rozbalené dáta. Funkcia vráti počet bajtov, ktorými bola komprimovaná hodnota uložená v signatúre. Postupným parsovaním signatúry tak získame počet a typ argumentov a návratovej hodnoty.

```
ULONG CorSigUncompressData(
[in]  PCCOR_SIGNATURE pData,
[out] ULONG           *pDataOut);
```

Zavolaním funkcie IMetaDataImport::EnumParams obdržíme pole obsahujúce metadatatokeny pre všetky parametre funkcie a názov parametra získame z tokenu metódou IMetaDataImport::GetParamProps. Teraz sa pozrieme na to, ako získať hodnoty jednotlivých parametrov. Štruktúra COR_PRF_FUNCTION_ARGUMENT_INFO predstavuje hodnoty argumentov funkcie.

```
typedef struct _COR_PRF_FUNCTION_ARGUMENT_INFO {
```

```
                ULONG numRanges;
                ULONG totalArgumentSize;
                COR_PRF_FUNCTION_ARGUMENT_RANGE ranges[];
} COR_PRF_FUNCTION_ARGUMENT_INFO;
```

Táto štruktúra obsahuje pole štruktúr _COR_PRF_FUNCTION_ARGUMENT_RANGE. Ich počet v poli je numRanges a celková veľkosť hodnôt argumentov v štruktúrach je totalArgumentSize [10]. Každá štruktúra _COR_PRF_FUNCTION_ARGUMENT_RANGE obsahuje hodnotu jedného argumentu funkcie. Upozorňujeme, že v závislosti od volacej konvencie funkcie môže byť v poli na začiatku štruktúra, ktorá obsahuje adresu, kam ukazuje smerník this (nultý parameter) a v tom prípade štruktúra s hodnotou prvého argumentu je až na druhom mieste (index 1 v poli ranges).

```
typedef struct _COR_PRF_FUNCTION_ARGUMENT_RANGE {
                UINT_PTR startAddress;
                ULONG length;
} COR_PRF_FUNCTION_ARGUMENT_RANGE;
```

Premenná startAddress je adresa v pamäti, kde sa nachádza hodnota daného argumentu funkcie a length je jeho veľkosť. Ak je typ očakávanej hodnoty jednoduchý ako int, char alebo boolean, v pamäti je priamo ich hodnota. Ak je očakávaný typ hodnoty String, startAddress je adresa, kde sa nachádza ObjectId reprezentujúce daný string. Aby sme sa dostali k samotnému stringu, potrebujeme zistiť, na akom offsete od Id sa string nachádza. Na to slúži funkcia ICorProfilerInfo2::GetStringLayout.

```
HRESULT GetStringLayout(
[out] ULONG *pBufferLengthOffset, // dĺžka buffera
[out] ULONG *pStringLengthOffset, // dĺžka stringu
[out] ULONG *pBufferOffset); // offset na string
```

Adresu, kde sa nachádza už konkrétny string dostaneme nasledovne:

```
char* stringAddr= startAddress+*pBufferOffset
```

Podľa dokumentácie, string môže byť ukončený nulou, ale nemusí, preto je potrebné brať do úvahy dĺžku stringu. Zdôrazňujeme, že všetky textové reťazce obdržané prostredníctvom rozhraní IMetaDataImport, ICorProfilerInfo2 a ICorProfilerCallback2 sú v kódovaní UNICODE. Ak je typ objektu pole, na adrese startAddress sa nachádza ObjectId pre dané pole. Informácie o poli získame zavolaním funkcie ICorProfilerInfo2::GetArrayObjectInfo.

```
HRESULT GetArrayObjectInfo(
[in] ObjectID objectId,
[in] ULONG32 cDimensions,
[out] ULONG32 pDimensionSizes[],
```

```
[out] int pDimensionLowerBounds[],
[out] BYTE **ppData);
```

ObjectId je Id objektu reprezentujúce pole, o ktorom chceme získať informácie. Počet rozmerov poľa je cDimensions a pole pDimensionSizes obsahuje veľkosti jednotlivých rozmerov poľa. Pole pDimensionLowerBounds obsahuje dolné indexy jednotlivých rozmerov poľa a ppData je smerník na adresu, kde sa nachádza priamo pole s hodnotami. Podobným spôsobom možno získať informácie o triede a jednotlivých členoch triedy funkciou ICorProfilerInfo2::GetClassLayout.

Pri návrate z funkcie je volaný callback FunctionLeave2.

```
void __stdcall FunctionLeave2(
[in] FunctionID funcId,
[in] UINT_PTR clientData,
[in] COR_PRF_FRAME_INFO frameInfo,
[in] COR_PRF_FUNCTION_ARGUMENT_RANGE *retvalRange)
```

FuncId predstavuje id funkcie. Vo funkcii FunctionEnter2 je žiadúce uložiť získané informácie o funkcii do vhodnej dátovej štruktúry a následne k týmto informáciám pristupovať prostredníctvom id pri návrate z funkcie. V našej implementácii používame dátovú štruktúru std::multimap, v ktorej uchovávame smerník na štruktúru obsahujúcu informácie o funkcii a ako kľúč je použité id funkcie. Táto štruktúra môže obsahovať viacero záznamov s rovnakým kľúčom, keďže id konkrétnej funkcie sa nemení a funkcia môže rekurzívne volať samu seba. Záznam o konkrétnej funkcii odstraňujeme až pri návrate z funkcie a pri rekurzívnom volaní je pred návratom volaná znova funkcia s rovnakým id, o ktorej potrebujeme uložiť informácie, aby sme ich pri návrate mohli použiť.

Štruktúra, kam ukazuje smerník retvalRange obsahuje návratovú hodnotu funkcie, ktorú získame obdobným spôsobom, ako bolo popísané vyššie.

Pri načítaní assembly do pamäte je volaný callback AssemblyLoadFinished.

```
HRESULT AssemblyLoadFinished(
[in] AssemblyID assemblyId,
[in] HRESULT    hrStatus);
```

HrStatus hovorí o tom, či načítanie bolo úspešné alebo nie. Názov assembly možno získať použitím funkcie ICorProfilerInfo:2:GetAssemblyInfo.

Kapitola 3

Implementácia

V tejto kapitole detailnejšie popíšeme niektoré časti implementácie profilovacieho nástroja, ktorý je súčasťou tejto práce. Pri implementácii sme informácie čerpali z dokumentácie Microsoftu, z webu codeproject a zdrojových kódov nástroja WinapiOverride [16, 2, 4, 17].

3.1 COM

Ako už bolo spomenuté v časti 2.2, profiler je COM DLL server. Aplikácia bola vytvorená prostredníctvom softvéru Microsoft Visual Studio. Pri implementácii sme použili šablónu ATL (Active Template Library). Šablóna ATL je skupina c++ tried umožňujúca zjednodušenie programovania COM objektov. Použitím šablóny odpadá potreba ručného implementovania COM častí ako implementácia rozhrania IUnknown, nutnosť vytvárania IDL špecifikujúceho implementované rozhrania, alebo generovanie GUID pre jednotlivé coclass triedy. Taktiež netreba implementovať DllMain a metódy zabezpečujúce pridanie a odobranie informácií o knižnici z registrov. CoClass trieda v našom projekte sa nazýva CProfiler a dedí od troch tried.

```
class ATL_NO_VTABLE CProfiler :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CProfiler, &CLSID_Profiler>,
public CCorProfilerCallbackImpl{
public:
  BEGIN_COM_MAP(CProfiler)
  COM_INTERFACE_ENTRY(ICorProfilerCallback)
  COM_INTERFACE_ENTRY(ICorProfilerCallback2)
  END_COM_MAP()
  STDMETHOD(Initialize)(IUnknown *pICorProfilerInfoUnk);
  STDMETHOD(Shutdown)();
  STDMETHOD(AssemblyLoadFinished)(AssemblyID assemblyID,HRESULT
      hrStatus);}
```

Trieda CComCoClass poskytuje metódy na vytvorenie inštancie triedy CProfiler a získanie informácií o triede. Udržiavanie počtu referencií na objekt zabezpečuje trieda CComObjectRootEx. Tieto triedy sú súčasťou ATL. Abstraktná trieda CCorProfilerCallbackImpl obsahuje implementácie funkcií rozhrania ICorProfilerCallback2 s prázdnym telom. Rozhrania, ktoré implementuje CoClass trieda CProfiler, sú špecifikované prostredníctvom makra COM_MAP. Pri definíciách funkcií implementujúcich rozhrania, ktoré poskytuje COM objekt sa používa makro STDMETHOD, pri funkciách, ktoré vracajú HRESULT. Jeho definícia je nasledovná:

```
#define STDMETHOD(method) virtual HRESULT STDMETHODCALLTYPE method
```

Kompilátor preloží STDMETHODCALLTYPE na direktívu __stdcall. Pri implementácii funkcií sa používa makro STDMETHODIMP, ktoré je definované takto:

```
#define STDMETHODIMP HRESULT __stdcall
```

3.2 Antidebug

Napriek tomu, že je možné jednoducho sledovať volania v .NET aplikáciách, so škodlivými aplikáciami môže byť problém. Autori sa snažia o znemožnenie analýzy anti-debug trikmi. Jedným so spôsobov, ako sa aplikácia môže brániť pred profilovaním, je pozrieť sa na premenné prostredia, a v prípade existencie premenných potrebných na profilovanie popísaných v časti 2.2.1 ukončiť svoju činnosť. Tento trik jednoducho obchádzame v implementácií tak, že v metóde Initialize zrušíme tieto premenné prostredia, keďže už nie sú potrebné, pretože profiler je už načítaný v pamäti.

3.3 Filtrovanie udalostí

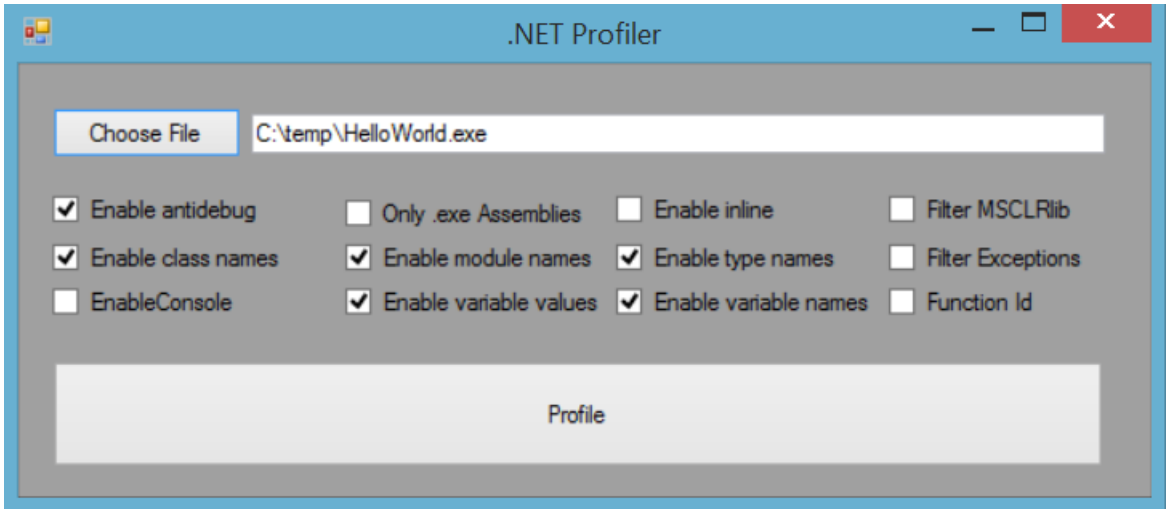
Profiler je informovaný o zavolaní každej funkcii vrámci sledovaného procesu. Väčšina funkcií však interne volá ďalšie funkcie a tie ďalšie.

Napríklad funkcia System.Console:ReadLine volá funkcie pre nastavenie vstupno-výstupných operácií a prípravu buffera a mnoho ďalších. Takéto interne volané funkcie však pre nás nie sú zaujímavé, a preto je dôležité vhodne filtrovať logovanie vyvolaných funkcií. Navyše, profilovanie výrazne spomaľuje vykonávanie aplikácie a čím rýchlejšie sa profiler vráti z funkcie FunctionEnter2, tým rýchlejšie bude profilovanie prebiehať. Vhodný spôsob, ako možno filtrovať funkcie, je na základe toho, kto funkciu zavolal. Stačí uviesť kritérium, že logované budú len funkcie zavolané z hlavného modulu (profilovaná aplikácia). V takom prípade sa nestane, že by sa v logu nachádzali záznamy o volaní interných funkcií ako v prípade funkcie System.Console:ReadLine. Profilovacie rozhrania však neposkytujú možnosť priamo zistiť informácie o volajúcom. Tento problém riešime v implementácii tak, že si ukladáme do dátovej štruktúry std::stack

postupne volajúcich (názov modulu, v ktorom je funkcia volaná vieme získať) a na základe toho buď logujeme informácie o funkcii, alebo nie.

3.4 Aplikácia

Profilovací nástroj, ktorý sme implementovali, pozostáva z dvoch častí. Prvá časť je GUI aplikácia napísaná v jazyku C#. Rozhranie aplikácie je zobrazené na obrázku 3.1. Táto aplikácia umožňuje užívateľovi vybrať program, ktorý má byť profilovaný a špecifikovať filtrovanie logovania volaných funkcií. Aplikácia spustí program s nastavenými profilovacími premennými prostredia tak, aby CLR načítalo profiler. Druhá časť je samotný profiler, COM DLL knižnica napísaná v jazyku C++.



Obr. 3.1: GUI časť profilovacieho nástroja.

Záver

V práci sme stručne predstavili platformu .NET a vysvetlili, akým spôsobom funguje. Popísali sme štruktúru spustiteľného .NET súboru a formu, akou uchováva informácie o programovom kóde, ktoré možno využiť pri získavaní informácií o bežiacom programe a popísali sme možnosti, ako získať názvy vyvolaných funkcií, hodnoty ich argumentov a návratových hodnôt v bežiacej .NET aplikácii. Implementovali sme jednoduchý nástroj na profilovanie, ktorý je schopný získať hodnoty niektorých základných typov, a ktorý bude používaný pri analýzach škodlivých .NET aplikácií.

Literatúra

[1] Kevin Burton. *.NET Common Language Runtime Unleashed*. Sams Publishing, 1st edition, 2002.

[2] Michael Dunn. Introduction to COM Part II - Behind the Scenes of a COM Server. Získané 26.4.2016 z `http://www.codeproject.com/Articles/901/Introduction-to-COM-Part-II-Behind-the-Scenes-of-a`.

[3] ECMA. Common Language Infrastructure (CLI). Získané 27.1.2016 z `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf`.

[4] Scott Hackett. Creating a Custom .NET Profiler, 2006. Získané 5.12.2015 z `http://www.codeproject.com/Articles/15410/Creating-a-Custom-NET-Profiler`.

[5] Marcin Hartung. UNPACK YOUR TROUBLES: .NET PACKER TRICKS AND COUNTERMEASURES. *Virus Bulletin*, pages 142–148, 9 2015.

[6] microfocus. COM Object Processing. Získané 29.4.2016 z `https://supportline.microfocus.com/documentation/books/nx40/dicwiz.htm`.

[7] Microsoft. Common HRESULT Values. Získané z 10.5.2016 na `https://msdn.microsoft.com/en-us/library/windows/desktop/aa378137(v=vs.85).aspx`.

[8] Microsoft. Component Object Model. Získané 25.1.2016 z `https://msdn.microsoft.com/en-us/library/ee663262(v=vs.85).aspx`.

[9] Microsoft. Compressed Integer In .NET/CLI Metadata. Získané 20.4.2016 z `http://www.cnblogs.com/AndersLiu/archive/2010/02/09/en-compressed-integer-in-metadata.html`.

[10] Microsoft. COR_PRF_FUNCTION_ARGUMENT_INFO Structure. Získané 20.4.2016 z `https://msdn.microsoft.com/en-us/library/ms404379(v=vs.110).aspx`.

[11] Microsoft. Metadata and Self-Describing Components. Získané 29.4.2016 z `https://msdn.microsoft.com/en-us/library/xcd8txaw(v=vs.110).aspx`.

[12] Microsoft. .NET malware dynamic instrumentation. Získané 27.4.2016 z `https://www.virusbulletin.com/uploads/pdf/conference_slides/2014/Hu-VB2014.pdf`.

[13] Microsoft. Profiler Compatibility Settings. Získané 20.4.2016 z `https://msdn.microsoft.com/en-us/library/dd778910(v=vs.100).aspx`.

[14] Microsoft. Metadata (Unmanaged API Reference), 2015. Získané 15.3.2015 z `https://msdn.microsoft.com/en-us/library/ms404384(v=vs.110).aspx`.

[15] Microsoft. Profiling Interfaces, 2015. Získané 5.12.2015 z `https://msdn.microsoft.com/en-US/library/ms404511(v=vs.110).aspx`.

[16] Microsoft. Profiling (Unmanaged API Reference), 2015. Získané 5.12.2015 z `https://msdn.microsoft.com/en-us/library/ms404386.aspx`.

[17] Jacquelin Potier. WinAPIOverride - Free Advanced API Monitor, spy or override API or exe internal functions, 2015. Získané 5.12.2015 z `http://jacquelin.potier.free.fr/winapioverride32/`.

[18] Wikipedia. Common Language Runtime. Získané 3.2.2016 z `https://en.wikipedia.org/wiki/Common_Language_Runtime`.