EDIZIONI LSWR

C# per hacker

Creare e automatizzare strumenti di sicurezza per Windows, Linux e macOS

Brandon Perry

Prefazione di Matt Graeber

no starch press

*pro
Digital Life Style

C#
per hacker

Creare e automatizzare strumenti di sicurezza per Windows, Linux e macOS

Brandon Perry

EDIZIONI
LSWR

Titolo originale: *Gray Hat C# | A Hacker's Guide to Creating and Automating Security Tools*
ISBN: 978-1-59327-759-8
Published by No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
www.nostarch.com

Edizione italiana:
C# per hacker | Creare e automatizzare strumenti di sicurezza per Windows, Linux e macOS

Traduzione di: Antonio Pelleriti
Publisher: Marco Aleotti
Immagine di copertina: © Sanja Kaz e © Kobsoft |Shutterstock

Collana: *pro DigitalLifeStyle

ISBN: 978-88-6895-589-2

EDIZIONI LSWR

Via G. Spadolini, 7
20141 Milano (MI)
Tel. 02 881841
www.edizionilswr.it

Printed in Italy

Finito di stampare nel mese di novembre 2017 presso "Rotolito Lombarda" S.p.A., Seggiano di Pioltello (MI) Italy

(*) Edizioni Lswr è un marchio di La Tribuna Srl. La Tribuna Srl fa parte di LSWR GROUP.

Sommario

Prefazione

Nel ruolo di "attaccante" o "difensore" che sviluppa software, bisogna ovviamente decidere che linguaggio ha più senso utilizzare. Idealmente, la scelta non ricadrà su un linguaggio semplicemente perché è quello su cui lo sviluppatore si trova più a suo agio. Piuttosto, tale scelta sarà dettata dalle risposte a una serie di domande come le seguenti.

- Quali sono i principali ambienti bersaglio di esecuzione?
- Qual è lo stato delle librerie di rilevazione e logging per i payload scritte in tale linguaggio?
- A che livello il mio software deve rimanere invisibile (per esempio residente in memoria)?
- Quanto è ben supportato il linguaggio sia lato client che lato server?
- C'è una grande comunità di sviluppatori in questo linguaggio?
- Qual è la curva di apprendimento e quanto è mantenibile il linguaggio?

C# ha diverse risposte convincenti a queste domande. Per quanto riguarda la domanda sull'ambiente bersaglio di esecuzione, .NET dovrebbe essere un candidato ovvio da considerare in ambiente Microsoft, dato che da anni è incluso in Windows. In ogni caso con l'apertura all'open source di .NET, C# è ora un linguaggio che può contare su un ambiente di esecuzione maturo per ogni sistema operativo. Quindi dovrebbe essere considerato un linguaggio estremamente entusiasmante per il vero supporto cross-platform.

C# è sempre stato la lingua franca dei linguaggi .NET. Come vedrete in questo libro, sarete attivi con C# in men che non si dica, grazie alla sua bassa soglia di ingresso e alla massiccia comunità di sviluppatori. In più, essendo quelli .NET dei linguaggi gestiti e ricchi di tipi, gli assembly compilati si prestano a essere facilmente decompilati in C#. Pertanto, chi scrive programmi offensivi in C#, non dovrà necessariamente sviluppare le proprie capacità dal nulla. Anzi, potrà attingere da un ricco insieme di esempi di malware .NET, decompilarli, leggere il loro codice sorgente, e "prendere in prestito" le loro capacità.

Si potrebbe addirittura spingersi oltre e utilizzare le .NET Reflection API per caricare ed eseguire dinamicamente questi esempi esistenti di malware .NET, assumendo, naturalmente, di averli modificati abbastanza da non fargli fare nulla di sovversivo.
Nel ruolo di chi ha passato anni a rendere popolare PowerShell come strumento offensivo, i miei sforzi hanno portato a un massiccio miglioramento della sicurezza e degli strumenti di logging, in seguito all'aumento di malware PowerShell.
L'ultima versione di PowerShell (v5 nel momento in cui scrivo) implementa più funzioni di logging di qualsiasi altro linguaggio esistente. Dalla prospettiva di un difensore, tutto ciò è fantastico. Da quella di un pentester, di nemico, o avversario, ciò aumenta notevolmente il rumore di un attacco.
Ma per un libro su C#, perché menzionarlo? Anche se ci ho messo anni a riconoscerlo, più PowerShell scrivo, più mi rendo conto che gli aggressori stanno guadagnando molta più agilità sviluppando i loro strumenti in C# piuttosto che facendolo esclusivamente in PowerShell. Lasciatemi spiegare.

- .NET offre ricche API di reflection che consentono con facilità di caricare e interagire dinamicamente con un assembly C# compilato in memoria. Con tutti i controlli eseguiti ora sui payload PowerShell, l'API di reflection consente a un utente malintenzionato di nascondersi meglio ai radar, sviluppando un payload in PowerShell che funge solo da caricatore ed esecutore di un assembly .NET.
- Come ha dimostrato Casey Smith (@subTee), ci sono molti file binari legittimi firmati da Microsoft presenti in un'installazione predefinita di Windows, che fungono da fantastici processi host per dei payload C#: msbuild.exe è fra i più invisibili. Usare MSBuild come processo host per un malware C# incarna perfettamente la metodologia "vivere di ciò che offre la terra": l'idea che gli attaccanti possano fondersi in un ambiente bersaglio e lasciare impronte minime prospererà per un lungo periodo di tempo.
- I produttori di antimalware, a oggi, sono per la maggior parte ancora ignari delle funzionalità degli assembly .NET a runtime. C'è ancora abbastanza codice malware non gestito là fuori, e l'attenzione non si è ancora spostata sul rilevare in modo efficace il .NET Runtime che sta eseguendo un'introspezione dinamica a runtime.
- Con l'accesso alla massiccia libreria di classi .NET, chi si trova a suo agio con PowerShell troverà il passaggio a C# relativamente semplice. Viceversa, chi conosce C# incontrerà una barriera d'ingresso più bassa nel trasferire le proprie competenze ad altri linguaggi .NET come PowerShell e F#.
- Come PowerShell, C# è un linguaggio di alto livello, il che significa che gli sviluppatori non devono preoccuparsi di codifica a basso livello e di gestione della memoria. A volte, tuttavia, si ha la necessità di andare a "basso livello" (per

esempio, interagendo con le API Win32). Fortunatamente, tramite le sue API di Reflection, P/Invoke e l'interfaccia di marshaling, C# consente di ottenere il basso livello necessario.

Ognuno ha motivazioni differenti per imparare C#. La mia motivazione era il bisogno di trasferire le mie conoscenze di PowerShell in maniera da riuscire a scrivere codice .NET su piattaforme differenti.
Tu, lettore, potresti essere stato attratto da questo libro per acquisire la mentalità da attaccante, e completare le tue conoscenze di C#. Al contrario, potresti applicare la tua mentalità da aggressore a un linguaggio supportato da tante piattaforme differenti. Qualunque sia la tua motivazione, tieniti pronto a un giro selvaggio attraverso la mente di Brandon, mentre impartisce la sua esperienza unica e la sua conoscenza nello sviluppo di C# offensivo e difensivo.

Matt Graeber
Microsoft MVP

Introduzione

Tante volte mi viene chiesto perché mi piace così tanto C#. Essendo io un sostenitore del software open source, un utente Linux e un contributore di Metasploit (scritto prevalentemente in Ruby), C# sembra una scelta strana come linguaggio preferito.
Quando tanti anni fa iniziai a scrivere in C#, Miguel de Icaza (noto per GNOME) aveva recentemente avviato un piccolo progetto denominato Mono. Mono è essenzialmente un'implementazione open source del Microsoft .NET Framework. C# come linguaggio è stato sottoposto all'ECMA come standard, e il .NET Framework è stato pubblicizzato da Microsoft come sostituto di Java dato che il codice può essere compilato su un sistema o piattaforma ed eseguito su un altro. L'unico problema è che Microsoft ha rilasciato il .NET Framework solo per il sistema operativo Windows. Miguel e un piccolo gruppo di collaboratori si sono impegnati a rendere il progetto Mono il ponte fra .NET e la comunità Linux. Fortunatamente un mio amico, che mi aveva raccomandato di imparare C# ma che sapeva anche del mio interesse per Linux, mi ha raccontato di questo progetto nascente per vedere se fosse possibile utilizzare sia C# che Linux. A quel punto ero incastrato.
C# è un linguaggio bellissimo. Il creatore e architetto principale del linguaggio, Anders Hejlsberg, ha iniziato a lavorare sui compilatori per Pascal prima e Delphi in seguito. Questa esperienza gli ha fornito una conoscenza profonda di caratteristiche veramente potenti in diversi linguaggi di programmazione. Hejlsberg è quindi entrato in Microsoft e C# è nato intorno all'anno 2000. Nei suoi primi anni, C# ha condiviso un sacco di caratteristiche con il linguaggio Java, come la sintassi, ma con il passare del tempo è cresciuto per conto proprio e ha introdotto tutta una serie di caratteristiche precedentemente appartenenti a Java, come LINQ, i delegati e i metodi anonimi.
Con C#, si dispone di molte delle potenti funzionalità di C e C++ e si possono scrivere applicazioni web complete utilizzando lo stack ASP.NET o ricche applicazioni desktop. Su Windows, la libreria utilizzata per la UI è WinForms, ma su Linux sono facili da usare le librerie GTK e QT. Più recentemente, Mono ha introdotto il supporto per Cocoa Toolkit su piattaforma macOS. Anche iPhone e Android sono supportati.

Perché mi devo fidare di Mono?

I detrattori del progetto Mono e del linguaggio C# sostengono che non è sicuro utilizzare tali tecnologie su piattaforme diverse da Windows. La loro convinzione che Microsoft, in un batter d'occhio, inizierà un contenzioso con Mono facendolo cadere nell'oblio porta molte persone a non prendere sul serio il progetto. Non credo sia un rischio plausibile.

Nel momento in cui scrivo, non solo Microsoft ha acquisito Xamarin, la compagnia che Miguel de Icaza ha fondato per supportare il framework Mono, ma ha anche reso open source grandi porzioni del .NET Framework. Ha abbracciato il software open source in maniera inimmaginabile sotto la guida di Steve Ballmer. Il nuovo direttore generale, Satya Nadella, ha dimostrato che Microsoft non ha alcun problema con il software open source e l'azienda è attivamente impegnata con la comunità Mono per consentire lo sviluppo mobile utilizzando tecnologie Microsoft.

A chi si rivolge questo libro?

Molta gente che lavora in ambito sicurezza, per esempio ingegneri di rete e specialisti di sicurezza informatica, si affida all'automazione in maniera più o meno estesa, sia per la scansione delle vulnerabilità che per l'analisi di malware. Dato che tanti di questi professionisti utilizzano una vasta varietà di sistemi operativi, scrivere strumenti che tutti possano eseguire facilmente può essere complicato.

Mono è un'ottima scelta perché è cross-platform e ha un eccellente insieme di librerie di base con cui automatizzare in maniera semplice molti aspetti del lavoro di un esperto della sicurezza.

Qualora foste interessati a scrivere exploit, automatizzare la scansione delle vulnerabilità di un'infrastruttura, decompilare altre applicazioni .NET, leggere offline gli hive di registro, o creare payload personalizzati multi-piattaforma, allora molti degli argomenti trattati nel libro vi permetteranno di iniziare a farlo (anche se non avete nessuna conoscenza di C#).

Organizzazione del libro

In questo libro copriremo le basi di C# e implementeremo rapidamente strumenti di sicurezza reali grazie alle numerose librerie disponibili per il linguaggio. Fin dall'inizio, scriveremo dei fuzzer per individuare possibili vulnerabilità e gli exploit per sfruttare quelle trovate. Dovrebbe risultare molto evidente quanto siano potenti le caratteristiche del linguaggio e delle librerie di base. Una volta coperti i fondamentali, automatizzeremo gli strumenti di sicurezza più diffusi come Nessus, sqlmap, Cuckoo Sandbox. Nel

complesso, una volta finito di leggere questo libro, avrai un eccellente repertorio di piccole librerie per automatizzare molti dei lavoretti eseguiti spesso dai professionisti.

Capitolo 1: Corso Intensivo di C#: in questo capitolo impareremo le basi della programmazione orientata agli oggetti in C# con piccoli esempi, ma affronteremo un'ampia varietà delle funzionalità del linguaggio. Cominceremo con un programma Hello World costruendo poi piccole classi per comprendere meglio cosa sia la programmazione orientata agli oggetti. A questo punto ci sposteremo verso le caratteristiche più avanzate di C#, come i metodi anonimi e P/Invoke.
Capitolo 2: Fuzzing ed exploiting con XSS e SQL Injection: in questo capitolo scriveremo dei fuzzer costituiti da piccole richieste HTTP nel tentare XSS e SQL injection con diversi tipi di dati e utilizzando librerie HTTP per comunicare con i server web.
Capitolo 3: Fuzzing di endpoint SOAP: in questo capitolo si porteranno i concetti del precedente capitolo a un livello superiore, scrivendo un piccolo fuzzer che ricava e analizza un SOAP WSDL per trovare potenziali SQL injection, generando automaticamente richieste HTTP. Tutto ciò mentre esamineremo anche le eccellenti classi XML disponibili nella libreria di base.
Capitolo 4: Scrivere payload di Connect-Back, Binding e Metasploit: in questo capitolo si passa da HTTP alla creazione di payload. In primo luogo creeremo un paio di semplici payload, uno su TCP e uno su UDP. Quindi si vedrà come generare shellcode X86/x86_64 in Metasploit per creare payload multipiattaforma e multiarchitettura.
Capitolo 5: Automazione di Nessus: in questo capitolo si torna ad HTTP per automatizzare il primo di diversi scanner di vulnerabilità, Nessus. Vedremo come creare, osservare e ottenere report dalla scansione di range CIDR in maniera programmatica.
Capitolo 6: Automazione di Nexpose: in questo capitolo si mantiene il focus sull'automazione di tool, spostandosi allo scanner di vulnerabilità Nexpose. Nexpose, le cui API sono anch'esse basate su HTTP, può essere automatizzato per eseguire la scansione di vulnerabilità e creare report. Rapid7, creatore di Nexpose, offre una licenza annuale gratuita per la versione community, molto utile per gli appassionati non aziendali.
Capitolo 7: Automazione di OpenVAS: in questo capitolo concludiamo l'analisi sull'automazione degli scanner di vulnerabilità con OpenVAS, che è open source. Le API di OpenVAS sono fondamentalmente differenti da quelle di Nexus e di Nexpose, in quanto utilizzano solo socket TCP e XML per il protocollo di comunicazione. Essendo gratuito, inoltre, è utile per hobbisti con poche disponibilità economiche che cercano di acquisire maggiore esperienza nella scansione di vulnerabilità.
Capitolo 8: Automazione di Cucko Sandbox: in questo capitolo si passerà all'informatica forense con Cucko Sandbox. Lavorando con delle semplici API REST JSON, automatizzeremo l'analisi di potenziali esempi di Malware e creeremo dei report sui risultati.

Capitolo 9: Automazione di sqlmap: in questo capitolo cominciamo a sfruttare le SQL injection in maniera più ampia, automatizzando sqlmap. Per prima cosa creeremo piccoli strumenti per aprire singoli URL con le semplici API JSON fornite da sqlmap. Una volta presa confidenza, li integreremo nel fuzzer SOAP WSDL del capitolo 3, in modo che qualsiasi potenziale vulnerabilità di SQL injection possa essere sfruttata e validata automaticamente.
Capitolo 10: Automazione di ClamAV: in questo capitolo ci concentriamo sull'interazione con le librerie native, non gestite, di ClamAV, un popolare progetto open source di antivirus. Non è scritto in un linguaggio .NET, ma è possibile interfacciarsi con le sue librerie di base e con il suo demone TCP, che ne consente l'utilizzo remoto. Vedremo come automatizzare ClamAV in entrambi gli scenari.
Capitolo 11: Automazione di Metasploit: in questo capitolo torniamo a Metasploit, in modo da imparare a utilizzarlo in maniera programmatica, per sfruttare e segnalare gli host scoperti tramite le librerie MSGPACK RPC incluse nel framework.
Capitolo 12: Automazione di Arachni: in questo capitolo vedremo come automatizzare lo scanner Arachni, un'applicazione web gratuita e open source, anche se con doppia licenza. Utilizzando sia le più semplici API REST HTTP che le più potenti MSGPACK RPC fornite dal progetto, creeremo piccoli strumenti per automatizzare la creazione di report sui risultati ottenuti analizzando un URL.
Capitolo 13: Decompilazione di Assembly .NET: in questo capitolo ci spostiamo al reverse engineering. Esistono decompilatori facili da usare per Windows, ma non per Mac e Linux, quindi ne scriveremo uno noi stessi.
Capitolo 14: Lettura offline del registro: in questo capitolo passiamo all'incident response e ci interesseremo degli hive di registro analizzando la struttura binaria del registro di Windows. Vedremo come analizzare e leggere offline gli hive, in maniera da ricavare la chiave di boot del sistema, utilizzata per criptare le password memorizzate nel registro stesso.

Ringraziamenti

La scrittura di questo libro ha richiesto un decennio, anche se è stato digitato in un elaboratore testi solo per tre anni. La mia famiglia e i miei amici hanno sicuramente notato che ho parlato continuamente di C#, ma sono stati ascoltatori più che comprensivi. Grazie ai fratelli e alle sorelle AHA che hanno ispirato molti dei progetti del libro. Tanti ringraziamenti a John Eldridge, un amico di famiglia che mi ha fatto conoscere C# e mi ha letteralmente iniziato alla programmazione. Brian Rogers è stato una delle migliori risorse tecniche con cui confrontare le idee durante lo sviluppo del libro, così come un eccellente editore tecnico con il suo attento occhio e le sue intuizioni. I miei dirigenti

di produzione Serena Yang e Alison Law hanno fatto avanti e indietro nel processo di editing in maniera più indolore possibile. E, naturalmente, Bill Pollock e Jan Cash sono stati in grado di scolpire le mie oscure parole in frasi chiare che chiunque potrebbe leggere. Un grazie enorme a tutto lo staff No Starch!

Una nota finale

Ognuno di questi capitoli scalfisce soltanto la superficie della potenza di C#, così come il potenziale degli strumenti che automatizzeremo e svilupperemo, specialmente perché molte delle librerie create devono essere flessibili ed estensibili. Spero che questo libro mostri quanto sia facile automatizzare compiti comuni e noiosi e ispiri a continuare lo sviluppo degli strumenti iniziati. Troverete il codice sorgente e aggiornamenti al libro, in inglese, all'indirizzo https://www.nostarch.com/ grayhatcsharp/.

1

Corso intensivo di C#

C# è un linguaggio di programmazione multipiattaforma e basato sul paradigma orientato agli oggetti. Grazie all'implementazione di Mono, è possibile eseguire applicazioni C# su sistemi operativi differenti, per esempio Windows e Linux.

A differenza di altri linguaggi, come Ruby, Python e Perl, i programmi C# possono essere eseguiti direttamente su tutti i moderni sistemi Windows. Inoltre, eseguire applicazioni C# su un sistema Linux come Ubuntu, Fedora o altro non potrebbe essere più semplice, specialmente da quando Mono può essere rapidamente installato dalla maggior parte dei gestori di pacchetti Linux, come `apt` o `yum`. Tutto ciò pone C# in una miglior posizione per venire incontro alle necessità multi-piattaforma rispetto alla maggior parte dei linguaggi, con il beneficio di una libreria di base semplice e potente a portata di mano. Nel complesso, C# e le librerie Mono/.NET costituiscono un framework interessante per chiunque voglia scrivere strumenti cross-platform in modo semplice e rapido.

Scegliere un IDE

La maggior parte di coloro che vogliono imparare C# utilizzeranno un ambiente di sviluppo integrato (IDE) come Visual Studio per scrivere e compilare il proprio codice. Visual Studio di Microsoft è lo standard *de facto* per lo sviluppo C# in tutto il mondo. La versione gratuita Visual Studio Community Edition è disponibile per utilizzo personale e può essere scaricata dal sito Microsoft all'url https://www.visualstudio.com/downloads/.

Durante la scrittura di questo libro, ho utilizzato MonoDevelop e Xamarin Studio a seconda che mi trovassi su Ubuntu o macOS, rispettivamente. Su Ubuntu, potete facilmente installare MonoDevelop utilizzando il gestore di pacchetti `apt`. MonoDevelop è sviluppato da Xamarin, la stessa azienda che mantiene Mono. Per installarlo, utilizzate il seguente comando:

```
$ sudo apt-get install monodevelop
```

Xamarin Studio è la versione macOS dell'IDE MonoDevelop. Xamarin Studio e MonoDevelop hanno le stesse funzionalità, ma interfacce utente leggermente differenti. L'installer di Xamarin Studio può essere scaricato dal sito Xamarin all'url https://www.xamarin.com/download-it/.
Uno di questi tre IDE soddisferà i nostri bisogni in questo libro. Infatti, se volete utilizzare il solo `vim`, non avrete neanche bisogno di un IDE! Vedremo a breve anche come compilare un semplice esempio utilizzando il compilatore C# a riga di comando rilasciato con Mono, anziché un IDE.

Un semplice esempio

A chiunque abbia utilizzato C o Java, la sintassi C# apparirà molto familiare. C# è un linguaggio fortemente tipizzato, come C e Java; ciò significa che una variabile dichiarata nel codice può essere di un solo tipo (un intero, una stringa), o di una classe (Dog, per esempio) e sarà sempre di questo tipo, in ogni caso. Iniziamo dando una rapida occhiata all'esempio Hello World nel Listato 1.1, che mostra un po' di tipi e sintassi C#.

```
using ❶ System;
namespace ❷ ch1_hello_world
{
   class ❸ MainClass
   {
      public static void ❹ Main(string[] ❺ args)
      {
      ❻ string hello = "Hello World!";
      ❼ DateTime now = DateTime.Now;
      ❽ Console.Write(hello);
      ❾ Console.WriteLine(" The date is " + now.ToLongDateString());
      }
   }
}
```

Listato 1.1 – Una semplice applicazione Hello World.

Per prima cosa, è necessario importare i namespace da utilizzare, e ciò avviene con l'istruzione `using` che importa il namespace `System` ❶. In tal modo viene abilitato l'ac-

cesso alle librerie nel programma, come avviene con `#include` in C, `import` in Java e Python, e `require` in Ruby e Perl. Dopo aver dichiarato la libreria che si vuol utilizzare, viene dichiarato il namespace ❷ di cui faranno parte le classi.

A differenza di C (e delle versioni più vecchie di Perl), C# è un linguaggio orientato agli oggetti, simile a Ruby, Python e Java. Ciò significa che possiamo costruire classi complesse per rappresentare strutture dati, assieme ai metodi per tali strutture dati, mentre scriviamo codice. I namespace permettono di organizzare le nostre classi e il nostro codice e di evitare potenziali conflitti di nomi, per esempio come avverrebbe se due programmatori creassero due classi con lo stesso nome. Se due classi con lo stesso nome sono in namespace differenti, non ci sarà nessun problema. Ogni classe deve necessariamente avere un namespace.

Ora che c'è un namespace possiamo dichiarare una classe ❸ che conterrà il metodo `Main()` ❹. Come detto in precedenza, le classi consentono di creare tipi di dati complessi e strutture dati che rappresentino meglio oggetti reali. In questo esempio, il nome della classe non ha importanza; serve solo da contenitore del metodo `Main()`, che è quello che realmente serve, perché è il metodo `Main()` che sarà eseguito all'avvio dell'applicazione. Ogni applicazione C# necessita di un metodo `Main()`, come in C e Java. Se l'applicazione accetta argomenti da linea di comando, si può utilizzare la variabile `args` ❺ per accedere agli argomenti passati.

In C# esistono strutture dati semplici, come le stringhe ❻, così come possono essere creati oggetti più complessi, per esempio una classe che rappresenta data e orario ❼. La classe `DateTime` è una delle classi base di C# e serve a lavorare con le date. Nell'esempio viene utilizzata per memorizzare la data e l'orario attuale (`DateTime.Now`) nella variabile `now`. Infine, dichiarate le variabili, è possibile stampare un messaggio utilizzando i metodi `Write()` ❽ e `WriteLine()` ❾ della classe `Console` (il secondo include un carattere per andare a capo alla fine del testo).

Se si utilizza un IDE, è possibile compilare ed eseguire il codice facendo clic sul pulsante Esegui, che si trova in genere in alto a sinistra e assomiglia a un pulsante Play, oppure premendo il tasto F5. In ogni caso, se si vuole è possibile anche compilare il sorgente da linea di comando con il compilatore Mono. Nella directory con il file contenente la classe C#, basta utilizzare il comando `mcs` rilasciato con Mono per compilare un eseguibile come segue:

```
$ mcs Main.cs -out:ch1_hello_world.exe
```

Eseguendo il codice del Listato 1.1, dovrebbe essere stampata la stringa "Hello World!" e la data attuale sulla stessa riga, come mostrato nel Listato 1.2.

```
$ ./ch1_hello_world.exe
Hello World! The date is Wednesday, June 28, 2017
```

Listato 1.2 – Esecuzione dell'applicazione Hello World.

Congratulazioni per la tua prima applicazione C#!

Introduzione a classi e interfacce

Classi e interfacce sono utilizzate per creare strutture dati complesse, che sarebbero difficili da rappresentare utilizzando soltanto le strutture predefinite. Classi e interfacce possono avere *proprietà*, che sono variabili con cui leggerne e impostarne i valori, e *metodi*, che sono come funzioni eseguibili sulla classe (o sottoclassi) e sull'interfaccia, e appartengono solo a esse. Proprietà e metodi sono utilizzati per rappresentare i dati di un oggetto. Per esempio, una classe `Pompiere` potrebbe avere necessità di una proprietà `int` per rappresentare l'importo della pensione, o un metodo che indichi al pompiere di andare a spegnere un incendio in un certo luogo.
Le classi possono essere usate come modello per creare altre classi, secondo una tecnica chiamata *subclassing*. Quando una classe è sottoclasse di un'altra, la prima eredita le proprietà e i metodi della seconda (chiamata classe *madre*). Le interfacce sono anch'esse utilizzabili come modelli per creare nuove classi, ma a loro differenza non supportano l'ereditarietà. Quindi una classe base che implementa un'interfaccia non passa le proprietà e i metodi dell'interfaccia se da essa viene ereditata una sottoclasse.

Creazione di una classe

Creeremo ora la semplice classe mostrata nel Listato 1.3 come esempio, che rappresenta la struttura dati di un impiegato pubblico, cioè di qualcuno il cui lavoro quotidiano è quello di renderci la vita migliore.

```
public ❶ abstract class PublicServant
{
   public int ❷ PensionAmount { get; set; }
   public abstract void ❸ DriveToPlaceOfInterest();
}
```

Listato 1.3 – La classe astratta PublicServant.

La classe `PublicServant` è un tipo speciale di classe. È una classe *astratta* ❶. In generale si può istanziare una classe come si fa con ogni altro tipo di variabile, e in questo caso si parla di *istanza* o *oggetto*. Le classi astratte, tuttavia, non possono essere istanziate come le altre; esse possono essere solamente ereditate da una sottoclasse. Esistono tanti tipi di impiegati pubblici, pompieri e poliziotti sono i primi esempi che mi vengo-

no subito in mente. Avrebbe quindi senso avere una classe base da cui questi due tipi possano ereditare. In questo caso, se queste due classi fossero sottoclassi di `PublicServant`, erediterebbero la proprietà ❷ `PensionAmount` e un metodo ❸ `DriveToPlaceOfInterest` che dovrà essere implementato dalle sottoclassi di `PublicServant`. Non esiste un lavoro da "impiegato pubblico" generico, quindi non c'è un motivo per creare direttamente un'istanza della classe `PublicServant`.

Creazione di un'interfaccia

Le interfacce sono un complemento per le classi di C#. Le *interfacce* consentono a un programmatore di forzare una classe a implementare certi metodi o proprietà non ereditati. Vediamo come creare una semplice interfaccia, come mostrato nel Listato 1.4. Questa interfaccia è chiamata `IPerson` e dichiarerà un paio di proprietà che la gente generalmente possiede.

```
public interface ❶ IPerson
{
   string ❷ Name { get; set; }
   int ❸ Age { get; set; }
}
```

Listato 1.4 – L'interfaccia `IPerson`.

> **NOTA**
> I nomi delle interfacce in C# sono generalmente preceduti da una I per distinguerle dalle classi che le implementano. Questa I non è obbligatoria, ma è una pratica molto comune nello sviluppo C#.

Se una classe volesse implementare l'interfaccia `IPerson` ❶, tale classe dovrebbe implementare entrambe le proprietà `Name` ❷ e `Age` ❸. In caso contrario non compilerebbe. Mostreremo esattamente cosa significa scrivendo nel prossimo paragrafo la classe `Firefighter`, che implementa l'interfaccia `IPerson`. Per ora, tenete a mente che le interfacce sono una funzionalità importante e utile di C#. Programmatori familiari con le interfacce in Java si sentiranno certamente a loro agio. I programmatori C possono pensare a esse come a dei file di intestazione contenenti le dichiarazioni delle funzioni che si aspettano un file *.c* con la loro implementazione. Chi sviluppa in Perl, Ruby o Python potrebbe trovare strane le interfacce a un primo impatto, perché non esiste una funzionalità simile in questi linguaggi.