7 Безопасность Ajax-приложений

В этой главе...

- Модель безопасности
- Удаленные Web-службы
- Защита пользовательских данных, передаваемых по Интернету
- Защита потоков данных Ajax

Наличие средств защиты — важное свойство Интернет-служб. Система Web изначально не защищена, поэтому качество Ajax-приложения во многом зависит от наличия в нем средств обеспечения безопасности. Независимо от того, покупает ли пользователь товары в интерактивном магазине или приобретает право воспользоваться сетевыми услугами, он платит деньги, и этот факт еще больше повышает важность средств защиты.

Безопасность — обширная тема, которой посвящены многие книги. К средствам защиты Ajax во многом предъявляются те же требования, что и к системе защиты классического Web-приложения. Поэтому в данной главе мы будем уделять внимание только вопросам безопасности, специфическим для инфраструктуры Ajax. В первую очередь мы поговорим о передаче исполняемого кода по сети и мерах, которые производители браузеров принимают для того, чтобы сделать эту процедуру безопасной. Мы также обсудим шаги, которые можно предпринять для того, чтобы несколько смягчить меры предосторожности (естественно, в тех случаях, когда это допустимо). Затем мы рассмотрим вопросы защиты пользовательских данных, передаваемых серверу, и обеспечения конфиденциальности при работе с Ajax-приложениями. И наконец, уделим внимание защите служб, используемых Ajax-клиентами, и предотвращению несанкционированного доступа к ним. Начнем с вопросов передачи клиентского кода по сети.

7.1. JavaScript и защита браузера

При загрузке Ajax-приложения Web-сервер посылает браузеру набор JavaScript-команд, предназначенных для выполнения на удаленной машине, о которой сервер не имеет практически никаких сведений. Позволяя командам выполняться в среде браузера, пользователь тем самым высказывает доверие приложению и его авторам. Это доверие не всегда оправдано, поэтому производители браузеров предусматривают ряд мер, направленных на защиту системы и информации. В данном разделе мы рассмотрим эти меры и их влияние на работу программ. Мы также обсудим ситуации, в которых ограничения оказываются слишком строгими и которые желательно смягчить. Наибольший интерес у разработчиков Ajax-приложений вызывает возможность непосредственно обращаться к Web-службам независимых производителей.

Прежде чем приступать к подробному обсуждению данной темы, определим понятие мобильного кода. Любой фрагмент содержимого жесткого диска компьютера представляет собой двоичные данные. Несмотря на это, существует возможность отличить собственно данные от машинных инструкций, которые могут быть выполнены на компьютере. Обычные данные не выполняют никаких действий, по крайней мере до тех пор, пока они не будут обработаны некоторым процессом. В первых приложениях клиент/сервер клиентский код устанавливался на компьютере пользователя наравне с другими приложениями, и весь трафик, передаваемый по сети, представлял собой обычные данные. Однако в Ajax-приложениях JavaScript-код может быть выполнен. Помимо того что он предоставляет ряд интересных возможностей, которые не могут обеспечить обычные данные, он также может стать источ-

ником опасности для системы. Назовем код мобильным, если он хранится на одной машине и может быть передан по сети для выполнения на другом компьютере. Компьютер, принимающий мобильный код, должен решить, доверяет ли он источнику кода. Это решение становится еще более ответственным, если исполняемый код получен из общедоступной сети. Необходим аргументированный ответ на вопрос, к каким из системных ресурсов можно предоставить доступ мобильному коду.

7.1.1. Политика "сервера-источника"

Как было сказано ранее, в среде Web-браузера выполняется код, написанный неким "третьим лицом", зачастую неизвестным пользователю. Выполнение мобильного кода, копируемого по сети, представляет потенциальную опасность для локальной системы. Для того чтобы уменьшить эту опасность, производители браузеров организуют запуск JavaScript-программ в специально сформированной среде, которая носит название "песочница" (sandbox). При этом программа имеет ограниченный доступ к ресурсам локальной системы либо не имеет его вовсе. Так, Ajax-приложение не может читать информацию из локальной файловой системы или записывать ее. Код Ajax-клиента также не может устанавливать сетевое соединение ни с одним сервером, за исключением того, с которого он был скопирован. Элемент `IFrame`, сгенерированный в результате выполнения программы, позволяет получать документы с любого узла и даже запускать их, но сценарии из различных фреймов не могут взаимодействовать друг с другом. Такой подход носит название политики "сервера-источника".

Рассмотрим очень простой пример. Определим в сценарии некоторую переменную.

```
x=3;
```

Во втором файле сценария попытаемся использовать ее.

```
alert(top.x+4);
```

Первый сценарий включен в документ верхнего уровня. По его инициативе был создан элемент `IFrame` и загружена страница, содержащая второй сценарий (рис. 7.1).

Если оба сценария получены из одной области или из одного домена, функция `alert()` выполняется успешно. В противном случае при выполнении второго сценария возникает ошибка.

7.1.2. Особенности выполнения сценариев в Ajax-приложении

При взаимодействии, ориентированном на сценарии (см. главу 5), JavaScript-код копируется с сервера и непосредственно выполняется на стороне клиента. В большинстве случаев клиент получает код с того же сервера, с которого он был скопирован сам, однако возможны случаи получения программного кода из другого домена. При этом возникает так называемый *кросс-сценарий*. Предоставляя право получать сценарии с произвольно выбранных узлов, мы тем самым формируем условия для возможной подмены документов или

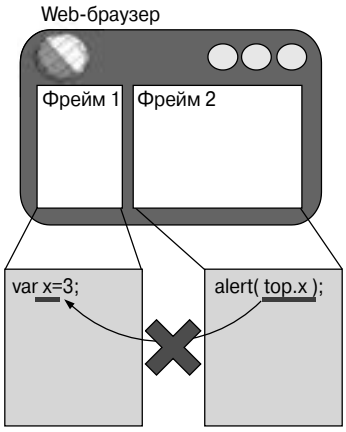

Рис. 7.1. Модель безопасности JavaScript запрещает сценариям, полученным из различных источников, взаимодействовать друг с другом

их искажения путем манипуляции DOM-информацией. Ограничения, предусмотренные в модели безопасности JavaScript, обычно обеспечивают реальную защиту от подобных явлений. Эта модель также предотвращает копирование клиентского кода Ajax на узлы, контролируемые злоумышленниками, и подмену сервера.

При взаимодействии, ориентированном на данные, риск значительно ниже, так как вместо исполняемого кода сервер предоставляет лишь данные. Тем не менее информация на серверах злоумышленников может быть подобрана так, чтобы нанести вред системе, используя известные недостатки в программах разбора. Таким образом, например, можно переопределить или удалить важную информацию или создать условия для неоправданно интенсивного потребления ресурсов.

7.1.3. Проблемы с поддоменами

Заметьте, что браузеры имеют весьма ограниченное представление о границах конкретной области, или домена, и в ряде случаев, когда можно было бы обойтись предупреждением, генерируют ошибку. Браузер идентифицирует домен по первой части URL и не делает попыток выяснить, указывают ли различные доменные имена на один и тот же IP-адрес. В табл. 7.1 приведено несколько примеров решений, принимаемых моделью безопасности браузера.

В случае поддоменов возможно выделить для сопоставления часть домена путем установки свойства `document.domain`. Так, например, мы можем включить приведенную ниже строку в сценарий, полученный с узла `http://www.myisp.com/dave`.

```
document.domain='myisp.com';
```

Этим мы разрешим взаимодействие со сценарием, полученным из поддомена `http://dave.myisp.com/`, при условии, что в нем также будет установлено значение `document.domain`. Заметьте, что установить произвольное значение `document.domain`, например `www.google.com`, невозможно.

Таблица 7.1. Примеры применения браузером политики безопасности

URL	Разрешены ли кросс-сценарии	Описание
`http://www.mysite.com/script1.js` `http://www.mysite.com/script2.js`	Да	Сценарии получены из одного домена
`http://www.mysite.com:8080/script1.js` `http://www.mysite.com/script2.js`	Нет	Номера портов не совпадают (первый сценарий получен посредством порта 8080)
`http://www.mysite.com/script1.js` `https://www.mysite.com/script2.js`	Нет	Протоколы не совпадают (второй сценарий получен посредством защищенного протокола HTTP)
`http://www.mysite.com/script1.js` `http://192.168.0.1/script2.js,`	Нет	Доменное имя `www.mysite.com` соответствует адресу 192.168.0.1, но браузер не проверяет этот факт
`http://www.mysite.com/script1.js` `http://scripts.mysite.com/script2.js`	Нет	Поддомены считаются различными доменами
`http://www.myisp.com/dave/script1.js` `http://www.myisp.com/eric/script2.js`	Да	Несмотря на то что сценарии получены с узлов, контролируемых различными владельцами, домены считаются совпадающими
`http://www.myisp.com/dave/script1.js` `https://www.mysite.com/script2.js`	Нет	Имя `www.mysite.com` указывает на `www.myisp.com/dave`, но браузер не проверяет этот факт

7.1.4. Несоответствие средств защиты в различных браузерах

Наш разговор о безопасности нельзя считать завершенным без рассмотрения существенного расхождения в средствах защиты разных браузеров. Система обеспечения безопасности Internet Explorer оперирует с набором "зон безопасности" с ограниченными правами. По умолчанию исполняемые файлы из локальной файловой системы имеют право взаимодействовать с узлами из всемирной сети, не оповещая об этом пользователя (по крайней мере, такое поведение предусмотрено в Internet Explorer 6). Таким образом, локальная

Рис. 7.2. Диалоговое окно с предупреждением, которое Internet Explorer отображает при попытке обращения к серверу, отличному от "сервера-источника". Если пользователь разрешит подобное взаимодействие, последующие обращения прерываться не будут

файловая система считается безопасной зоной. При попытке выполнения того же кода, но полученного с локального Web-сервера, отобразится диалоговое окно, показанное на рис. 7.2.

В случае необходимости можно написать сложное Ajax-приложение и проверять работоспособность больших фрагментов кода, оперируя данными из файловой системы. Временное исключение Web-сервера из работы системы несколько упрощает создание кода. Однако мы настоятельно рекомендуем разработчикам наряду с отработкой взаимодействия на локальной файловой системе проверять его на реальном Web-сервере. В браузере Mozilla понятие зон не используется. Приложение, загруженное из локальной системы, испытывает на себе те же ограничения, что и приложение, загруженное с Web-сервера. В Internet Explorer коды, полученные из различных зон безопасности, будут вести себя по-разному.

Итак, мы сформулировали основные ограничения, которые могут быть применены к сценариям в составе Ajax-приложений. Модель безопасности JavaScript в ряде случаев затрудняет работу, но в целом делает ее более надежной. Без нее доверие пользователей к службам Интернета и в том числе к средствам, предоставляемым Ajax, было бы столь низким, что их бы практически никто не применял.

В то же время бывает необходимо выполнять сценарии из доменов, отличающихся от того, из которого был получен сценарий. Такая необходимость возникает, например, при работе с рядом Web-служб. В следующем разделе вы увидите, как можно ослабить ограничения, связанные с защитой.

7.2. Взаимодействие с удаленным сервером

Средства защиты в составе браузера, конечно же, необходимы, но бывают ситуации, когда они лишь мешают нормальной работе. Чтобы система, обеспечивающая безопасность, работала эффективно, она обязана подвергать сомнению правомочность каждого действия. Тем не менее в ряде случаев приложению бывает необходимо обратиться к стороннему серверу. Теперь, когда вы имеете представление о политике безопасности браузера, давайте подумаем о том, как ослабить налагаемые ограничения. Мы рассмотрим два возможных решения этой задачи. Одно из них предполагает наличие дополнительного кода на стороне сервера, а второе затрагивает только клиентскую программу.

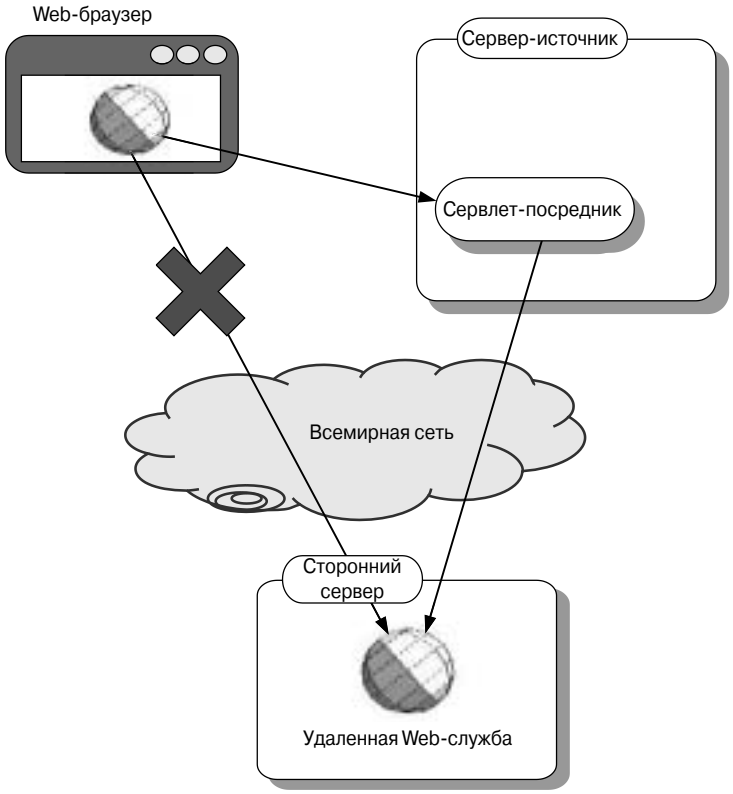

Рис. 7.3. Если Ajax-приложению необходимо получить доступ к ресурсам из другого домена, "сервер-источник" выступает в роли посредника, или прокси-сервера

7.2.1. Сервер в роли посредника при обращении к удаленной службе

В соответствии с политикой "сервера-источника" Ajax-приложение может копировать данные лишь из своего собственного домена. Если нам надо получать информацию с другого сервера, можно организовать обращение к нему не с клиентской машины, а с "сервера-источника", а ответ перенаправить клиенту (рис. 7.3).

В этом случае с точки зрения браузера данные поступают с того сервера, с которого загружена клиентская программа, что не противоречит политике "сервера-источника". Кроме того, перед перенаправлением информации клиенту на сервере может быть выполнена проверка на наличие недопустимых данных.

Недостатком такого подхода является увеличение нагрузки на сервер. Решение, которое мы рассмотрим ниже, предполагает обращение браузера непосредственно к требуемому серверу.

7.2.2. Взаимодействие с Web-службами

В настоящее время многие организации предоставляют Web-службы, которые могут быть использованы различными программами, включая JavaScript-клиентов. При работе клиента Ajax желательно иметь возможность непосредственно обращаться к Web-службе. Помехой этому является политика "сервера-источника". Решить данную проблему можно, запрашивая из программы привилегии, необходимые для выполнения требуемых действий в сети. Запрос на получение привилегий отображается на экране, чтобы пользователь подтвердил его правомочность. Браузер может запомнить решение пользователя и в дальнейшем оно будет приниматься автоматически.

В данном разделе мы рассмотрим обращение клиента Ajax к произвольно выбранной Web-службе. Браузеры Internet Explorer и Mozilla Firefox обрабатывают запрос на получение привилегий по-разному, поэтому нам необходимо найти способы, позволяющие работать с ними обоими.

Программа, рассматриваемая в качестве примера, устанавливает взаимодействие с одной из Web-служб Google, используя для этой цели SOAP (Simple Object Access Protocol). Протокол SOAP базируется на HTTP. В составе запроса серверу передается XML-документ, содержащий значения параметров, а ответом сервера является XML-документ, описывающий результаты. XML-документ, пересылаемый по протоколу SOAP, имеет достаточно простую структуру и состоит из заголовков и содержимого, помещенных в своеобразный "конверт". Использование формата XML упрощает применение объекта `XMLHttpRequest` для работы с данным протоколом.

Для работы со своей поисковой машиной Google предоставляет интерфейс SOAP. Пользователь может передавать ключевые слова в составе запроса и получать в ответ XML-документ, содержащий результаты. Ответ в формате XML очень похож на данные, отображаемые Google на странице поиска. Каждый пункт набора результатов содержит заголовок, фрагмент текста, описание и URL. В ответе также указывается приблизительное число документов, удовлетворяющих условиям поиска.

Наше приложение представляет собой своеобразную игру с применением Интернета. В ней будет использоваться информация о количестве документов, полученных в результате поиска. Пользователю предлагается простая форма и случайным образом сгенерированное большое число (рис. 7.4). Задача пользователя — задать ключевые слова, при обработке которых сервером Google будет получено количество результатов, отличающихся от заданного числа не более, чем на 1000.

Мы будем взаимодействовать с SOAP-службой Google посредством `XMLHttpRequest` в составе объекта `ContentLoader` (его мы разработали в главе 3). Последний раз мы обращались к данному объекту в главе 6, где оснастили его некоторыми новыми возможностями. Если мы используем последнюю версию `ContentLoader` для взаимодействия с Google, то получим желаемые результаты при работе с браузером Internet Explorer, но не с Mozilla. Рассмотрим вкратце особенности поведения каждого браузера.

Рис. 7.4. Использование API SOAP, предоставляемого Google, в простом Ajax-приложении. Пользователь вводит ключевые слова для поиска. Число результатов, предоставляемых Google, должно укладываться в заранее заданный диапазон значений

Браузер Internet Explorer и Web-службы

Как было сказано ранее, система защиты Internet Explorer базируется на понятии зон безопасности. Если мы скопируем наше приложение с Web-сервера, даже локального, оно будет рассматриваться как представляющее опасность для системы. Когда мы в первый раз обратимся к Google, используя `ContentLoader`, то получим сообщение, подобное тому, которое было представлено на рис. 7.2. Если пользователь щелкнет на кнопке `Yes`, передача данного запроса, а также всех последующих запросов к тому же серверу будет разрешена. Если пользователь щелкнет на кнопке `No`, запрос будет отвергнут и управление получит обработчик ошибок `ContentLoader`. В данном случае обеспечивается средний уровень защиты, а пользователь испытывает небольшие неудобства в работе.

Как вы помните, при запуске Ajax-клиента из локальной файловой системы браузер Internet Explorer рассматривает эту среду как безопасную, и диалоговое окно не отображается.

Браузеры Mozilla, в том числе Firefox, используют более строгий подход к обеспечению защиты, и получить требуемые права сложнее.

Mozilla PrivilegeManager

Модель защиты браузера Mozilla основана на понятии привилегий. Считается, что каждое действие, независимо от того, является ли оно обращением к стороннему Web-серверу или чтением файлов из локальной файловой системы, может представлять опасность для системы. Чтобы выполнить действие, код приложения должен запросить соответствующие привилегии. Привилегиями управляет объект `netscape.security.PrivilegeManager`. Если мы хотим, чтобы клиентская программа Ajax взаимодействовала с сервером Google, нам надо сначала организовать обращение к `PrivilegeManager`. Firefox может быть настроен так, что `PrivilegeManager` даже не будет принимать обращения, причем для данных, обслуживаемых сервером, подобные

Рис. 7.5. Если приложение запрашивает у браузера Firefox дополнительные привилегии, на экране отображается окно с предупреждающим сообщением

установки принимаются по умолчанию. Таким образом, подход, рассматриваемый в данном разделе, в основном подходит для использования в сетях intranet. Поэтому последующий материал ориентирован в основном на разработчиков, которые создают программное обеспечение для внутренних сетей предприятия, а также на тех, кого интересуют особенности работы Firefox.

Запросить привилегии позволяет метод `enablePrivilege`. При обращении к нему выполнение сценария приостанавливается и на экране отображается диалоговое окно, показанное на рис. 7.5.

В окне выводится предупреждение о том, что сценарий собирается выполнить действие, которое потенциально может быть опасным. Пользователь может разрешить или запретить предоставление привилегий. В любом случае выполнение сценария возобновляется. Если привилегии получены, программа может выполнить требуемые действия. В противном случае она лишь пытается предпринять их, но в результате возникает ошибка.

Как вы уже знаете, Internet Explorer запоминает первое решение пользователя и, единожды отобразив предупреждающее сообщение, больше не повторяет его. Браузер Mozilla предоставляет привилегии только на время действия запросившей их функции, и, если пользователь не установит флажок опции `Remember my decision`, действия программы будут прерываться каждый раз, когда ей потребуются привилегии. (Как вы увидите, это может произойти дважды в течение одного запроса по сети.) Очевидно, что в данном случае требования безопасности и применимости противоречат друг другу.

Вернемся к объекту `ContentLoader` (мы уже обсуждали его в главах 3, 5 и 6). Постараемся ответить на вопрос, что надо сделать, чтобы передать запрос серверу Google. В модифицированном коде предусмотрены запросы к объекту `PrivilegeManager` (листинг 7.1). Мы также реализовали возможность формирования произвольных HTTP-заголовков. Это необходимо для создания сообщений SOAP.

Листинг 7.1. `ContentLoader`, взаимодействующий со средствами защиты

```
net.ContentLoader=function(
  url,onload,onerror,method,params,contentType,
 headers,secure
){
  this.req=null;
  this.onload=onload;
  this.onerror=(onerror) ? onerror :
  this.defaultError;
   this.secure=secure;
  this.loadXMLDoc(url,method,params,contentType,  headers);
}
net.ContentLoader.prototype={
  loadXMLDoc:function(url,method,params,contentType,headers){
    if (!method){
      method="GET";
    }
    if (!contentType && method=="POST"){
      contentType='application/x-www-form-urlencoded';
    }
    if (window.XMLHttpRequest){
      this.req=new XMLHttpRequest();
    } else if (window.ActiveXObject){
      this.req=new ActiveXObject("Microsoft.XMLHTTP");
    }
     if (this.req){
       try{
        try{
          if (this.secure && netscape
            && netscape.security.PrivilegeManager.enablePrivilege) {
// ❶ Получение привилегий для передачи запроса
            netscape.security.PrivilegeManager
              .enablePrivilege('UniversalBrowserRead');
          }
        }catch (err){}
        this.req.open(method,url,true);
        if (contentType){
          this.req.setRequestHeader('Content-Type', contentType);
        }
 // ❷ Добавление поля заголовка HTTP
        if (headers){
          for (var h in headers){
            this.req.setRequestHeader(h,headers[h]);
          }
        }
        var loader=this;
        this.req.onreadystatechange=function(){
          loader.onReadyState.call(loader);
        }
        this.req.send(params);
      }catch (err){
        this.onerror.call(this);
      }
    }
},
```

```
onReadyState:function(){
  var req=this.req;
  var ready=req.readyState;
  if (ready==net.READY_STATE_COMPLETE){
    var httpStatus=req.status;
    if (httpStatus==200 || httpStatus==0){
      try{
        if (this.secure && netscape
          && netscape.security.PrivilegeManager.enablePrivilege) {
          // Получение привилегий для разбора ответа
netscape.security.PrivilegeManager
.enablePrivilege('UniversalBrowserRead');
        }
      }catch (err){}
      this.onload.call(this);
    }else{
      this.onerror.call(this);
    }
  }
},
  defaultError:function(){
    alert("error fetching data!"
      +"\n\nreadyState:"+this.req.readyState
      +"\nstatus: "+this.req.status
      +"\nheaders: "+this.req.getAllResponseHeaders());
  }
}
```

К конструктору добавлены дополнительные параметры. Первый параметр представляет собой массив полей заголовка HTTP ❷. Эти поля нам придется использовать при формировании запроса SOAP. Второй дополнительный параметр — логическое значение, выполняющее роль флага, который показывает, следует ли запрашивать привилегии на определенных этапах выполнения программы.

Когда мы запрашиваем привилегии, обращаясь к объекту `netscape.PrivilegeManager`, мы получаем их лишь на время выполнения текущей функции. Следовательно, нам надо запрашивать привилегии дважды: для обращения к удаленному серверу ❶ и для чтения полученного ответа ❷. Функцию обработчика `onload` мы вызываем в области видимости функции `onReadyState()`, поэтому полученные привилегии будут действовать и для нее.

Браузер Internet Explorer не поддерживает `PrivilegeManager`, и при обращении к данному объекту будет сгенерировано исключение. Поэтому мы помещаем обращения в блок `try...catch`, в результате чего исключение будет перехвачено и должным образом обработано. Если приведенный выше код будет выполнен в среде Internet Explorer, в блоке `try...catch` возникнет ошибка и программа не будет завершена аварийно. Под управлением браузера Mozilla работа с `PrivilegeManager` не вызовет исключения.

Воспользуемся модифицированным объектом `ContentLoader` для передачи запроса Google. В листинге 7.2 приведен HTML-код, обеспечивающий работу приложения.

Листинг 7.2. Содержимое файла `googleSoap.html`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Google Guessing</title>
  <script type="text/javascript" src='net_secure.js'></script>
  <script type="text/javascript" src='googleSoap.js'></script>
  <script type='text/javascript'>
    var googleKey=null;
    var guessRange = 1000;
    var intNum = Math.round(Math.random()
      * Math.pow(10,Math.round(Math.random()*8)));
    window.onload = function(){
      document.getElementById("spanNumber")
        .innerHTML = intNum + " and "
        + (intNum + guessRange);
    }

</script>
</head>
<body>
  <form name="Form1" onsubmit="submitGuess();return false;">
  Obtain a search that returns between 
  <span id="spanNumber"></span>  results!<br/>
  <input type="text" name="yourGuess" value="Ajax">
    <input type="submit" name="b1" value="Guess"/><br/><br/>
    <span id="spanResults"></span>
  </form>
  <hr/>
  <textarea rows='24' cols='100' id='details'></textarea>
</body>
</html>
```

В состав HTML-файла мы включаем элементы формы и код для вычисления достаточно большого псевдослучайного числа. Мы также объявляем переменную `googleKey`. Это ключ, позволяющий использовать API Google SOAP. Мы не приводим здесь реального ключа, так как это противоречило бы лицензионным соглашениям. Ключи предоставляются бесплатно и дают возможность выполнять в течение дня ограниченное число операций поиска. Процедура получения ключа достаточно проста. Для этого надо обратиться по URL, приведенному в конце данной главы.

Передача запроса

Основная часть работы выполняется функцией `submitGuess()`, которая вызывается при активизации кнопки `sumit`. Эта функция определена во включаемом JavaScript-файле. JavaScript-код этой функции, осуществляющий обращение к API Google, приведен в листинге 7.3.

Листинг 7.3. Функция `submitGuess()`

```
function submitGuess(){
// ❶ Проверка ключа лицензирования
  if (!googleKey){
    alert("You will need to get a license key "
      +"from Google,\n and insert it into"
      +"the script tag in the html file\n"
      +"before this example will run.");
    return null;
  }
  var myGuess=document.Form1.yourGuess.value;

// ❷ Создание сообщения SOAP
  var strSoap='<?xml version="1.0" encoding="UTF-8"?>'
    +'\n\n<SOAP-ENV:Envelope'
    +' xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"'
    +' xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"'
    +' xmlns:xsd="http://www.w3.org/1999/XMLSchema">'
    +'<SOAP-ENV:Body><ns1:doGoogleSearch'
    +' xmlns:ns1="urn:GoogleSearch"'
    +' SOAP-ENV:encodingStyle='
    +'"http://schemas.xmlsoap.org/soap/encoding/">'
    +'<key xsi:type="xsd:string">' + googleKey + '</key>'
    +'<q xsi:type="xsd:string">'+myGuess+'</q>'
    +'<start xsi:type="xsd:int">0</start>'
    +'<maxResults xsi:type="xsd:int">1</maxResults>'
    +'<filter xsi:type="xsd:boolean">true</filter>'
    +'<restrict xsi:type="xsd:string"></restrict>'
    +'<safeSearch xsi:type="xsd:boolean">false</safeSearch>'
    +'<lr xsi:type="xsd:string"></lr>'
    +'<ie xsi:type="xsd:string">latin1</ie>'
    +'<oe xsi:type="xsd:string">latin1</oe>'
    +'</ns1:doGoogleSearch>'
    +'</SOAP-ENV:Body>'
    +'</SOAP-ENV:Envelope>';

// ❸ Создание ContentLoader, в том числе
// предоставление URL Google API и
// передача дополнительных полей заголовка
  var loader=new net.ContentLoader(
    "http://api.google.com/search/beta2",
    parseGoogleResponse,
    googleErrorHandler,
    "POST",
    strSoap,
    "text/xml",
    {
      Man:"POST http://api.google.com/search/beta2 HTTP/1.1",
      MessageType:"CALL"
    },
    true
  );
}
```

Выполнение функции `submitGuess()` начинается с проверки наличия ключа лицензирования. Если ключ отсутствует, программа напоминает об этом пользователю ❶. Если вы скопируете код данного примера, значение ключа лицензирования будет установлено равным `null`, поэтому вам надо получить с сервера Google реальный ключ.

Следующая задача — создать SOAP-сообщение ❷ большого объема, содержащее ключевые слова поиска и ключ лицензирования. Разработчики SOAP предполагали, что все основные действия будут автоматизированы, поэтому, создавая XML-файл вручную, мы отступаем от основных принципов работы со службами. Браузеры Internet Explorer и Mozilla предоставляют объекты, упрощающие взаимодействие посредством SOAP. Однако на наш взгляд, будет полезно хотя бы раз вручную сформировать SOAP-запрос и разобрать ответ.

Имея готовый XML-текст, мы создаем объект `ContentLoader` ❸, при этом задаем SOAP XML в качестве содержимого HTTP-запроса, а также указываем URL API Google и поля HTTP-заголовка. В качестве типа содержимого указывается значение `text/xml`. Заметьте, что MIME-тип определяет тип данных в теле сообщения, а не тип информации, которую мы ожидаем получить в ответе (в данном случае эти типы совпадают). Последний параметр — значение `true`, которое указывает на то, что мы должны запрашивать необходимые права с помощью объекта `PrivilegeManager`.

Разбор ответа

Объект `ContentLoader` создает запрос, в ответ на который (с согласия пользователя) будет получен большой объем XML-данных. Ниже приведен фрагмент ответа, представляющего собой результаты поиска по ключевому слову Ajax.

```
<?xml version='1.0' encoding='utf-8'?>
<soap-env:envelope
  xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/xmlschema-instance"
  xmlns:xsd="http://www.w3.org/1999/xmlschema">
<soap-env:body>
  <ns1:dogooglesearchresponse xmlns:ns1="urn:googlesearch"
    soap-env:encodingstyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="ns1:googlesearchresult">
<directorycategories
  xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
  xsi:type="ns2:array"
  ns2:arraytype="ns1:directorycategory[1]">
...
<estimateisexact xsi:type="xsd:boolean">false</estimateisexact>
<estimatedtotalresultscount
xsi:type="xsd:int">741000</estimatedtotalresultscount>
...
<hostname xsi:type="xsd:string"></hostname>
<relatedinformationpresent xsi:type="xsd:boolean">true
    </relatedinformationpresent>
<snippet xsi:type="xsd:string">de officiлle site van afc
```

```
    &lt;b&gt;ajax&lt;/b&gt;.</snippet>
<summary xsi:type="xsd:string">official club site,
  including roster, history, wallpapers, and video clips.
    &lt;br&gt; [english/dutch]</summary>
<title xsi:type="xsd:string">
&lt;b&gt;ajax&lt;/b&gt;.nl - splashpagina
</title>
...
```

Реальный SOAP-ответ содержит гораздо больший объем информации. В первой части ответа определены некоторые заголовки, имеющие отношение к доставке данных, в частности, там указан источник, из которого поступил ответ. В теле документа присутствуют элементы, описывающие объем набора результатов. В данном случае количество результатов приблизительно оценено как 741000. Также мы видим часть первого результата поиска — ссылку на страницу голландского футбольного клуба. В листинге 7.4 показан обработчик обратного вызова, с помощью которого мы осуществляем разбор результатов.

Листинг 7.4. Функция `parseGoogleResponse()`

```
function parseGoogleResponse(){
  var doc=this.req.responseText.toLowerCase();
  document.getElementById('details').value=doc;
  var startTag='<estimatedtotalresultscount xsi:type="xsd:int">';
  var endTag='</estimatedtotalresultscount>';
  var spot1=doc.indexOf(startTag);
  var spot2=doc.indexOf(endTag);
  var strTotal1=doc.substring(spot1+startTag.length,spot2);
  var total1=parseInt(strTotal1);
  var strOut="";
  if(total1>=intNum && total1<=intNum+guessRange){
    strOut+="You guessed right!";
  }else{
    strOut+="WRONG! Try again!";
  }
  strOut+="<br/>Your search for <strong>"
        +document.Form1.yourGuess.value
        +"</strong> returned " + strTotal1 + " results!";
  document.getElementById("spanResults").innerHTML = strOut;
}
```

В данном случае нас интересует не структура SOAP-сообщения, а лишь предполагаемое количество результатов. Ответ представляет собой корректный XML-документ, и мы могли бы использовать для его разбора свойство `responseXML` объекта `XMLHttpRequest`. Однако мы пойдем по пути наименьшего сопротивления и извлечем число результатов путем преобразования строки. Затем, выполняя операции со структурой DOM, мы сообщим пользователю о том, насколько удачно он подобрал ключевые слова. Для тех, кто хочет более подробно разобраться с данными SOAP, мы выведем весь XML-ответ в текстовой области.

Обеспечение доступа к PrivilegeManager в Firefox

Как было сказано ранее, `PrivilegeManager` может быть сконфигурирован так, что он не будет реагировать на запросы из программы. Для того чтобы выяснить, выполнены ли в браузере Firefox подобные установки, надо ввести в строке, предназначенной для указания адреса, `about:config`. В ответ будет выведена информация о текущей настройке. Используя поле редактирования, выполняющее фильтрацию, найдем пункт `signed.applets.codebase_principal_support`. Если значение данного свойства равно `true`, рассмотренный ранее код будет работать. В противном случае нам не удастся организовать взаимодействие с Google.

В ранних версиях Mozilla установки должны были выполняться вручную, после чего необходимо было перезапустить браузер. В Firefox, дважды щелкнув на соответствующей строке списка установок, можно заменить значение `true` на `false` и обратно. Изменения, вносимые таким образом, вступают в действие немедленно; при этом не только не требуется перезапускать браузер, но, если параметры отображаются на отдельной вкладке, не надо даже обновлять страницу.

Подписание клиентского кода для выполнения в среде Mozilla

Поскольку Internet Explorer не использует `PrivilegeManager`, многократного подтверждать правомочность действий программы не приходится. В среде Mozilla дважды отображается диалоговое окно, что создает определенные неудобства для пользователя (конечно, окно выводится только в случае, если конфигурация браузера позволяет использовать `PrivilegeManager`). Повторного отображения диалогового окна можно избежать, установив флажок опции `Remember my decision` (рис. 7.5), но мы, как разработчики, лишены возможности сделать это из программы.

Существует решение этой проблемы, но оно предполагает специальное оформление приложения. Web-приложение может быть подписано с помощью сертификата. Чтобы сделать это, нам надо представить приложение браузеру в виде JAR-файла — сжатого архива, содержащего все сценарии, HTML-страницы, изображения и другие ресурсы. Подготовленный JAR-файл можно подписать, используя сертификат, полученный у Thawte, VeriSign или другой компании, занимающейся подобной деятельностью. Для обращения к ресурсам, содержащимся в подписанном JAR-файле, используются специальные выражения, подобные представленному ниже.

```
jar:http://myserver/mySignedJar.jar!/path/to/someWebPage.html
```

Когда пользователь копирует подписанное Web-приложение, он должен лишь один раз подтвердить предоставление привилегий; повторно ему подобные вопросы не задаются.

Браузер Mozilla предоставляет бесплатные инструментальные средства для подписания JAR-файлов. Те пользователи, которые собираются лишь поэкспериментировать с данной технологией, могут сгенерировать неаутентифицируемые сертификаты. Для этой цели предназначена утилита `keytool`, поставляемая в составе Java Development Kit (JDK). Мы же рекомендуем

обзавестись настоящим сертификатом, который можно будет использовать в практических целях.

Подписанные JAR-файлы не являются переносимыми и могут работать только с браузерами Mozilla. По этой причине мы не будем подробно рассматривать процедуру подписания кода. Те, кого интересует данный вопрос, могут воспользоваться информацией из источников, ссылки на которые приведены в конце данной главы.

На этом мы завершаем обсуждение вопросов взаимодействия приложений Ajax с удаленными службами. Мы выяснили, как приложение, выполняющееся в среде браузера, может обмениваться данными с сервером, с которого оно было скопировано, а при необходимости и со сторонними серверами. Программы, полученные с сервера, вряд ли смогут повредить вашей системе. Однако остается еще одна опасность, связанная с обменом информацией, особенно конфиденциальной. В следующем разделе мы поговорим о том, как скрыть данные пользователя от постороннего взгляда.

7.3. Защита конфиденциальной информации

Компьютер пользователя, на котором работает браузер, как правило, не имеет непосредственного соединения с сервером. Данные, передаваемые на сервер, проходят через промежуточные узлы (маршрутизаторы и прокси-серверы). Обычные HTTP-данные представляют собой незашифрованный текст, который можно прочитать в любой точке по пути следования пакета. Таким образом, информация может стать доступна любому, кто имеет контроль над промежуточными узлами.

7.3.1. Вмешательство в процесс передачи данных

Предположим, что вы написали приложение Ajax, которое передает по Интернету финансовую информацию, например, сведения о банковских счетах или номера платежных карточек. Если маршрутизатор работает корректно, он передает пакеты в неизменном виде и не читает их содержимое. Он лишь извлекает из заголовков данные, необходимые для выбора маршрута. Если же маршрутизатор контролируется злоумышленниками (рис. 7.6), он может собирать содержащиеся в пакетах сведения, выявляя, например, номера платежных карточек или почтовые адреса для рассылки спама. Он даже может изменять маршрут так, чтобы пакет попал на другой сервер, или модифицировать передаваемые данные (чтобы, например, положить деньги на другой счет).

В Ajax-приложениях протокол HTTP используется как для копирования клиентского кода, так и для передачи запросов клиента серверу. Все способы взаимодействия, которые мы рассматривали до сих пор, — скрытые элементы `IFrame`, HTML-формы, объекты XMLHttpRequest — с этой точки зрения одинаковы. Любое Web-приложение, в том числе и инфраструктура Ajax, имеет ряд уязвимых мест, которыми могут воспользоваться злоумышленники. Атака путем перехвата передаваемых данных на промежуточном узле носит

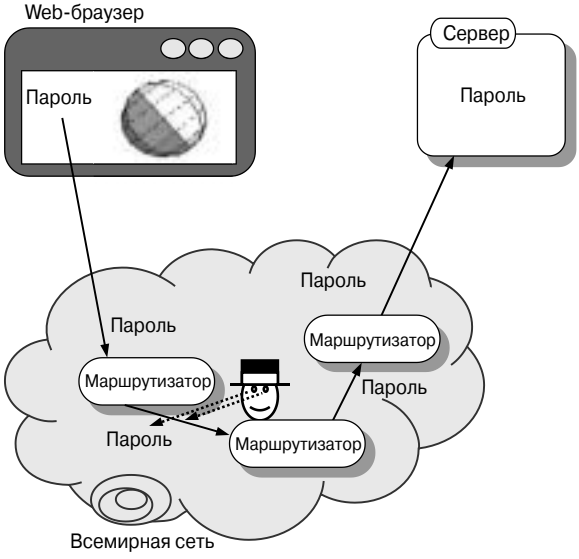

Рис. 7.6. При обычном HTTP-взаимодействии данные передаются по глобальной сети в виде незашифрованного текста, что позволяет злоумышленникам, контролирующим промежуточные узлы, прочитать информацию и даже изменить ее

название "человек посередине". Рассмотрим меры, которые можно принять для противодействия подобным атакам.

7.3.2. Организация защищенного HTTP-взаимодействия

Если вам необходимо защитить трафик между клиентской программой Ajax и сервером, самая очевидная мера, которую вы можете предпринять, — кодировать данные, используя защищенное соединение. Hypertext Transfer Protocol на базе Secure Socket Layer (HTTPS) реализует оболочку для обычного HTTP. Кодирование данных, передаваемых в обоих направлениях, осуществляется посредством пары ключей (открытого и закрытого). "Человеку посередине" по-прежнему доступно содержимое пакетов, но оно закодировано, и извлечь выгоду из данной информации невозможно (рис. 7.7).

Для того чтобы протокол HTTPS можно было применить на стороне браузера и на стороне сервера, необходима поддержка платформенно-ориентированного кода. Современные браузеры содержат встроенные средства для работы с HTTPS, и многие компании, предоставляющие на своих серверах пространство для Web-узлов, также предлагают защищенные соединения. При поддержке протокола HTTPS выполняются интенсивные вычисления, поэтому решение формировать передаваемые двоичные данные в программе на JavaScript вряд ли можно считать удачным. Мы не ставим задачу повторно реализовать с помощью JavaScript DOM, CSS или HTTP, поэтому

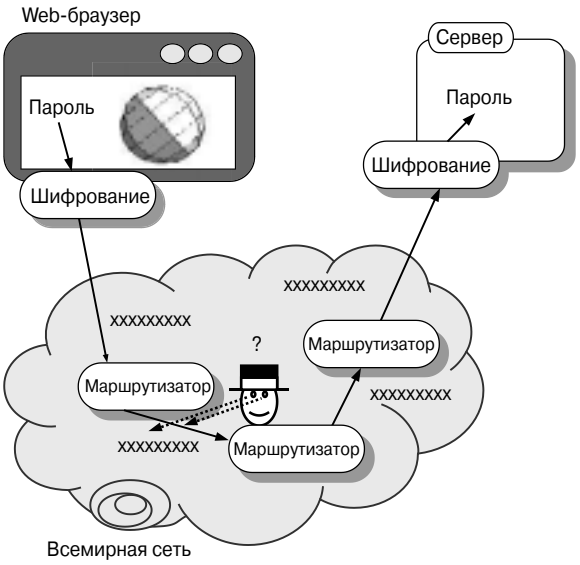

Рис. 7.7. При использовании защищенного HTTP-соединения данные, передаваемые в обоих направлениях, кодируются. На промежуточных узлах можно прочитать кодированную информацию, но ключ, требуемый для ее расшифровки, недоступен

HTTPS лучше всего рассматривать как используемую нами службу, а не как нечто, требующее переопределения.

Против использования HTTPS можно выдвинуть ряд аргументов. Во-первых, для кодирования и декодирования требуется большой объем вычислительных ресурсов. Это не создает проблему на стороне клиента, так как клиентская программа должна обрабатывать один поток данных. На сервере же дополнительная нагрузка крайне нежелательна. В особенности это важно на больших узлах. В классических Web-приложениях принято передавать посредством HTTPS только критичные данные, а обычное содержимое, например изображения или дескрипторы, формирующее общую структуру документа, пересылается посредством протокола HTTP. В Ajax-приложении необходимо учитывать влияние модели безопасности JavaScript, согласно которой `http://` и `https://` считаются различными протоколами. Во-вторых, HTTPS защищает только сам процесс передачи данных, но не обеспечивает безопасность приложения в целом. Если вы передадите по защищенному каналу номер платежной карточки, а затем поместите его в базу данных, в системе защиты которой имеются недостатки, ценная информация вполне может быть похищена.

Тем не менее HTTPS можно рекомендовать для передачи важных данных по сети. Стоимость такой передачи высока, и ее не всегда можно реализовать на небольших узлах. Если требования к защите не столь критичны, можно использовать обычный протокол HTTP для передачи шифрованных данных.

7.3.3. Передача шифрованных данных в ходе обычного HTTP-взаимодействия

Предположим, что вы поддерживаете небольшой Web-узел и при его работе не возникает потребности передавать секретные данные по защищенному каналу. Однако пользователи регистрируются на вашем узле, и вы хотите принять меры, чтобы пароль не попал в руки постороннему. В этой ситуации вам придет на помощью JavaScript. Рассмотрим основные принципы, на которых базируется возможное решение, а затем попробуем его реализовать.

Открытые и закрытые ключи

Если нам надо организовать передачу пароля, мы можем пересылать его в зашифрованном виде. Алгоритм шифрования преобразует входную строку в последовательность, с виду напоминающую случайный набор знаков. В качестве алгоритма кодирования можно выбрать MD5. Ряд возможностей, обеспечиваемых данным алгоритмом, позволяет применять его для обеспечения безопасности. Во-первых, в результате преобразования одного и того же фрагмента данных каждый раз будут получены одинаковые результаты. Во-вторых, вероятность того, что при преобразовании двух различных фрагментов будут получены одинаковые выходные данные, исчезающе мала. Благодаря этим особенностям данные, полученные в результате применения алгоритма, или MD5-дайджест, могут служить достаточно надежным идентификатором ресурсов. Есть и третья особенность. Она состоит в том, что алгоритма обратного преобразования не существует. Поэтому MD5-дайджест может пересылаться по сети в открытом виде. Если злоумышленник и перехватит его, он не сможет восстановить исходное сообщение.

Например, в результате преобразования строки `Ajax in action` будет сгенерирован MD5-дайджест `8bd04bbe6ad2709075458c03b6ed6c5a`. Выходные данные будут неизменными при каждом преобразовании. Мы можем закодировать строку на стороне клиента и передавать ее в таком виде серверу. Сервер извлечет пароль из базы, закодирует его, используя тот же алгоритм, и сравнит две строки. Если они совпадут, регистрация будет считаться успешной. Заметьте, что пароль в исходном виде по сети не передается.

При регистрации на сервере MD5-дайджест нельзя непосредственно передавать по Интернету. Злоумышленник не может выяснить, на основе какой исходной строки был сгенерирован дайджест, но может использовать сам дайджест в процессе регистрации. Здесь на помощь приходит механизм открытого и закрытого ключа. Вместо того чтобы кодировать один пароль, мы зашифруем его вместе с псевдослучайной последовательностью символов, предоставленной сервером. Сервер при каждой регистрации генерирует новую псевдослучайную последовательность и передает ее по сети клиенту.

На уровне клиента пользователь вводит пароль, а мы присоединяем к нему строку, предоставленную сервером, и шифруем результат конкатенации. Переданная клиенту последовательность символов хранится на сервере. Получив от клиента идентификатор пользователя, сервер читает из базы пароль, присоединяет к нему запомненную последовательность симво-

лов, шифрует их и сравнивает результаты. При совпадении закодированных последовательностей регистрация считается успешной. В противном случае (например, если пользователь неправильно ввел пароль) процедура регистрации повторяется, но при этом уже используется другая последовательность символов.

Предположим, что сервер передал строку `abcd`. MD5-дайджест для последовательности символов `Ajax in actionabcd` имеет вид `e992dc25b473842023f06a61f03f3787`. При следующей попытке регистрации клиенту будет передана строка `wxyz`. Присоединив ее к паролю и преобразовав результат, мы получим MD5-дайджест `3f2da3b3ee2795806793c56bf00a8b94`. Злоумышленник видит каждое сообщение, может сохранить его, но не способен зарегистрироваться посредством имеющейся информации. Таким образом, несмотря на возможность перехвата сообщений, воспроизвести последовательность действий. необходимую для успешной регистрации, не удастся.

Псевдослучайная строка выполняет роль открытого ключа и доступна для всех. Пароль можно считать закрытым ключом. Он хранится в течение длительного времени и не должен быть доступен посторонним.

Реализация процедуры регистрации средствами JavaScript

Для реализации описанного решения необходимо, чтобы и на стороне клиента, и на стороне сервера присутствовал MD5-генератор. Для клиентских программ существует свободно распространяемая библиотека генерации, реализованная на JavaScript Полом Джонстоном (Paul Johnston). Для того чтобы воспользоваться возможностями этой библиотеки, надо лишь включить ее в состав программы и вызвать соответствующую функцию.

```
<script type='text/javascript' src='md5.js'></script>
<script type='text/javascript'>var encrypted=str_md5('Ajax');
  // Выполнение требуемых действий ...
</script>
```

Для сервера алгоритм MD5 также поддерживается в ряде библиотек. В PHP, начиная с версии 3, содержится встроенная функция `md5()`. В классе `java.security.MessageDigest` реализованы популярные алгоритмы шифрования, в том числе MD5. В .NET Framework доступен класс `System.Security.Cryptography.MD5`.

Данный подход имеет ограниченную область применения, так как, для того, чтобы выполнять сравнение кодированных строк, серверу должен быть заранее известен пароль. Таким образом, рассмотренный механизм является хорошей альтернативой передаче ресурсов по протоколу HTTPS, но он не может полностью заменить этот протокол.

Итак, на данный момент мы получили следующие результаты. Политика "сервера-источника" позволяет противодействовать передаче на клиентскую машину кода, способного повредить систему. Нам известны меры борьбы с атакой типа "человек посередине" при обмене данными клиента с сервером. В последнем разделе мы сосредоточим внимание на сервере, который также может стать объектом атаки. Выясним, как можно защитить Web-службы от несанкционированного доступа.

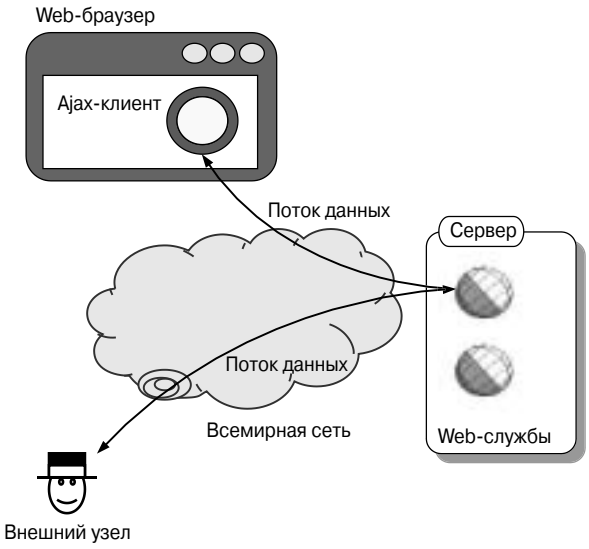

Рис. 7.8. В рамках инфраструктуры Ajax сервер предлагает свои услуги по Интернету. Доступ к серверу осуществляется по стандартному протоколу, чаще всего HTTP. Ajax-клиент получает потоки данных с сервера. Вследствие того что Web-службы общедоступны, внешние узлы сети могут получать данные непосредственно, минуя клиента

7.4. Управление доступом к потокам данных Ajax

Рассмотрим стандартную архитектуру Ajax с точки зрения наличия уязвимых мест в защите. Клиент, выполняющийся в среде браузера, передает запросы серверу, используя протокол HTTP. Эти запросы обслуживаются процессами на стороне сервера (сервлетами, динамическими документами и программами других типов), которые возвращают данные клиенту. Такое взаимодействие условно показано на рис. 7.8.

Услуги сервера или документы становятся доступными внешним узлам автоматически вследствие самой природы Интернета. В некоторых случаях мы даже поощряем подобное взаимодействие с сервером, в частности, публикуем API таких служб, как eBay, Amazon и Google. Однако и в этом случае нельзя упускать из виду вопросы защиты. Меры, которые мы можем предпринять для обеспечения защиты, описаны в следующих двух разделах. Во-первых, мы можем создать интерфейс Web-служб, или API, таким образом, чтобы им невозможно было воспользоваться некорректно, например, заказать товары, не заплатив за них. Во-вторых, имеется возможность ограничить доступ к Web-службам.

7.4.1. Создание защищенных программ на уровне сервера

Говоря о Web-приложении, мы обычно имеем в виду некоторую последовательность выполняемых действий. Например, при работе с интерактивным магазином пользователь просматривает имеющиеся товары и помещает некоторые из них в “корзинку”. Процесс выбора товаров четко определен: указывается адрес, по которому должны быть доставлены товары, способ оплаты и порядок подтверждения заказа. До тех пор, пока пользователь работает

с приложением, мы можем быть уверены, что взаимодействие будет осуществляться корректно. Если же некоторый узел сети предпримет попытку непосредственного обращения к Web-службе — возможны проблемы.

Программы сбора информации и Ajax-приложение

Классическое Web-приложение уязвимо для программ сбора информации, которые отслеживают последовательность действий пользователя, анализируя HTTP-запросы и извлекая те из них, в которых содержатся данные, введенные посредством формы. Программы сбора информации могут активно вмешиваться в действия приложения, нарушая порядок событий, обрабатываемых сервером, или перегружая сервер повторяющимися запросами. С точки зрения безопасности действия подобных программ могут стать источником серьезных проблем.

В классическом Web-приложении данные часто теряются в большом объеме HTML-кода, предназначенного для их внешнего оформления. В грамотно написанном Ajax-приложении Web-страницы, передаваемые клиенту, гораздо проще и представляют собой в основном структурированные данные. Разделение логики и представления является признаком профессионального подхода к написанию программ, но оно же упрощает работу программы сбора данных, поскольку информация, передаваемая сервером, предназначена в основном для разбора, а не для отображения посредством браузера. Когда внешний вид документа изменяется, программы сбора данных начинают работать менее надежно и могут даже вовсе выйти из строя. В Ajax-приложении правила обращения клиента к Web-службе меняются редко. Чтобы обеспечить целостность приложения, нам надо тщательно продумывать структуру высокоуровневого API, используемого при взаимодействии клиента и сервера. В данном случае под API мы понимаем не HTTP, SOAP или XML, а динамические страницы и параметры, передаваемые им.

Интерактивная игра “морской бой”

Чтобы лучше разобраться в том, как структура API Web-службы влияет на безопасность приложения, рассмотрим простой пример. Создадим интерактивную версию известной всем игра “морской бой”. В процессе игры Ajax-клиент будет взаимодействовать с сервером, используя Web-службы. Нам надо принять меры для того, чтобы правила игры выполнялись даже в том случае, когда один из игроков попытается мошенничать: захватит контроль над клиентом, чтобы передавать данные серверу вне очереди.

Задача игрока — угадать расположение кораблей противника. Игра состоит из двух этапов. Сначала игроки расставляют свои корабли на игровом поле. Затем они по очереди называют позиции, пытаясь попасть на расположение корабля противника. Копии игровых полей в процессе игры хранятся на сервере. Каждый клиент также поддерживает модель своего поля и поле другого игрока. Вначале второе поле пустое, но по мере угадывания позиций оно заполняется (рис. 7.9).

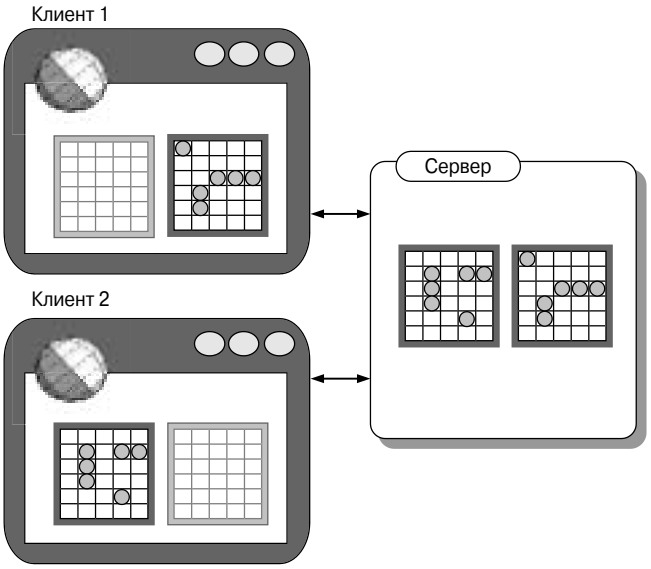

Рис. 7.9. Модели данных в Ajax-приложении, реализующем игру "морской бой". После расстановки кораблей на сервере располагаются модели игровых полей каждого участника. Вначале на стороне клиента содержится лишь модель своего игрового поля. Модель поля соперника формируется в процессе игры

Таблица 7.2. API для пошагового формирования игрового поля

URL	Параметры	Возвращаемые данные
clearBoard.do	Идентификатор пользователя	Подтверждение
positionShip.do	Идентификатор пользователя, длина корабля, координаты (x,y), ориентация (N,S,E или W)	Подтверждение или сообщение об ошибке

Рассмотрим первый этап игры. Вначале игровое поле пустое. Затем пользователь размещает на нем условные изображения своих кораблей. Организовать вызов сервера в процессе расстановки кораблей можно двумя способами. Первый способ предполагает обращения для очистки поля и для постановки каждого корабля в конкретной позиции. В процессе начальных установок обращение к серверу произойдет несколько раз: один раз при очистке поля и по одному обращению для постановки каждого корабля. API, соответствующий данному подходу, показан в табл. 7.2.

Второй способ предполагает очистку поля и установку всех кораблей в рамках одного обращения к серверу. В этом случае при настройке сервер вызывается лишь один раз. Соответствующий API показан в табл. 7.3.

Таблица 7.3. API, предполагающий формирование игрового поля в рамках одного запроса

URL	Параметры	Возвращаемые данные
setupBoard.do	Идентификатор пользователя, массив структур (x,y, длина корабля, ориентация)	Подтверждение или сообщение об ошибке

Таблица 7.4. API для второго этапа игры (используется независимо от того, какой подход был применен при конфигурировании игрового поля)

URL	Параметры	Возвращаемые данные
guessPosition.do	Идентификатор пользователя, координаты (x,y)	“hit” (попадание) “miss” (промах) или “not your turn” (не ваша очередь) плюс обновление с учетом хода другого игрока

Подходы, подобные описанным здесь, мы сравнивали между собой в главе 5 при обсуждении SOA. Однократное обращение по сети оказывается более эффективным и лучше обеспечивает разделение уровней. Кроме того, оно упрощает задачу обеспечения безопасности.

В рамках первого подхода клиент отвечает за контроль правильности количества и типов установленных кораблей. Модель, поддерживаемая на сервере, должна обеспечить проверку правильности конфигурации в конце настройки. Для второго подхода такая проверка может быть предусмотрена в формате документа, используемого при обращении к серверу.

По окончании настройки в действие вступает служба, обеспечивающая очередность ходов участников игры. Исходя из правил игры каждое обращение клиента должно соответствовать одному ходу — попытке угадать клетку, занимаемую каким-либо кораблем. Соответствующий API описан в табл. 7.4.

Если игра проходит корректно, оба пользователя по очереди обращаются к URL `guessPosition.do`. Сервер следит за очередностью ходов и в случае ее нарушения возвращает нарушителю сообщение `not your turn`.

Предположим теперь, что один из участников пытается играть нечестно. В его распоряжении клиентская программа, которая может обращаться к API Web-службы в любой момент. Что он может предпринять, чтобы получить преимущество в игре? Дополнительный ход сделать невозможно, так как сервер отслеживает эту ситуацию — очередность ходов заложена в API.

Единственный способ мошенничества — изменение положения корабля после окончания настройки. В рамках подхода, предполагающего многократные обращения к серверу, можно попытаться вызвать `positionShip.do` в процессе игры. Если серверная часть кода разработана грамотно, сервер определит, что это обращение противоречит правилам, и откажется обслуживать запрос. Однако, делая подобную попытку, игрок ничего не теряет, а задача разработчика серверной программы — предусмотреть подобную возможность и принять соответствующие меры.

Если в процессе настройки разрешено лишь одно обращение к серверу, изменить позицию одного корабля невозможно без очистки всего игрового поля. Таким образом, изменить настройку в свою пользу в принципе невозможно.

Настройка, допускающая единственное обращение к серверу, ограничивает свободу действий нечестного участника, не затрагивая при этом интересы игрока, соблюдающего правила. Если модель на стороне сервера функционирует правильно, API, предполагающий многократные обращения к серверу, не должен предоставить возможность несанкционированного вмешательства, однако число точек входа, в которых такое вмешательство оказывается потенциально возможным, значительно больше. В результате разработчику серверной программы приходится выполнять намного больший объем работы по проверке безопасности.

В разделе 5.3.4 мы говорили о том, что для упрощения API желательно применять образ разработки Façade. Подобный подход желательно использовать и сейчас, на этот раз для повышения уровня безопасности. Чем меньше точек входа, доступных из Интернета, тем проще реализовать требуемую политику.

Структура API может в какой-то степени защитить приложение от нежелательного воздействия извне, однако некоторые точки входа должны существовать для того, чтобы клиентская программа Ajax могла взаимодействовать с сервером. В следующем разделе мы рассмотрим способы их защиты.

7.4.2. Ограничение доступа к данным из Web

В идеале хотелось бы разрешить доступ к динамическим данным, подготовленным для нашего приложения, Ajax-клиенту (и, возможно, другим авторизованным программам) и запретить его всем остальным. Некоторые технологии, обеспечивающие работу с богатыми клиентами, дают возможность использовать специальные протоколы, однако в Ajax-приложениях наш выбор ограничен HTTP. Как вы уже знаете, защищенный протокол HTTP позволяет скрыть передаваемые данные от постороннего взгляда, но он не позволяет определить, кто обращается по конкретному URL.

К счастью, протокол HTTP обладает большими потенциальными возможностями, а объект `XMLHttpRequest` обеспечивает дополнительный контроль над ним. Когда запрос поступает на сервер, серверная программа имеет доступ к полям заголовка, из которых можно извлечь информацию об источнике запроса.

Фильтрация HTTP-запросов

Для написания конкретных примеров мы будем использовать язык Java. Существуют и другие технологии, позволяющие добиться аналогичных результатов. При создании Web-приложения на Java мы можем определить объект `javax.servlet.Filter`, который будет перехватывать запросы. В подклассе `Filter` переопределим метод `doFilter()` так, чтобы он анализирован HTTP-запрос перед тем, как принять решение о его передаче по назначению. В листинге 7.5 показан код простого фильтра, который перехватывает запрос,

а после либо продолжает его обычную обработку, либо появляется страница с сообщением об ошибке.

Листинг 7.5. Java-фильтр, используемый в системе защиты

```
public abstract class GenericSecurityFilter implements Filter {
  protected String rejectUrl=null;
  public void init(FilterConfig config)
  throws ServletException {
// ❶ Указание URL, соответствующего отвергнутому запросу
    rejectUrl=config.getInitParameter("rejectUrl"); }
  public void doFilter(
    ServletRequest request, ServletResponse response,
    FilterChain chain)
    throws IOException, ServletException {
// ❷ Проверка допустимости запроса
    if (isValidRequest(request)){
      chain.doFilter(request, response);
    }else if (rejectUrl!=null){
// ❸ Обращение к URL, соответствующему отвергнутому запросу
      RequestDispatcher dispatcher
       =request.getRequestDispatcher(rejectUrl);
      dispatcher.forward(request, response); }
  }
  protected abstract boolean
    isValidRequest(ServletRequest request);
  public void destroy(){}
}
```

Фильтр представляет собой абстрактный класс, в котором объявлен абстрактный метод `isValidRequest()`. Решение о дальнейшей судьбе запроса принимается на основании его анализа. Если в результате вызова метода `isValidRequest()` ❷ возвращается значение `false`, запрос перенаправляется по URL ❸, определенному в конфигурационном файле Web-приложения ❶.

Рассматриваемый здесь фильтр предоставляет возможность создавать конкретные подклассы, т.е. его можно адаптировать для различных стратегий защиты.

Использование HTTP-сеанса

Для поддержки сеанса часто используется следующий подход. При регистрации пользователя создается объект-идентификатор. При последующих обращениях производится проверка на наличие этого объекта. Простой фильтр, решающий данную задачу, показан в листинге 7.6.

Листинг 7.6. Фильтр для проверки идентификатора сеанса

```
public class SessionTokenSecurityFilter
extends GenericSecurityFilter {
  protected boolean isValidRequest(ServletRequest request) {
    boolean valid=false;
    HttpSession session=request.getSession();
    if (session!=null){
      UserToken token=(Token) session.getAttribute('userToken');
```

```
      if (token!=null){
        valid=true;
      }
    }
    return valid;
  }
}
```

Данный подход типичен для традиционных Web-приложений. Если проверка дает отрицательный результат, пользователю предлагается страница регистрации. В Ajax-приложениях ответ сервера может быть более простым по сравнению с традиционными Web-приложениями и содержать данные в формате XML, JSON либо в виде обычного текста. В ответ на получение определенных данных клиент может самостоятельно предоставить пользователю средства регистрации. В главе 11 мы обсудим реализацию средств регистрации в рамках приложения Ajax Portal.

Использование зашифрованных HTTP-заголовков

Существует еще один способ определения корректности запроса. Он предполагает добавление к HTTP-заголовку дополнительного поля и проверку его наличия посредством фильтра. В листинге 7.7 приведен пример фильтра, который ищет конкретное поле и проверяет совпадение зашифрованного значения с ключом, хранящимся на сервере.

Листинг 7.7. Фильтр для проверки поля HTTP-запроса

```
public class SecretHeaderSecurityFilter
extends GenericSecurityFilter {
  private String headerName=null;
  public void init(FilterConfig config) throws ServletException {
    super.init(config);
// Имя поля задается как конфигурационный параметр
    headerName=config.getInitParameter("headerName");
  }
  protected boolean isValidRequest(ServletRequest request) {
    boolean valid=true;
    HttpServletRequest hrequest=(HttpServletRequest)request;
    if (headerName!=null){
      valid=false;
// ❶ Получение значения заголовка
      String headerVal=hrequest.getHeader(headerName);
      Encrypter crypt=EncryptUtils.retrieve(hrequest);
      if (crypt!=null){
// ❷ Сравнение значения заголовка
        valid=crypt.compare(headerVal);
      }
    }
    return valid;
  }
}
```

При проверке запроса фильтр читает заголовок с определенным именем ❶ и сравнивает его с кодированным значением, содержащимся на сервере ❷. Данное значение не постоянно; оно генерируется для каждого конкретного сеанса, что затрудняет незаконное обращение к системе. Класс `Encrypter` использует для генерации MD5-значения Apache Commons Codec и `javax.security.MessageDigest`. Полностью набор классов можно получить, скопировав коды примеров для данной книги. Принцип формирования MD5-значения в шестнадцатеричном формате показан ниже.

```
MessageDigest
digest=MessageDigest.getInstance("MD5");
byte[] data=privKey.getBytes();
digest.update(data);
byte[] raw=digest.digest(pubKey.getBytes());
byte[] b64=Base64.encodeBase64(raw);
return new String(b64);
```

где `privKey` и `pubKey` — соответственно закрытый и открытый ключ.

Чтобы настроить фильтр для проверки всех URL, соответствующих пути `/Ajax/data`, надо добавить к конфигурационному файлу `web.xml` нашего приложения следующее определение:

```
<filter id='securityFilter_1'>
  <filter-name>HeaderChecker</filter-name>
  <filter-class>
   com.manning.ajaxinaction.web.SecretHeaderSecurityFilter
  </filter-class>
  <init-param id='securityFilter_1_param_1'>
    <param-name>rejectUrl</param-name>
    <param-value>/error/reject.do</param-value>
  </init-param>
  <init-param id='securityFilter_1_param_2'>
    <param-name>headerName</param-name>
    <param-value>secret-password</param-value>
  </init-param>
</filter>
```

Согласно данной конфигурации отклоненные запросы после проверки значения поля `secret-password` направляются URL `/error/reject.do`. Кроме того, мы определяем отображение фильтрации, в результате чего фильтр вступает в действие для любого запроса, соответствующего указанному пути.

```
<filter-mapping>
  <filter-name>HeaderChecker</filter-name>
  <url-pattern>/ajax/data/*</url-pattern>
</filter-mapping>
```

На стороне клиента для генерации MD5-дайджеста Base64 используется библиотека Пола Джонстона (она упоминалась ранее в этой главе). Для того чтобы добавить поле HTTP-заголовка, мы вызываем метод `setRequestHeader()`.

```
function loadXml(url){
  var req=null;
  if (window.XMLHttpRequest){
    req=new XMLHttpRequest();
```

```
  } else if (window.ActiveXObject){
    req=new ActiveXObject("Microsoft.XMLHTTP");
  }
  if (req){
    req.onreadystatechange=onReadyState;
    req.open('GET',url,true);
    req.setRequestHeader('secret-password',getEncryptedKey());
    req.send(params);
  }
}
```

Здесь функция кодирования создает MD5-дайджест Base64 для указанной строки.

```
var key="password";
function getEncryptedKey(){
  return b64_md5(key);
}
```

Данное решение предполагает передачу Ajax-клиенту значения переменной `key`. Ключ для сеанса можно передать по протоколу HTTPS при регистрации пользователя. Он должен представлять собой псевдослучайное значение, а не строку символов, например.

Положительная особенность данного решения состоит в том, что поле заголовка HTTP-запроса не может быть модифицировано посредством стандартной гипертекстовой ссылки или HTML-формы. Злоумышленникам придется программировать HTTP-клиент, а это требует определенного уровня подготовки. Очевидно, что по мере роста популярности объекта `XMLHttpRequest` среди разработчиков информация о том, как сформировать поле заголовка в составе запроса, будет становиться все более доступной. Следует заметить, что средствами, подобными Apache HTTPClient и Perl LWP::UserAgent, эту задачу можно было решить и раньше.

Фильтры и другие средства аналогичного назначения не являются непреодолимой преградой для тех, кто хочет обратиться к вашему узлу, но существенно усложняют эту задачу. Как и у любого разработчика, в распоряжении хакера имеются лишь ограниченные ресурсы, поэтому, защитив приложение несколькими способами, описанными выше, вы существенно снизите вероятность несанкционированного доступа к поддерживаемой вами службе.

На этом мы заканчиваем разговор о защите Ajax-приложений. Существуют вопросы, которые не нашли отражения в данной главе. Мы решили не обсуждать их лишь потому, что они типичны не только для инфраструктуры Ajax, но и для любых Web-приложений. Надежные средства аутентификации и авторизации помогут вам управлять доступом к службам. Стандартные HTTP-заголовки могут быть использованы для определения источника запроса, что затрудняет обращение к службе, минуя официальные каналы (однако не делает такое обращение невозможным). Те, кого заинтересовали вопросы безопасности Ajax-приложений, могут найти более подробные сведения в литературе, специально посвященной этой теме.

И наконец, помните, что безопасность — понятие относительное. Никакая защита не может быть абсолютно надежной. Максимум, чего можно добиться — опережать на шаг злоумышленников. Применение HTTPS и проверку

HTTP-запросов в тех случаях, когда такие меры оправданы, можно рассматривать как шаги в нужном направлении.

7.5. Резюме

В данной главе обсуждались вопросы безопасности Ajax-приложений. Мы сосредоточили внимание на тех особенностях, которые отличают защиту инфраструктуры Ajax от защиты обычных Web-приложений. Сначала мы рассмотрели "песочницу" (среду для выполнения JavaScript-программ в составе Web-браузера) и правила, предотвращающие взаимодействие фрагментов кода, полученных из различных источников. Мы также рассмотрели, как можно сделать политику "сервера-источника" менее строгой и разрешить доступ к сторонним Интернет-службам, например к API Google.

Кроме того, мы рассмотрели способы защиты данных, которыми клиент обменивается с сервером. Достаточно надежную защиту обеспечивает протокол HTTPS, но существует и более простое решение, обеспечивающее безопасную передачу пароля средствами обычного протокола HTTP. И наконец, мы показали вам уязвимое место Ajax-приложений, связанное с передачей низкоуровневых данных с сервера. Поскольку в некоторых случаях это может представлять серьезную угрозу, мы рассмотрели варианты архитектуры сервера, минимизирующие опасность. Мы также обсудили средства на стороне сервера, усложняющие несанкционированный доступ к данным.

Вопросы, рассмотренные в данной главе, помогут вам обеспечить работу Ajax-приложений в реальных условиях. В следующей главе мы продолжим разговор о характеристиках приложений, влияющих на их практическое применение. На этот раз речь пойдет о производительности программ.

7.6. Ресурсы

Ключи для использования API Web-служб Google можно получить, обратившись по адресу `http://www.google.com/apis/`.

JavaScript-библиотеки Пола Джонстона, позволяющие создавать MD5-дайджесты, представлены по адресу `http://pajhome.org.uk/crypt/md5/md5src.html`. Для тех, кому хочется быстро проверить MD5 в действии, можно посоветовать URL генератора контрольных сумм (`http://www.fileformat.info/tool/hash.htm?text= ajax+in+action`).

Библиотеку Apache Commons Codec для Java, которую мы использовали при генерации Base64 MD5 на сервере, можно скопировать с узла `http://jakarta.apache.org/commons/codec/`.

В разделе 7.1 мы рассмотрели подписание JAR-файлов для создания защищенных приложений, ориентированных на браузеры Mozilla. Дополнительная информация по этому вопросу содержится в документе `http://www.mozilla.org/projects/security/components/signed-scripts.html`. Сведения об игре "морской бой" доступны по адресу `http://gamesmuseum.uwaterloo.ca/vexhibit/Whitehill/Battleship/`.