CHAPTER 7

Client-State Manipulation

This chapter describes an additional type of attack that can occur due to unvalidated input: client-state manipulation.

In a web application, web clients (or browsers) make requests to web servers to access web pages. Web servers often invoke additional programs to help them construct the web pages that they send to clients. These additional programs are collectively referred to as a *web application*.

Web applications often accept input from their users. To be secure, *web applications should not trust clients*, and should validate all input received from clients.

The protocol that web clients and web servers use to communicate, HTTP, is *stateless*—web servers are not required to implicitly keep track of any state, or information, about their clients. Since HTTP was originally developed to just serve documents, a web client requests a document and the web server (possibly with the help of a web application) provides it. However, to conduct transactions (such as online purchases and funds transfers), a web application may have to receive input from and serve more than one document to a particular client. To keep track of which client is which while serving multiple clients at a time, a web server application can provide state information about a transaction to a client, which the client may echo back to the server in later requests. The echoed state information is input that the server receives as part of the HTTP request from the client.

In an example that follows, we illustrate a vulnerability that can exist if a web server does not validate such input itself. In our example, the web server uses "hidden" values in HTML forms to store sensitive information.

Hidden values in HTML forms are not directly shown to the user in the web browser's graphical user interface (GUI). However, you will see that these hidden values can be easily manipulated by malicious clients. You will learn that data submitted from hidden form fields should be considered input and validated just like all other input, even though the server typically generates information that is stored in hidden form fields.

7.1. Pizza Delivery Web Site Example

In our example, a user places an order for a pizza from a web site. Once the order is complete, a delivery person is dispatched by the web site to deliver the pizza to the user. There are three major steps that take place in our example pizza ordering application.[1] These steps are shown in Figure 7-1, and are described following:

1. *Order*: A user requests an `order.html` file from the web server, which contains an order form. The order form allows the user to choose the number of pizzas she wants to buy and input her credit card details to pay for the pizza. The order form is processed by a `confirm_order` script on the web server. Assume the user wants to buy one pizza that costs $5.50.

2. *Confirmation*: In this step, the user confirms the purchase. The user's web browser receives an HTML form generated by the `confirm_order` script on the web server, which states the cost of the pizza as $5.50 and presents the user with buttons to either proceed with or cancel the transaction. The following code shows the HTML form that is used to confirm the purchase:

```
1  <HTML>
2  <HEAD>
3  <TITLE>Pay for Pizza</TITLE>
4  </HEAD>
5  <BODY>
6  <FORM ACTION="submit_order" METHOD="GET">
7  The total cost is 5.50.
8  Are you sure you would like to order?
9  <INPUT TYPE="hidden" NAME="price" VALUE="5.50">
10 <INPUT TYPE="submit" NAME ="pay" VALUE="yes">
11 <INPUT TYPE="submit" NAME ="pay" VALUE="no">
12 </BODY>
13 </HTML>
```

Figure 7-2 shows how the browser displays this HTML form to the user. The title field in line 3 of the HTML is displayed in the title bar of the browser. The text "The total cost is $5.50. Are you sure you would like to order?" in lines 7 and 8 is displayed as is to the user. The hidden form field in line 9 that the server uses to store the price of the transaction is not displayed to the user. The remaining form fields are displayed as buttons labeled "yes" or "no," which the user can click to confirm or cancel the transaction. Since both these buttons are of type `submit`, the browser will issue an HTTP request such as the following when the user clicks on one of these buttons:

```
GET /submit_order?price=5.50&pay=yes HTTP/1.0
```

If the user clicks the "yes" button, the preceding HTTP request is issued.

1. Even if each of these steps takes place over an SSL connection, using SSL in itself does not prevent the client-state manipulation attacks that we describe in this chapter.

3. *Fulfillment*: Once the web server receives the HTTP request, it then sends a request to a credit card payment gateway to charge $5.50. Once the credit card payment gateway accepts the charge, the web server can dispatch the delivery person. Pseudocode for the actions the server conducts in the `submit_order` script is shown here:

```
if (pay = yes) {
     success = authorize_credit_card_charge(price);
     if (success) {
          settle_transaction(price);
          dispatch_delivery_person();
     }
     else {
          // Could not authorize card
          tell_user_card_declined();
     }
}
else {
     // pay = no
     display_transaction_cancelled_page();
}
```

In the preceding pseudocode, the variables `pay` and `price` are retrieved from the HTTP request.

The `submit_order` program first checks if the user clicked the "yes" button, indicating that she would like to purchase the pizza. If so, it attempts to authorize a credit card transaction for `price` dollars. If the authorization is successful, it settles the transaction with the credit card company and dispatches a pizza delivery person. If the credit card authorization does not succeed, the program aborts the transaction.

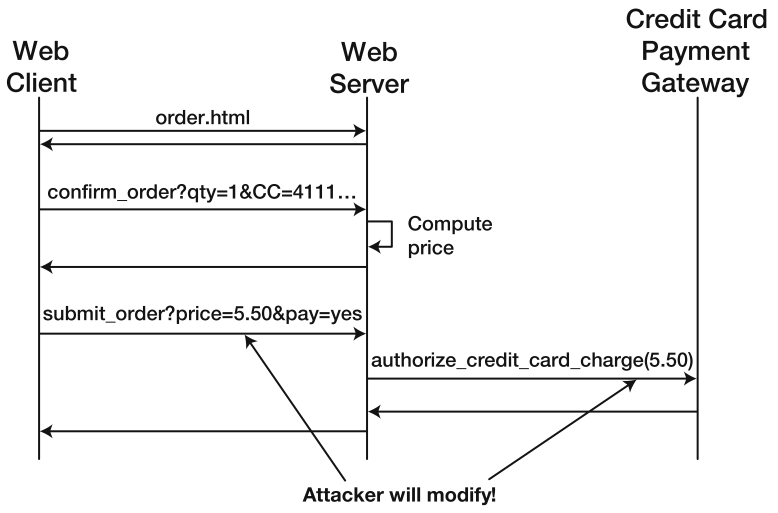

Web Client
Web Server
Credit Card Payment Gateway
order.html
confirm_order?qty=1&CC=4111...
Compute price
submit_order?price=5.50&pay=yes
authorize_credit_card_charge(5.50)
Attacker will modify!

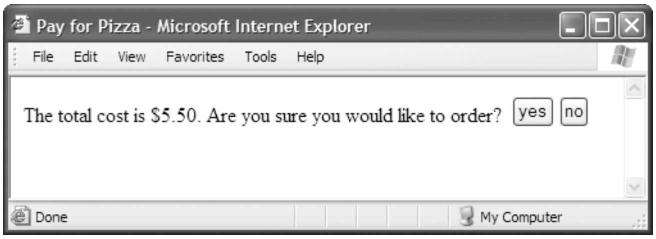
Figure 7-1. *Transaction flow*

The preceding is an overview of the "normal" interaction that occurs between the user's browser and the web server (also depicted in Figure 7-1). Due to the way that the `submit_order` code is written, an attacker will be able to purchase the pizza for a price of her choice instead of $5.50. We show how shortly.

Note that we are presenting a "toy" example here, in which we have left out many details. For example, the web site would need to have the user input her street address so that it knows where to send the pizza! In addition, the quantity and selection of toppings would also need to be chosen. These parameters could be included on the order form and stored as additional hidden form variables in the confirmation form.

7.1.1. Attack Scenario

Let's consider a scenario in which an attacker wants to order a pizza for $0.01 instead of $5.50. In the confirmation step of the transaction flow just described, the server sends back a page to the client with the computed total price for the pizza(s), and asks the user to confirm the transaction.


Pay for Pizza - Microsoft Internet Explorer
File Edit View Favorites Tools Help
yes
no
Done
My Computer

The total cost is $5.50. Are you sure you would like to order?

Figure 7-2. *Confirmation form*

The user can view the HTML source code that makes up the order confirmation form by selecting View | Source in the browser's menu bar.

Let's take a closer look at the HTML source code for the order confirmation form, which we repeat here for convenience:

```
1  <HTML>
2  <HEAD>
3  <TITLE>Pay for Pizza</TITLE>
4  </HEAD>
5  <BODY>
6  <FORM ACTION="submit_order" METHOD="GET">
7  The total cost is 5.50.
8  Are you sure you would like to order?
9  <INPUT TYPE="hidden" NAME="price" VALUE="5.50">
10 <INPUT TYPE="submit" NAME ="pay" VALUE="yes">
11 <INPUT TYPE="submit" NAME ="pay" VALUE="no">
12 </BODY>
13 </HTML>
```

In response to clicking either "yes" or "no" in the order confirmation form, a `submit_order` program will be run on the web server, as per the `action` attribute in the `form` tag in line 6 in the preceding code. The order confirmation form tells the user "The total cost is $5.50," and asks the user "Are you sure you would like to order?" This text can be seen in the HTML source just below the `form` tag. Below that is a "hidden" HTML form field that has a name `price` and a value of `5.50`, the total cost of the transaction. While the "hidden" form field is not shown on the browser user interface in Figure 7-2, it can easily be seen by viewing the HTML source code.

In this example, the server is storing the price of the transaction in the form sent to the client. Even worse, it is trusting the client with the price of the transaction. After the user clicks "yes" or "no," the user's response is recorded by a `pay` variable, and the `price` variable with a corresponding value is sent back to the server. To change the value of the transaction, the attacker can view the source code of the HTML form in a text editor, and change the value in the hidden form field from `5.50` to `0.01`. An attacker could simply save the modified HTML to disk, reopen it with a browser, and submit the form with the modified price to the server!

When the attacker clicks "yes" in the reopened HTML page, the HTTP request that is constructed based on the manipulated form is for the $0.01 transaction instead of the correct price of $5.50.

The HTTP request that is sent to the server looks like the following:[2]

```
GET /submit_order?price=0.01&pay=yes HTTP/1.0
```

The `submit_order` program simply retrieves the $0.01 price from the HTTP request, and authorizes a credit card charge of $0.01 prior to delivering the pizza!

To summarize, hidden form fields are only visually hidden from the user, but are effectively sent "in the clear" from a security standpoint. As such, they can be easily accessed and manipulated by malicious clients.

In this particular case, we have shown how the attacker can use the browser and a text editor to send an HTTP request with an altered price. However, there is no reason that an attacker needs to use a browser or text editor to place HTTP requests to buy pizzas. In fact, if the attacker did not want to order just one pizza for herself, but wanted to order pizzas for all her friends, using a browser and text editor to generate the HTTP requests might be quite tedious. Instead, the attacker could use tools such as `curl` (`http://curl.haxx.se`) or `Wget` (`www.gnu.org/software/wget`) to do so. These are open source, command-line tools that can be used to generate HTTP and other types of requests in an automated fashion. For instance, the preceding HTTP request could be automatically generated by issuing the following command:

```
curl https://www.deliver-me-pizza.com/submit_order?price=0.01&pay=yes
```

A similar HTTP request can be generated with `Wget`. Thus far, we have used `GET` HTTP requests, but switching to `POST` would not help very much. The attacker could still save the HTML for the form to disk, edit hidden values, and submit the form. In addition, the attacker could still use tools like `curl` and `Wget` to submit malicious requests. `POST` parameters can be submitted as follows using `curl`:

```
curl -dprice=0.01 -dpay=yes https://www.deliver-me-pizza.com/submit_order
```

2. For clarity, we have not shown other parameters that may need to be specified, such as the name of the person who ordered the pizza or the address to which it should be delivered.

`Wget` can also submit `POST` parameters as follows:

```
wget --post-data 'price=0.01&pay=yes' https://www.deliver-me-pizza.com/submit_order
```

Note that the attacker does not have to traverse through the order or confirmation web pages to issue the HTTP request to purchase the pizzas.

The big problem here is that there is no reason that the web server should trust any of its clients. By sending the transaction state back to the client in response to the order and confirmation forms, it gives the client the ability to tamper with that state.

There are a variety of possible solutions to this problem—we will discuss two of them. The first solution involves keeping an authoritative copy of the session state in a database at the server. The second solution involves sending the authoritative state back to the client, but with a "signature" that will alert the server to any potential tampering with the state.

7.1.2. Solution 1: Authoritative State Stays at Server

In this solution, the price of the transaction is not sent back to the client. Instead, a session identifier, the *session-id*, is sent to the client, and the server keeps a table of session-ids and the corresponding prices for client transactions. In response to filling out an order form, the server randomly generates a new 128-bit session-id, and sends it back as a hidden field in the confirmation form, as follows:

```
<HTML>
<HEAD>
<TITLE>Pay for Pizza</TITLE>
</HEAD>
<BODY>
<FORM ACTION="submit_order" METHOD="GET">
The total cost is 5.50.
Are you sure you would like to order?
<INPUT TYPE="hidden" NAME="session-id"
       VALUE="3927a837e947df203784d309c8372b8e">
<INPUT TYPE="submit" NAME ="pay" VALUE="yes">
<INPUT TYPE="submit" NAME ="pay" VALUE="no">
</BODY>
</HTML>
```

In the preceding HTML form, the session-id is `3927a837e947df203784d309c8372b8e`. Note that the price is not in the form. Instead, the server inserts the session-id and price into a database, as shown in Table 7-1.

Table 7-1. *Session-Id/Price Table*

Session-Id	Price
3927a837e947df203784d309c8372b8e	5.50

When the client submits the form, the following HTTP request arrives at the server:

```
GET /submit_order?session-id=3927a837e947df203784d309c8372b8e&pay=yes HTTP/1.0
```

The server now uses the following algorithm in the `submit_order` script to determine the price and conduct the transaction:

```
1  if (pay = yes) {
2       price = lookup(session-id);
3       if (price != NULL) {
4            success = authorize_credit_card_charge(price);
5            if (success) {
6                 settle_transaction(price);
7                 dispatch_delivery_person();
8            }
9            else {
10                // Could not authorize card
11                tell_user_card_declined();
12           }
13      }
14      else {
15           // Cannot find session
16           display_transaction_cancelled_page();
17           log_client_IP_and_info();
18      }
19 }
20 else {
21      // pay = no
22      display_transaction_cancelled_page();
23 }
```

If the user clicks "yes," then the server looks up the session-id in the database in line 2. If the session-id is present in the database, then the corresponding price will be returned.

The price never leaves the server, and the client does not have the opportunity to alter it. The database table stores the authoritative state, and the session-id effectively serves as a "pointer" to the client's state.

If for any reason the session-id is not present in the database, `lookup()` returns `NULL`, and the transaction is cancelled, just as if the user had clicked "no." Then, the client's IP address, and any other "forensic" information that appears in the HTTP request, is logged. The cause of the missing session-id might be benign, but if you see a large number of requests with invalid session-ids, it may be an indication that an attacker is at play trying to guess a valid session-id.

In this solution, it is important for the session-id to be difficult for an attacker to guess. If an attacker were able to guess valid session-ids, he might be able to manipulate the state of a transaction. In our simple example, the attacker could, for instance, issue HTTP requests for session-ids with `pay=yes` even though the client may have wanted to cancel the transaction. Also, in a real application, some additional state that might need to be kept in the database includes the customer's address, the quantity of pizzas, the user's credit card number, and other transaction details. If the attacker could guess session-ids, he may be able to modify an existing order to include additional pizzas to be sent to his address, but have the legitimate customer's credit card charged for the transaction! By choosing a 128-bit randomly generated session-id, you limit the attacker's probability of success to $n / 2^{128}$, where n is the number of session-ids in the server's database.

To further minimize the ability of the attacker to guess session-ids, you can have session-ids "timeout," or expire after some time period. For instance, you might decide that anyone who starts ordering a pizza should be able to complete their order within a *k*-minute period (even if they happen to be very indecisive between choosing anchovies or pineapples for toppings). If the user does not complete the order in *k* minutes, you have the right to just forget about their order. You could add an additional column to the database table that records the date and time when the session-id was created. When an HTTP request with a session-id is received, you can check to see if the session-id is more than *k* minutes old. If it is, you erase all the state associated with that session-id before processing the client's request, and/or require that client use a new session-id. Now, the probability that an attacker will be able to guess a valid client ID (that has not expired) is n_k / 2^{128}, where n_k is the number of clients that issued HTTP requests in the last *k* minutes. The only remaining problem is that lots of session-ids might sit around in your database for a long time until they happen to get used again. To deal with this problem, you can periodically (once every few minutes) clear out all the expired session-ids.

Another technique that can be used to make it even harder for attackers to use guessed session-ids is to have the session-id be the "hash" of a pseudo-random number and the IP address that the web server reports the client is connected from. (See Chapters 14 and 15 for a discussion of pseudo-random numbers and hash functions, respectively.) If you use this technique, an attacker not only needs to guess a valid session-id, but also needs to spoof the IP address of the client in order to use the session-id.

The process by which session-ids are provided to clients, associated with state, verified, and invalidated is often referred to as *session management*. Doing session management correctly and securely is challenging. We have only discussed the very basic ideas here. It is typically best to reuse existing session management code in web application frameworks such as Java Servlets/JSP (`http://java.sun.com/products/jsp/docs.html`) (Jorelid 2001) or ASP (Homer and Sussman 2003). Nevertheless, there have been attacks published against the session management functionality in some of these frameworks as well (Gutterman and Malkhi 2005).

By storing authoritative state in a database and never giving the client access to it, you can thwart client-state manipulation attacks. The downside of using a database is that your server-side infrastructure is no longer stateless. Every time an HTTP request arrives at your web server, a database lookup needs to be done, and could turn the database into a performance bottleneck. In addition, if the database lookup takes nontrivial computational resources, an attacker could issue many such requests with random session-ids as part of a DoS attack. The server would be forced to look up each of the session-ids to determine if the request is from a legitimate user, but the act of doing the lookups could overload the server and prevent it from responding to legitimate clients. To deal with this bottleneck, the database can be distributed and HTTP requests can be load balanced across distributed database servers.

7.1.3. Solution 2: Signed State Sent to Client

We now outline another solution in which the server can continue to be stateless. In this solution, the authoritative state is returned to the client—but to prevent a client from tampering with the state, a "signature" is also sent to the client with the transaction state. If the client attempts to alter the state, the signature will no longer match, and the server will disregard the client's request. In our solution, the server possesses a cryptographic key known only to it that it uses to produce the signature. The client will not be able produce modified signatures to match the altered state, because it does not know the server's key.

When a client fills out an order form, the server sends back a form that includes all the parameters of the transaction (including the price) and a signature:

```
1  <HTML>
2  <HEAD>
3  <TITLE>Pay for Pizza</TITLE>
4  </HEAD>
5  <BODY>
6  <FORM ACTION="submit_order" METHOD="GET">
7  The total cost is 5.50.
8  Are you sure you would like to order?
9  <INPUT TYPE="hidden" NAME="item-id" VALUE="1384634">
10 <INPUT TYPE="hidden" NAME="qty" VALUE="1">
11 <INPUT TYPE="hidden" NAME="address"
12        VALUE="123 Main St, Stanford, CA">
13 <INPUT TYPE="hidden" NAME="credit_card_no"
14        VALUE="5555 1234 4321 9876">
15 <INPUT TYPE="hidden" NAME="exp_date" VALUE="1/2012">
16 <INPUT TYPE="hidden" NAME="price" VALUE="5.50">
17 <INPUT TYPE="hidden" NAME="signature"
18        VALUE="a2a30984f302c843284e9372438b33d2">
19 <INPUT TYPE="submit" NAME ="pay" VALUE="yes">
20 <INPUT TYPE="submit" NAME ="pay" VALUE="no">
21 </BODY>
22 </HTML>
```

This form has more data than the previous one, and we have added this data because it is more essential to the solution. The signature in line 18 was generated by computing a message authentication code (MAC) over all the other parameters of the transaction, including the item-id, quantity, address, credit card number, expiration date, and price. The MAC is also a function of a cryptographic key only known to the server. (MACs were introduced in Section 1.5, and are covered in more detail in Section 15.2.) If the client attempts to change the price or any of the other parameters, the client will not be able to recompute a corresponding signature because it does not know the key.

After the client submits the form, the server uses the following algorithm to process the client's request:[3]

```
if (pay = yes) {
     // Aggregate transaction state parameters
     // Note: | is the concatenation operator
```

3. In the code, a delimiter (#) is used in constructing the signature to distinguish, for instance, the case in which a quantity of 1 for address 110 Main St is submitted from the case in which a quantity of 11 for the address 10 Main St is submitted.

```
     // and # is a delimiter.
     state = item-id |  # | qty | # | address | # |
             credit_card_no | # | exp_date | # | price;

     // Compute message authentication code with
     // server key K.
     signature_check = MAC(K, state);
     if (signature == signature_check) {
          success = authorize_credit_card_charge(price);
          if (success) {
               settle_transaction(price);
               dispatch_delivery_person();
          }
          else {
               // Could not authorize card
               tell_user_card_declined();
          }
     }
     else {
          // Invalid signature
          display_transaction_cancelled_page();
          log_client_IP_and_info();
     }
}
else {
     display_transaction_cancelled_page();
}
```

If the user clicks "yes" when asked to confirm the transaction, the server first verifies the signature. The signature is verified by computing `signature_check`. The algorithm concatenates all the relevant pieces of state information into `state`. Then, it computes the MAC over the `state` using the server's key. If `signature_check` matches the `signature` provided in the HTTP request, then the request has not been tampered with, and the algorithm proceeds with credit card authorization. If the `signature_check` does not match the `signature` provided in the HTTP request, then the client may have tried to alter one or more of the parameters. Even though the parameters are sent to the client "in the clear," the server will be able to reliably detect if the client sent back different or altered parameters.

By using this signature-based approach, the server does not need to keep track of session-ids. It can continue to be stateless at the expense of having to compute MACs when processing HTTP requests and having to stream state information to and from the client. At the same time, for state-intensive applications, the amount of extra bandwidth required to stream state may be more costly than the server-side storage required for user data in a session-id–based solution.

SIGN IT ALL!

One caveat to using a signature-based approach is that the *entire* transaction state must be signed—not just part of it (such as the price). Otherwise, an attacker can conduct (part of) a legitimate transaction to coerce the server into generating a signature for her, and she can then conduct an illegitimate transaction by pasting in parameters of her choice that are not included in the signature. For instance, if only the price is signed, the attacker can go through the order process having selected a cheap item to obtain a signature on the price, and then submit that signature and price in an HTTP request to purchase a more expensive item.

7.2. Using HTTP POST Instead of GET

In previous sections, the server embedded session-ids and state in hidden form fields. In order for that state to be relayed back to the server on each subsequent HTTP request, the form field parameters and values need to be included in URLs. In the previous example, we used the GET method with hidden form fields, but we could have just as easily used links as follows:

```
<A HREF=/submit_order? ➥
session-id=3927a837e947df203784d309c8372b8e> ➥
Pay Now</A>
```

or

```
<A HREF=/submit_order? ➥
session-id=3927a837e947df203784d309c8372b8e> ➥
Cancel Order</A>
```

Using hidden form fields can be an awkward way to carry state from one step in a web transaction to the next. Consider the case in which you use a database at the server to maintain the state of the user's transaction. After the user enters the number of pizzas he would like to order, along with his credit card number and expiration date, he receives an order confirmation page. The URL in the address bar of the browser for the confirmation page might read as follows:

```
https://www.deliver-me-pizza.com/confirm_order? ➥
session-id=3927a837e947df203784d309c8372b8e
```

If a user, Alice, copies the preceding address and pastes it into an e-mail to her "friend" Meg asking, "Hey Meg, should we order this pizza?" then Meg would be able to click "yes" and continue the transaction without Alice's consent—nevertheless, the pizza would be charged to Alice's credit card. Depending upon how the web site was implemented, it could also be possible for Meg to change the address to which the pizza is sent to be her own address. Meg could then respond to Alice, saying "No, I don't think we should order the pizza. Maybe next time." Meg would get to eat pizza that was ordered using Alice's credit card. Of course, Alice's credit card would be charged, and when she receives her credit card bill at the end of the month, she will call her credit card company and complain that she did not order the pizza. Of course, her "friend" Meg did!

Another reason not to use GET has to do with HTTP referrer fields. When a user clicks on a hyperlink in a web page on web site A, and is referred to web site B, the browser's request to web site B usually includes an HTTP header that lets web site B know that the user came from web site A. For instance, after processing the purchase of a pizza, the submit_order program could output a link to a grocery store web site that has information about frozen versions of their pizzas. The HTML outputted by the submit_order script might look as follows:

```
<HTML>
<HEAD>
<TITLE>Pizza Order Complete</TITLE>
</HEAD>
<BODY>
Thank you for your pizza order.
It will arrive piping hot
within 30 to 45 minutes!
<A HREF=confirm_order?
session-id=3927a837e947df203784d309c8372b8e>
Click here to order one more pizza!
</A>
You may also be interested in trying
our frozen pizzas at
<A HREF=http://www.grocery-store-site.com/>
GroceryStoreSite
</A>
</BODY>
</HTML>
```

This web page includes two hyperlinks. The first is the one that allows the user to purchase another pizza. Since the user has already entered her credit card number, the pizza web site could accept another order without requiring her to enter her card number again. Granted, some might consider that a bad design, but nevertheless it might lead to some extra orders. To facilitate the order, the session-id is included in the hyperlink. The second hyperlink is to www.grocery-store-site.com.

Note that the URL for the preceding web page is

```
https://www.deliver-me-pizza.com/submit_order? ➥
session-id=3927a837e947df203784d309c8372b8e
```

If the user, instead of ordering another pizza, is interested in the frozen pizzas from the grocery store, he may decide to click the second link, to www.grocery-store-site.com. The HTTP request to www.grocery-store-site.com would be as follows:

```
GET / HTTP/1.0 Referer: https://www.deliver-me-pizza.com/submit_order? ➥
session-id=3927a837e947df203784d309c8372b8e
```

When www.grocery-store-site.com's web server receives the preceding request, it will serve its index.html page, and will also log the referrer field. Note that the user's session-id from www.deliver-me-pizza.com gets stored in www.grocery-store-site.com's logs! If the administrator of www.grocery-store-site.com's web server is malicious, she could paste the URL from the referrer field into her browser and order additional pizzas! (To make it interesting, as before, the administrator would change the address to be her own but still use the user's credit card number.)

To prevent users from exchanging URLs with each other in dangerous ways and having sensitive information show up in web logs of other web sites, you could use POST as the HTTP method by which to submit the form to remove the session-id from the URL. The revised order confirmation form would use the following form action tag, in which the form uses the HTTP POST method instead of GET:

```
<FORM ACTION="confirm_order" METHOD="POST">
```

When the form is submitted, the HTTP request might look as follows:

```
POST /confirm_order HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 45

session-id%3D3927a837e947df203784d309c8372b8e
```

The URL in the address bar would simply read https://www.deliver-me-pizza.com/confirm_order, with no session-id included in it.

If Alice were to paste the preceding URL into an e-mail sent to Meg, then Meg would not be able to see the same confirmation form that Alice does because the session-id is not included in the URL. Alice might now instead have to send a screenshot of the pizza order screen to Meg to share the details of the order. While that might be more inconvenient for Alice, using the POST method prevents her from simply sending a URL to Meg that would allow Meg to place the order without Alice's consent.

While POST can sometimes be used to prevent the type of information leakage shown previously, referrers can also leak in other ways that do not require any user interaction. For instance, if instead of a link to www.grocery-store-site.com on the order completion page, an image tag such as <IMG SRC=http://www.grocery-store-site.com/banner.gif> were included, the referrer URL with the session-id would appear in www.grocery-store-site.com's logs due to the GET request for banner.gif.

7.3. Cookies

An alternative to using HTTP POST to maintain state across HTTP requests would be to use cookies. A *cookie* is a piece of state that is maintained by a client. When a web server gives a cookie to a client browser, that client browser is expected to give the cookie back to the server in subsequent HTTP requests. However, since web servers cannot, in general, trust web clients, web servers do not have any guarantees that a web client will return the cookie that it was given. We cover cookies here because they are an alternative to transmitting session-ids and other authentication credentials.

A web server gives a cookie to a client by including a Set-Cookie field in an HTTP response. To illustrate, consider the preceding solution, in which we used session-ids to serve as pointers to state in the web server's database. Instead of sending the session-id from the server to the client as a hidden form field, we could have sent a cookie as follows:

```
HTTP/1.0 200 OK
Set-Cookie: session-id=3927a837e947df203784d309c8372b8e; secure

<HTML>
<HEAD>
<TITLE>Pay for Pizza</TITLE>
</HEAD>
<BODY>
<FORM ACTION="submit_order" METHOD="GET">
The total cost is 5.50.
Are you sure you would like to order?
<INPUT TYPE="submit" NAME ="pay" VALUE="yes">
<INPUT TYPE="submit" NAME ="pay" VALUE="no">
</BODY>
</HTML>
```

We show the HTTP response header in addition to the confirmation form issued by the server to illustrate the use of the Set-Cookie HTTP response header field, whereas we typically did not need to show the HTTP response header in previous examples. Note that in addition to the session-id specified in the cookie, a secure attribute is used to specify that the client should only send the cookie back to the server over an SSL connection.

In the preceding example, when the user clicks the submit button, the browser sends the following HTTP request to the server:

```
GET /submit_order?pay=yes HTTP/1.0
Cookie: session-id=3927a837e947df203784d309c8372b8e
```

The algorithm that the server uses to process the order is still more or less the same, except that the value of the session-id variable is retrieved from the cookie instead of the URL parameters.

In general, using a cookie is different from using hidden form fields because a well-behaved browser will typically send the cookie back to the web server on each HTTP request, without requiring a form submission or "tacking on" additional parameters to the URL.

You must be careful when using cookies since they are stored by the browser. If Alice uses your web site and does not explicitly log out (or you do not expire the session-id after some time period), there may be an additional security risk. If Mallory can use Alice's browser to visit the same site, the browser may send back the cookie, and Mallory may be able to impersonate Alice. Hence, it is extremely important to make sure that session-ids have a limited lifetime by associating an expiration time with them on the server, and providing users with the ability to explicitly log out.

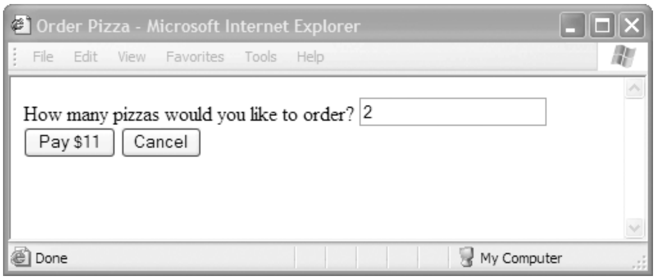
7.4. JavaScript

JavaScript is a scripting language that can be used to write scripts that interact with web pages. JavaScript is a language that is separate and distinct from Java, but derives its name from its Java-like syntax. JavaScript code can be included within HTML web pages, and the code is executed by a JavaScript interpreter once downloaded to the web browser. We cover JavaScript in this chapter for two reasons: (1) sometimes programmers rely on JavaScript for tasks that they should not, and (2) sometimes attackers can use JavaScript to help construct attacks. We illustrate how using JavaScript carelessly can give rise to a security vulnerability in this section, and provide a description of how attackers can use JavaScript to construct more attacks in Chapter 10.

In the following example, we show some JavaScript that can be used to help compute the price of an order:

```
<HTML>
<HEAD>
<TITLE>Order Pizza</TITLE>
</HEAD>
<BODY>
<FORM ACTION="submit_order" METHOD="GET" NAME="f">
How many pizzas would you like to order?
<INPUT TYPE="text" NAME="qty" VALUE="1" onKeyUp="computePrice();">
<INPUT TYPE="hidden" NAME="price" VALUE="5.50"><BR>
<INPUT TYPE="submit" NAME ="Order" VALUE="Pay">
<INPUT TYPE="submit" NAME ="Cancel" VALUE="Cancel">
<SCRIPT>
function computePrice() {
     f.price.value = 5.50 * f.qty.value;
     f.Order.value = "Pay $" + f.price.value
}
</SCRIPT>
</BODY>
</HTML>
```

The preceding pizza order form looks similar to ones used earlier in the chapter, with just a few differences that help the browser compute the price of the order. First, the form has been given a name attribute that specifies that its name is `f`. The form is given a name so that JavaScript code elsewhere in the HTML can refer to the components of the form, such as the text field `qty`, which contains the user-specified number of pizzas to order; and the submit button named `Order`, which the user can click to execute the order. Second, an `onKeyUp` “handler” has been added to the `qty` text field. The `onKeyUp` handler tells the browser to call the `computePrice()` JavaScript function whenever the user has made a change to the `qty` text field. Third, the definition of the `computePrice()` JavaScript function has been included in the HTML using the `<SCRIPT>` tag. The `computePrice()` function first updates the value of the hidden `price` field based on the quantity the user has selected, and then updates the order submit button to read “Pay $*X*,” where *X* is the computed price. The page rendered by the browser for the preceding HTML and JavaScript code is shown in Figure 7-3.


Order Pizza - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Done
My Computer

How many pizzas would you like to order? 2
Pay $11 Cancel

Figure 7-3. *JavaScript HTML order page*

In the preceding example, the client browser computes the price to be paid based on the number of pizzas the user would like to order. However, as you learned before, you cannot trust the client! A malicious user could simply save the HTML page to disk (as we illustrated earlier in this chapter), delete the JavaScript from the HTML page, substitute 10000 for the quantity and 0 for the price, and submit the form. Alternatively, a malicious user could also just submit an HTTP request such as

```
GET /submit_order?qty=1000&price=0&Order=Pay
```

and completely bypass the price computation done by the JavaScript! The solution to eliminating the problem of not being able to trust the client, in this case, is to do the price computation on the server, and charge the user the price that is computed by the server. While JavaScript can be used to make the web page more interactive for the client, any data validation or computations done by the JavaScript cannot be trusted by the server. The computations must be redone on the server to ensure security.