

Bypassing Client-Side Controls

Chapter 1 described how the core security problem with web applications arises because clients can submit arbitrary input. Despite this fact, a large proportion of web applications, nevertheless, rely on various measures implemented on the client side to control the data that they submit to the server. In general, this represents a fundamental security flaw: the user has full control over the client and the data it submits and can bypass any controls that are implemented on the client side and are not replicated on the server.

An application may rely on client-side controls to restrict user input in two broad ways. First, an application may transmit data via the client component using a mechanism that it assumes will prevent the user from modifying that data when the application later reads it. Second, an application may implement measures on the client side that control the user's interaction with his or her own client, with the aim of restricting functionality and/or applying controls around user input before it is submitted. This may be achieved using HTML form features, client-side scripts, or browser extension technologies.

This chapter looks at examples of each kind of client-side control and describes ways in which they can be bypassed.

Transmitting Data Via the Client

It is common to see an application passing data to the client in a form that the end user cannot directly see or modify, with the expectation that this data will be sent back to the server in a subsequent request. Often, the application's developers simply assume that the transmission mechanism used will ensure that the data transmitted via the client will not be modified along the way.

Because everything submitted from the client to the server is within the user's control, the assumption that data transmitted via the client will not be modified is usually false and often leaves the application vulnerable to one or more attacks.

You may reasonably wonder why, if the server knows and specifies a particular item of data, the application would ever need to transmit this value to the client and then read it back. In fact, writing applications in this way is often easier for developers for various reasons:

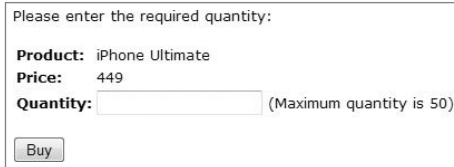
- It removes the need to keep track of all kinds of data within the user's session. Reducing the amount of per-session data being stored on the server can also improve the application's performance.
- If the application is deployed on several distinct servers, with users potentially interacting with more than one server to perform a multistep action, it may not be straightforward to share server-side data between the hosts that may handle the same user's requests. Using the client to transmit data can be a tempting solution to the problem.
- If the application employs any third-party components on the server, such as shopping carts, modifying these may be difficult or impossible, so transmitting data via the client may be the easiest way of integrating these.
- In some situations, tracking a new piece of data on the server may entail updating a core server-side API, thereby triggering a full-blown formal change-management process and regression testing. Implementing a more piecemeal solution involving client-side data transmission may avoid this, allowing tight deadlines to be met.

However, transmitting sensitive data in this way is usually unsafe and has been the cause of countless vulnerabilities in applications.

Hidden Form Fields

Hidden HTML form fields are a common mechanism for transmitting data via the client in a superficially unmodifiable way. If a field is flagged as hidden, it is not displayed on-screen. However, the field's name and value are stored within the form and are sent back to the application when the user submits the form.

The classic example of this security flaw is a retailing application that stores the prices of products within hidden form fields. In the early days of web applications, this vulnerability was extremely widespread, and by no means has it been eliminated today. Figure 5-1 shows a typical form.



Please enter the required quantity:

Product: iPhone Ultimate

Price: 449

Quantity: (Maximum quantity is 50)

Figure 5-1: A typical HTML form

The code behind this form is as follows:

```
<form method="post" action="Shop.aspx?prod=1">
Product: iPhone 5 <br/>
Price: 449 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="hidden" name="price" value="449">
<input type="submit" value="Buy">
</form>
```

Notice the form field called `price`, which is flagged as hidden. This field is sent to the server when the user submits the form:

```
POST /shop/28/Shop.aspx?prod=1 HTTP/1.1
Host: mdsec.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 20

quantity=1&price=449
```

TRY IT!

<http://mdsec.net/shop/28/>

Although the `price` field is not displayed on-screen, and the user cannot edit it, this is solely because the application has instructed the browser to hide the field. Because everything that occurs on the client side is ultimately within the user's control, this restriction can be circumvented to edit the price.

One way to achieve this is to save the source code for the HTML page, edit the field's value, reload the source into a browser, and click the Buy button. However, an easier and more elegant method is to use an intercepting proxy to modify the desired data on-the-fly.

An intercepting proxy is tremendously useful when attacking a web application and is the one truly indispensable tool you need. Numerous such tools are available. We will use Burp Suite, which was written by one of this book's authors.

The proxy sits between your web browser and the target application. It intercepts every request issued to the application, and every response received back, for both HTTP and HTTPS. It can trap any intercepted message for inspection or modification by the user. If you haven't used an intercepting proxy before, you can read more about how they function, and how to get them configured and working, in Chapter 20.

Once an intercepting proxy has been installed and suitably configured, you can trap the request that submits the form and modify the `price` field to any value, as shown in Figure 5-2.

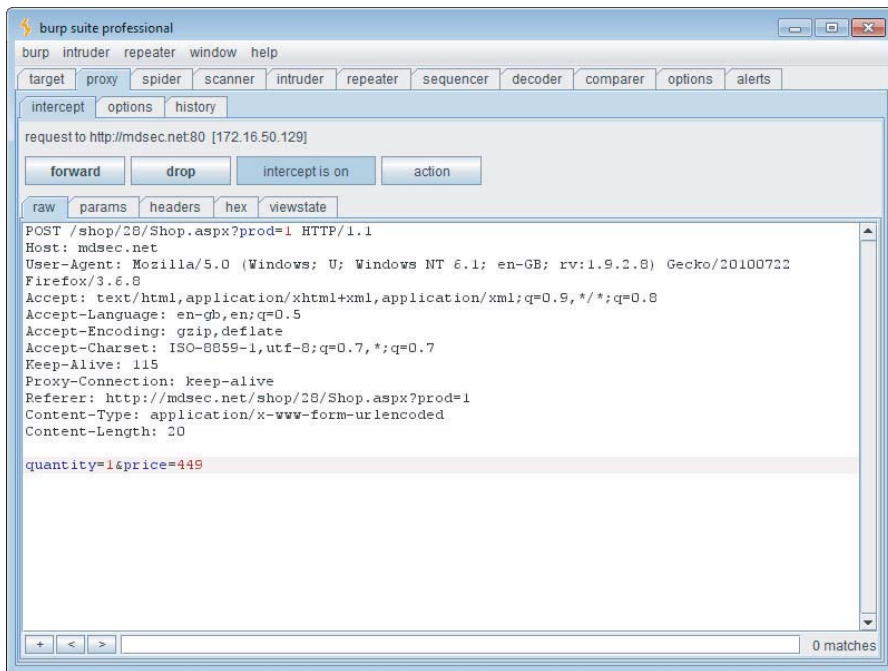


Figure 5-2: Modifying the values of hidden form fields using an intercepting proxy

If the application processes the transaction based on the price submitted, you can purchase the product for the price of your choice.

TIP If you find an application that is vulnerable in this way, see whether you can submit a negative amount as the price. In some cases, applications have actually accepted transactions using negative prices. The attacker receives a refund to his credit card and also the item he ordered – a win-win situation, if ever there was one.

HTTP Cookies

Another common mechanism for transmitting data via the client is HTTP cookies. As with hidden form fields, normally these are not displayed on-screen, and the user cannot modify them directly. They can, of course, be modified using an intercepting proxy, by changing either the server response that sets them or subsequent client requests that issue them.

Consider the following variation on the previous example. After the customer has logged in to the application, she receives the following response:

```
HTTP/1.1 200 OK
Set-Cookie: DiscountAgreed=25
Content-Length: 1530
...
```

This `DiscountAgreed` cookie points to a classic case of relying on client-side controls (the fact that cookies normally can't be modified) to protect data transmitted via the client. If the application trusts the value of the `DiscountAgreed` cookie when it is submitted back to the server, customers can obtain arbitrary discounts by modifying its value. For example:

```
POST /shop/92/Shop.aspx?prod=3 HTTP/1.1
Host: mdsec.net
Cookie: DiscountAgreed=25
Content-Length: 10

quantity=1
```

TRY IT!

```
http://mdsec.net/shop/92/
```

URL Parameters

Applications frequently transmit data via the client using preset URL parameters. For example, when a user browses the product catalog, the application may provide him with hyperlinks to URLs like the following:

```
http://mdsec.net/shop/?prod=3&pricecode=32
```

When a URL containing parameters is displayed in the browser's location bar, any parameters can be modified easily by any user without the use of tools. However, in many instances an application may expect that ordinary users cannot view or modify URL parameters:

- Where embedded images are loaded using URLs containing parameters
- Where URLs containing parameters are used to load a frame's contents

- Where a form uses the `POST` method and its target URL contains preset parameters
- Where an application uses pop-up windows or other techniques to conceal the browser location bar

Of course, in any such case the values of any URL parameters can be modified as previously discussed using an intercepting proxy.

The Referer Header

Browsers include the `Referer` header within most HTTP requests. It is used to indicate the URL of the page from which the current request originated — either because the user clicked a hyperlink or submitted a form, or because the page referenced other resources such as images. Hence, it can be leveraged as a mechanism for transmitting data via the client. Because the URLs processed by the application are within its control, developers may assume that the `Referer` header can be used to reliably determine which URL generated a particular request.

For example, consider a mechanism that enables users to reset their password if they have forgotten it. The application requires users to proceed through several steps in a defined sequence before they actually reset their password's value with the following request:

```
GET /auth/472/CreateUser.ashx HTTP/1.1
Host: mdsec.net
Referer: https://mdsec.net/auth/472/Admin.ashx
```

The application may use the `Referer` header to verify that this request originated from the correct stage (`Admin.ashx`). If it did, the user can access the requested functionality.

However, because the user controls every aspect of every request, including the HTTP headers, this control can be easily circumvented by proceeding directly to `CreateUser.ashx` and using an intercepting proxy to change the value of the `Referer` header to the value that the application requires.

The `Referer` header is strictly optional according to w3.org standards. Hence, although most browsers implement it, using it to control application functionality should be regarded as a hack.

TRY IT!

```
http://mdsec.net/auth/472/
```

COMMON MYTH

It is often assumed that HTTP headers are somehow more “tamper-proof” than other parts of the request, such as the URL. This may lead developers to implement functionality that trusts the values submitted in headers such as `Cookie` and `Referer` while performing proper validation of other data such as URL parameters. However, this perception is false. Given the multitude of intercepting proxy tools that are freely available, any amateur hacker who targets an application can change all request data with ease. It is rather like supposing that when the teacher comes to search your desk, it is safer to hide your water pistol in the bottom drawer, because she will need to bend down farther to discover it.

HACK STEPS

1. Locate all instances within the application where hidden form fields, cookies, and URL parameters are apparently being used to transmit data via the client.
2. Attempt to determine or guess the role that the item plays in the application’s logic, based on the context in which it appears and on clues such as the parameter’s name.
3. Modify the item’s value in ways that are relevant to its purpose in the application. Ascertain whether the application processes arbitrary values submitted in the parameter, and whether this exposes the application to any vulnerabilities.

Opaque Data

Sometimes, data transmitted via the client is not transparently intelligible because it has been encrypted or obfuscated in some way. For example, instead of seeing a product’s price stored in a hidden field, you may see a cryptic value being transmitted:

```
<form method="post" action="Shop.aspx?prod=4">
Product: Nokia Infinity <br/>
Price: 699 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="hidden" name="price" value="699">
<input type="hidden" name="pricing_token"
value="E76D213D291B8F216D694A34383150265C989229">
<input type="submit" value="Buy">
</form>
```

When this is observed, you may reasonably infer that when the form is submitted, the server-side application checks the integrity of the opaque string, or even decrypts or deobfuscates it to perform some processing on its plaintext value. This further processing may be vulnerable to any kind of bug. However, to probe for and exploit this, first you need to wrap up your payload appropriately.

TRY IT!

```
http://mdsec.net/shop/48/
```

NOTE Opaque data items transmitted via the client are often part of the application's session-handling mechanism. Session tokens sent in HTTP cookies, anti-CSRF tokens transmitted in hidden fields, and one-time URL tokens for accessing application resources, are all potential targets for client-side tampering. Numerous considerations are specific to these kinds of tokens, as discussed in depth in Chapter 7.

HACK STEPS

Faced with opaque data being transmitted via the client, several avenues of attack are possible:

1. If you know the value of the plaintext behind the opaque string, you can attempt to decipher the obfuscation algorithm being employed.
2. As described in Chapter 4, the application may contain functions elsewhere that you can leverage to return the opaque string resulting from a piece of plaintext you control. In this situation, you may be able to directly obtain the required string to deliver an arbitrary payload to the function you are targeting.
3. Even if the opaque string is impenetrable, it may be possible to replay its value in other contexts to achieve a malicious effect. For example, the `pricing_token` parameter in the previously shown form may contain an encrypted version of the product's price. Although it is not possible to produce the encrypted equivalent for an arbitrary price of your choosing, you may be able to copy the encrypted price from a different, cheaper product and submit this in its place.
4. If all else fails, you can attempt to attack the server-side logic that will decrypt or deobfuscate the opaque string by submitting malformed variations of it – for example, containing overlong values, different character sets, and the like.

The ASP.NET ViewState

One commonly encountered mechanism for transmitting opaque data via the client is the ASP.NET `ViewState`. This is a hidden field that is created by default in all ASP.NET web applications. It contains serialized information about the

state of the current page. The ASP.NET platform employs the `ViewState` to enhance server performance. It enables the server to **preserve** elements within the user interface across successive requests without needing to maintain all the relevant state information on the server side. For example, the server may populate a drop-down list on the basis of parameters submitted by the user. When the user makes subsequent requests, the browser does not submit the contents of the list back to the server. However, the browser does submit the hidden `ViewState` field, which contains a serialized form of the list. The server deserializes the `ViewState` and recreates the same list that is presented to the user again.

In addition to this core purpose of the `ViewState`, developers can use it to store arbitrary information across successive requests. For example, instead of saving the product's price in a hidden form field, an application may save it in the `ViewState` as follows:

```
string price = getPrice(prodno);
ViewState.Add("price", price);
```

The form returned to the user now looks something like this:

```
<form method="post" action="Shop.aspx?prod=3">
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwULLTE1ODcxNjkwNjIPFgIeBXByaWNlBQMzOTlkZA==" />
Product: HTC Avalanche <br/>
Price: 399 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="submit" value="Buy">
</form>
```

When the user submits the form, her browser sends the following:

```
POST /shop/76/Shop.aspx?prod=3 HTTP/1.1
Host: mdsec.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 77

__VIEWSTATE=%2FwEPDwULLTE1ODcxNjkwNjIPFgIeBXByaWNlBQMzOTlkZA%3D%3D&
quantity=1
```

The request apparently does not contain the product price — only the quantity ordered and the opaque `ViewState` parameter. Changing that parameter at random results in an error message, and the purchase is not processed.

The `ViewState` parameter is actually a Base64-encoded string that can be easily decoded to see the price parameter that has been placed there:

```
3D FF 01 0F 0F 05 0B 2D 31 35 38 37 31 36 39 30 ; =ÿ.....-15871690
36 32 0F 16 02 1E 05 70 72 69 63 65 05 03 33 39 ; 62.....price..39
39 64 64 ; 9dd
```


TRY IT!

```
http://mdsec.net/shop/76/
```

HACK STEPS

1. If you are attacking an ASP.NET application, verify whether MAC protection is enabled for the `ViewState`. This is indicated by the presence of a 20-byte hash at the end of the `ViewState` structure, and you can use the `ViewState` parser in Burp Suite to confirm whether this is present.
2. Even if the `ViewState` is protected, use Burp to decode the `ViewState` on various application pages to discover whether the application is using the `ViewState` to transmit any sensitive data via the client.
3. Try to modify the value of a specific parameter within the `ViewState` without interfering with its structure, and see whether an error message results.
4. If you can modify the `ViewState` without causing errors, you should review the function of each parameter within the `ViewState` and see whether the application uses it to store any custom data. Try to submit crafted values as each parameter to probe for common vulnerabilities, as you would for any other item of data being transmitted via the client.
5. Note that MAC protection may be enabled or disabled on a per-page basis, so it may be necessary to test each significant page of the application for `ViewState` hacking vulnerabilities. If you are using Burp Scanner with passive scanning enabled, Burp automatically reports any pages that use the `ViewState` without MAC protection enabled.

Capturing User Data: HTML Forms

The other principal way in which applications use client-side controls to restrict data submitted by clients occurs with data that was not originally specified by the server but that was gathered on the client computer itself.

HTML forms are the simplest and most common way to capture input from the user and submit it to the server. With the most basic uses of this method, users type data into named text fields, which are submitted to the server as name/value pairs. However, forms can be used in other ways; they can impose restrictions or perform validation checks on the user-supplied data. When an

application employs these client-side controls as a security mechanism to defend itself against malicious input, the controls can usually be easily circumvented, leaving the application potentially vulnerable to attack.

Length Limits

Consider the following variation on the original HTML form, which imposes a maximum length of 1 on the quantity field:

```
<form method="post" action="Shop.aspx?prod=1">
Product: iPhone 5 <br/>
Price: 449 <br/>
Quantity: <input type="text" name="quantity" maxlength="1"> <br/>
<input type="hidden" name="price" value="449">
<input type="submit" value="Buy">
</form>
```

Here, the browser prevents the user from entering more than one character into the input field, so the server-side application may assume that the `quantity` parameter it receives will be less than 10. However, this restriction can easily be circumvented either by intercepting the request containing the form submission to enter an arbitrary value, or by intercepting the response containing the form to remove the `maxlength` attribute.

INTERCEPTING RESPONSES

When you attempt to intercept and modify server responses, you may find that the relevant message displayed in your proxy looks like this:

```
HTTP/1.1 304 Not Modified
Date: Wed, 6 Jul 2011 22:40:20 GMT
Etag: "6c7-5fcc0900"
Expires: Thu, 7 Jul 2011 00:40:20 GMT
Cache-Control: max-age=7200
```

This response arises because the browser already possesses a cached copy of the resource it requested. When the browser requests a cached resource, it typically adds two headers to the request — `If-Modified-Since` and `If-None-Match`:

```
GET /scripts/validate.js HTTP/1.1
Host: wahn-app.com
If-Modified-Since: Sat, 7 Jul 2011 19:48:20 GMT
If-None-Match: "6c7-5fcc0900"
```

These headers tell the server when the browser last updated its cached copy. The `Etag` string, which the server provided with that copy of the resource, is a kind of serial number that the server assigns to each cacheable resource.

It updates each time the resource is modified. If the server possesses a newer version of the resource than the date specified in the `If-Modified-Since` header, or if the `Etag` of the current version matches the one specified in the `If-None-Match` header, the server responds with the latest version of the resource. Otherwise, it returns a 304 response, as shown here, informing the browser that the resource has not been modified and that the browser should use its cached copy.

When this occurs, and you need to intercept and modify the resource that the browser has cached, you can intercept the relevant request and remove the `If-Modified-Since` and `If-None-Match` headers. This causes the server to respond with the full version of the requested resource. Burp Proxy contains an option to strip these headers from every request, thereby overriding all cache information sent by the browser.

HACK STEPS

1. Look for form elements containing a `maxlength` attribute. Submit data that is longer than this length but that is formatted correctly in other respects (for example, it is numeric if the application expects a number).
2. If the application accepts the overlong data, you may infer that the client-side validation is not replicated on the server.
3. Depending on the subsequent processing that the application performs on the parameter, you may be able to leverage the defects in validation to exploit other vulnerabilities, such as SQL injection, cross-site scripting, or buffer overflows.

Script-Based Validation

The input validation mechanisms built into HTML forms themselves are extremely simple and are insufficiently fine-grained to perform relevant validation of many kinds of input. For example, a user registration form might contain fields for name, e-mail address, telephone number, and zip code, all of which expect different types of input. Therefore, it is common to see customized client-side input validation implemented within scripts. Consider the following variation on the original example:

```
<form method="post" action="Shop.aspx?prod=2" onsubmit="return  
validateForm(this)">  
Product: Samsung Multiverse <br/>  
Price: 399 <br/>
```

```
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="submit" value="Buy">
</form>

<script>function validateForm(theForm)
{
    var isInteger = /^\d+$/;
    var valid = isInteger.test(quantity) &&
        quantity > 0 && quantity <= 50;
    if (!valid)
        alert('Please enter a valid quantity');
    return valid;
}
</script>
```

TRY IT!

<http://mdsec.net/shop/139/>

The `onsubmit` attribute of the `form` tag instructs the browser to execute the `validateForm` function when the user clicks the Submit button, and to submit the form only if this function returns true. This mechanism enables the client-side logic to intercept an attempted form submission, perform customized validation checks on the user's input, and decide whether to accept that input. In the preceding example, the validation is simple; it checks whether the data entered in the `amount` field is an integer and is between 1 and 50.

Client-side controls of this kind are usually easy to circumvent. Usually it is sufficient to disable JavaScript within the browser. If this is done, the `onsubmit` attribute is ignored, and the form is submitted without any custom validation.

However, disabling JavaScript may break the application if it depends on client-side scripting for its normal operation (such as constructing parts of the user interface). A neater approach is to enter a benign (known good) value into the input field in the browser, intercept the validated submission with your proxy, and modify the data to your desired value. This is often the easiest and most elegant way to defeat JavaScript-based validation.

Alternatively, you can intercept the server's response that contains the JavaScript validation routine and modify the script to neutralize its effect — in the previous example, by changing the `validateForm` function to return true in every case.

HACK STEPS

1. Identify any cases where client-side JavaScript is used to perform input validation prior to form submission.
2. Submit data to the server that the validation ordinarily would have blocked, either by modifying the submission request to inject invalid data or by modifying the form validation code to **neutralize** it.
3. As with length restrictions, determine whether the client-side controls are replicated on the server and, if not, whether this can be exploited for any malicious purpose.
4. Note that if multiple input fields are subjected to client-side validation prior to form submission, you need to test each field individually with invalid data while leaving valid values in all the other fields. If you submit invalid data in multiple fields simultaneously, the server might stop processing the form when it identifies the first invalid field. Therefore, your testing won't reach all possible code paths within the application.

NOTE Client-side JavaScript routines to validate user input are common in web applications, but do not conclude that every such application is vulnerable. The application is exposed only if client-side validation is not replicated on the server, and even then only if crafted input that circumvents client-side validation can be used to cause some undesirable behavior by the application.

In the majority of cases, client-side validation of user input has beneficial effects on the application's performance and the quality of the user experience. For example, when filling out a detailed registration form, an ordinary user might make various mistakes, such as omitting required fields or formatting his telephone number incorrectly. In the absence of client-side validation, correcting these mistakes may entail several reloads of the page and round-trip messages to the server. Implementing basic validation checks on the client side makes the user's experience much smoother and reduces the load on the server.

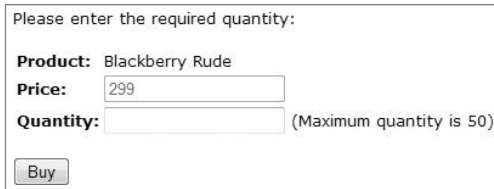
Disabled Elements

If an element on an HTML form is flagged as disabled, it appears on-screen but is usually grayed out and cannot be edited or used in the way an ordinary control can be. Also, it is not sent to the server when the form is submitted. For example, consider the following form:

```
<form method="post" action="Shop.aspx?prod=5">  
Product: Blackberry Rude <br/>  
Price: <input type="text" disabled="true" name="price" value="299">
```

```
<br/>  
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)  
<br/>  
<input type="submit" value="Buy">  
</form>
```

This includes the price of the product as a disabled text field and appears on-screen as shown in Figure 5-4.



Please enter the required quantity:

Product: Blackberry Rude

Price:

Quantity: (Maximum quantity is 50)

Figure 5-4: A form containing a disabled input field

When this form is submitted, only the `quantity` parameter is sent to the server. However, the presence of a disabled field suggests that a `price` parameter may originally have been used by the application, perhaps for testing purposes during development. This parameter would have been submitted to the server and may have been processed by the application. In this situation, you should definitely test whether the server-side application still processes this parameter. If it does, seek to exploit this fact.

TRY IT!

<http://mdsec.net/shop/104/>

HACK STEPS

1. Look for disabled elements within each form of the application. Whenever you find one, try submitting it to the server along with the form's other parameters to determine whether it has any effect.
2. Often, submit elements are flagged as disabled so that buttons appear as grayed out in contexts when the relevant action is unavailable. You should always try to submit the names of these elements to determine whether the application performs a server-side check before attempting to carry out the requested action.

3. **Note that browsers do not include disabled form elements when forms are submitted. Therefore, you will not identify these if you simply walk through the application's functionality, monitoring the requests issued by the browser. To identify disabled elements, you need to monitor the server's responses or view the page source in your browser.**
4. **You can use the HTML modification feature in Burp Proxy to automatically re-enable any disabled fields used within the application.**

Capturing User Data: Browser Extensions

Besides HTML forms, the other main method for capturing, validating, and submitting user data is to use a client-side component that runs in a browser extension, such as Java or Flash. When first employed in web applications, browser extensions were often used to perform simple and often **cosmetic** tasks. Now, companies are increasingly using browser extensions to create fully functional client-side components. These run within the browser, across multiple client platforms, and provide feedback, flexibility, and handling of a desktop application. A side effect is that processing tasks that previously would have taken place on the server may be offloaded onto the client for reasons of speed and user experience. In some cases, such as online trading applications, speed is so critical that much of the key application logic takes place on the client side. The application design may **deliberately sacrifice** security in favor of speed, perhaps in the mistaken belief that traders are trusted users, or that the browser extension includes its own defenses. Recalling the core security problem discussed in Chapter 2, and the earlier sections of this chapter, we know that the concept of a client-side component defending its business logic is impossible.

Browser extensions can capture data in various ways — via input forms and in some cases by interacting with the client operating system's filesystem or registry. They can perform arbitrarily complex validation and manipulation of captured data before submission to the server. Furthermore, because their internal workings are less transparent than HTML forms and JavaScript, developers are more likely to assume that the validation they perform cannot be circumvented. For this reason, browser extensions are often a fruitful target for discovering vulnerabilities within web applications.

A classic example of a browser extension that applies controls on the client side is a casino component. Given what we have observed about the fallible nature of client-side controls, the idea of implementing an online gambling application using a browser extension that runs locally on a potential attacker's

machine is intriguing. If any aspect of the game play is controlled within the client instead of by the server, an attacker could manipulate the game with precision to improve the odds, change the rules, or alter the scores submitted to the server. Several kinds of attacks could occur in this scenario:

- The client component could be trusted to maintain the game state. In this instance, local tampering with the game state would give an attacker an advantage in the game.
- An attacker could bypass a client-side control and perform an illegal action designed to give himself an advantage within the game.
- An attacker could find a hidden function, parameter, or resource that, when invoked, allows illegitimate access to a server-side resource.
- If the game involves any peers, or a house player, the client component could be receiving and processing information about other players that, if known, could be used to the attacker's advantage.

Common Browser Extension Technologies

The browser extension technologies you are most likely to encounter are Java applets, Flash, and Silverlight. Because these are competing to achieve similar goals, they have similar properties in their architecture that are relevant to security:

- They are compiled to an intermediate bytecode.
- They execute within a virtual machine that provides a sandbox environment for execution.
- They may use remoting frameworks employing serialization to transmit complex data structures or objects over HTTP.

Java

Java applets run in the Java Virtual Machine (JVM) and are subject to the sandboxing applied by the Java Security Policy. Because Java has existed since early in the web's history, and because its core concepts have remained relatively unchanged, a large body of knowledge and tools are available for attacking and defending Java applets, as described later in this chapter.

Flash

Flash objects run in the Flash virtual machine, and, like Java applets, are sandboxed from the host computer. Once used largely as a method of delivering animated content, Flash has moved on. With newer versions of ActionScript,

Flash is now squarely billed as capable of delivering full-blown desktop applications. A key recent change in Flash is ActionScript 3 and its remoting capability with Action Message Format (AMF) serialization.

Silverlight

Silverlight is Microsoft's alternative to Flash. It is designed with the similar goal of enabling rich, desktop-like applications, allowing web applications to provide a scaled-down .NET experience within the browser, in a sandboxed environment. Technically, Silverlight applications can be developed in any .NET-compliant language from C# to Python, although C# is by far the most common.

Approaches to Browser Extensions

You need to employ two broad techniques when targeting applications that use browser extension components.

First, you can intercept and modify the requests made by the component and the responses received from the server. In many cases, this is the quickest and easiest way to start testing the component, but you may encounter several limitations. The data being transmitted may be obfuscated or encrypted, or may be serialized using schemes that are specific to the technology being used. By looking only at the traffic generated by the component, you may overlook some key functionality or business logic that can be discovered only by analyzing the component itself. Furthermore, you may encounter obstacles to using your intercepting proxy in the normal way; however, normally these can be circumvented with some careful configuration, as described later in this chapter.

Second, you can target the component itself directly and attempt to decompile its bytecode to view the original source, or interact dynamically with the component using a debugger. This approach has the advantage that, if done thoroughly, you identify all the functionality that the component supports or references. It also allows you to modify key data submitted in requests to the server, regardless of any obfuscation or encryption mechanisms used for data in transit. A disadvantage of this approach is that it can be time-consuming and may require detailed understanding of the technologies and programming languages used within the component.

In many cases, a combination of both these techniques is appropriate. The following sections look at each one in more detail.

Intercepting Traffic from Browser Extensions

If your browser is already configured to use an intercepting proxy, and the application loads a client component using a browser extension, you may see requests from this component passing through your proxy. In some cases, you

don't need to do anything more to begin testing the relevant functionality, because you can intercept and modify the component's requests in the usual way.

In the context of bypassing client-side input validation that is implemented in a browser extension, if the component submits the validated data to the server transparently, this data can be modified using an intercepting proxy in the same way as already described for HTML form data. For example, a browser extension supporting an authentication mechanism might capture user credentials, perform some validation on these, and submit the values to the server as plain-text parameters within the request. The validation can be circumvented easily without performing any analysis or attack on the component itself.

In other cases, you may encounter various obstacles that make your testing difficult, as described in the following sections.

Handling Serialized Data

Applications may serialize data or objects before transmitting them within HTTP requests. Although it may be possible to decipher some of the string-based data simply by inspecting the raw serialized data, in general you need to unpack the serialized data before it can be fully understood. And if you want to modify the data to interfere with the application's processing, first you need to unpack the serialized content, edit it as required, and reserialize it correctly. Simply editing the raw serialized data will almost certainly break the format and cause a parsing error when the application processes the message.

Each browser extension technology comes with its own scheme for serializing data within HTTP messages. In general, therefore, you can infer the serialization format based on the type of client component that is being employed, but the format usually is evident in any case from a close inspection of the relevant HTTP messages.

Java Serialization

The Java language contains native support for object serialization, and Java applets may use this to send serialized data structures between the client and server application components. Messages containing serialized Java objects usually can be identified because they have the following `Content-Type` header:

```
Content-Type: application/x-java-serialized-object
```

Having intercepted the raw serialized data using your proxy, you can deserialize it using Java itself to gain access to the primitive data items it contains.

DSer is a handy plug-in to Burp Suite that provides a framework for viewing and manipulating serialized Java objects that have been intercepted within Burp. This tool converts the primitive data within the intercepted object into XML format for easy editing. When you have modified the relevant data, DSer then reserializes the object and updates the HTTP request accordingly.

You can download DSer, and learn more about how it works, at the following URL:

<http://blog.andlabs.org/2010/09/re-visiting-java-de-serialization-it.html>

Flash Serialization

Flash uses its own serialization format that can be used to transmit complex data structures between server and client components. Action Message Format (AMF) normally can be identified via the following `Content-Type` header:

`Content-Type: application/x-amf`

Burp natively supports AMF format. When it identifies an HTTP request or response containing serialized AMF data, it unpacks the content and presents this in tree form for viewing and editing, as shown in Figure 5-5. When you have modified the relevant primitive data items within the structure, Burp reserializes the message, and you can forward it to the server or client to be processed.

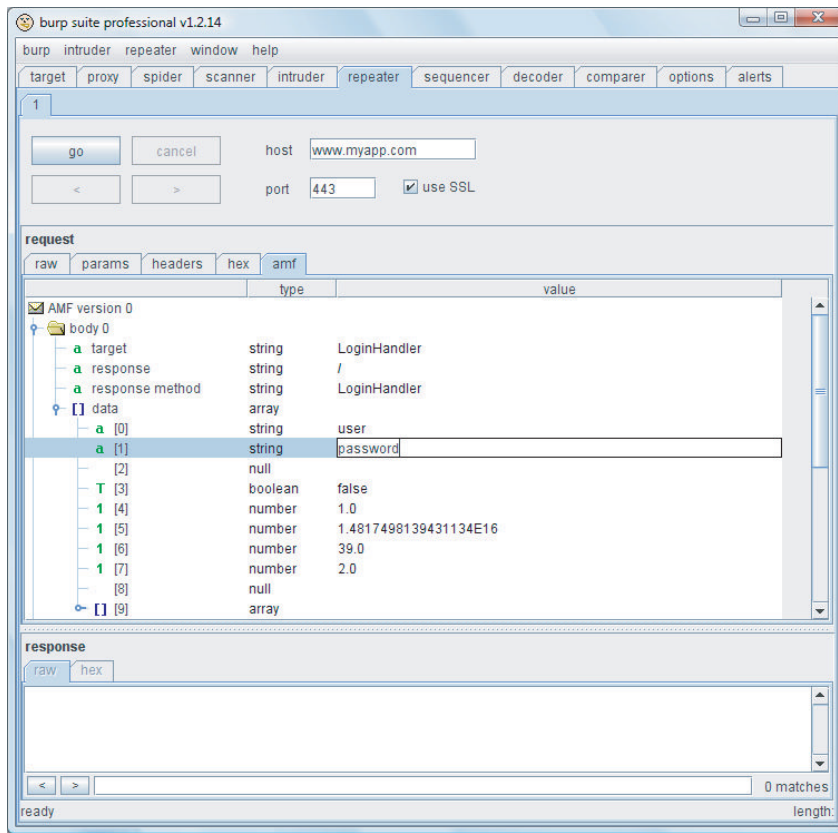


Figure 5-5: Burp Suite supports AMF format and lets you view and edit the deserialized data

Silverlight Serialization

Silverlight applications can make use of the Windows Communication Foundation (WCF) remoting framework that is built in to the .NET platform. Silverlight client components using WCF typically employ Microsoft's .NET Binary Format for SOAP (NBFS), which can be identified via the following `Content-Type` header:

```
Content-Type: application/soap+msbin1
```

A plug-in is available for Burp Proxy that automatically deserializes NBFS-encoded data before it is displayed in Burp's interception window. After you have viewed or edited the decoded data, the plug-in re-encodes the data before it is forwarded to the server or client to be processed.

The WCF binary SOAP plug-in for Burp was produced by Brian Holyfield and is available to download here:

```
www.gdssecurity.com/l/b/2009/11/19/wcf-binary-soap-plug-in-for-burp/
```

Obstacles to Intercepting Traffic from Browser Extensions

If you have set up your browser to use an intercepting proxy, you may find that requests made by browser extension components are not being intercepted by your proxy, or are failing. This problem usually is due to issues with the component's handling of HTTP proxies or SSL (or both). Typically it can be handled via some careful configuration of your tools.

The first problem is that the client component may not honor the proxy configuration you have specified in your browser or your computer's settings. This is because components may issue their own HTTP requests, outside of the APIs provided by the browser itself or the extension framework. If this is happening, you can still intercept the component's requests. You need to modify your computer's hosts file to achieve the interception and configure your proxy to support invisible proxying and automatic redirection to the correct destination host. See Chapter 20 for more details on how to do this.

The second problem is that the client component may not accept the SSL certificate being presented by your intercepting proxy. If your proxy is using a generic self-signed certificate, and you have configured your browser to accept it, the browser extension component may reject the certificate nonetheless. This may be because the browser extension does not pick up the browser's configuration for temporarily trusted certificates, or it may be because the component itself programmatically requires that untrusted certificates should not be accepted. In either case, you can circumvent this problem by configuring your proxy to use a master CA certificate, which is used to sign valid per-host certificates for each site you visit, and installing the CA certificate in your computer's trusted certificate store. See Chapter 20 for more details on how to do this.

In some rare cases you may find that client components are communicating using a protocol other than HTTP, which simply cannot be handled using an

intercepting proxy. In these situations, you still may be able to view and modify the affected traffic by using either a network sniffer or a function-hooking tool. One example is Echo Mirage, which can inject into a process and intercept calls to socket APIs, allowing you to view and modify data before it is sent over the network. Echo Mirage can be downloaded from the following URL:

www.bindshell.net/tools/echomirage

HACK STEPS

1. **Ensure that your proxy is correctly intercepting all traffic from the browser extension. If necessary, use a sniffer to identify any traffic that is not being proxied correctly.**
2. **If the client component uses a standard serialization scheme, ensure that you have the tools necessary to unpack and modify it. If the component is using a proprietary encoding or encryption mechanism, you need to decompile or debug the component to fully test it.**
3. **Review responses from the server that trigger key client-side logic. Often, timely interception and modification of a server response may allow you to “unlock” the client GUI, making it easy to reveal and then perform complex or multistaged privileged actions.**
4. **If the application performs any critical logic or events that the client component should not be trusted to perform (such as drawing a card or rolling dice in a gambling application), look for any correlation between execution of critical logic and communication with the server. If the client does not communicate with the server to determine the outcome of the event, the application is definitely vulnerable.**

Decompiling Browser Extensions

By far the most thorough method of attacking a browser extension component is to decompile the object, perform a full review of the source code, and if necessary modify the code to change the object’s behavior, and recompile it. As already discussed, browser extensions are compiled into bytecode. Bytecode is a high-level platform-independent binary representation that can be executed by the relevant interpreter (such as the Java Virtual Machine or Flash Player), and each browser extension technology uses its own bytecode format. As a result, the application can run on any platform that the interpreter itself can run on.

The high-level nature of bytecode representation means that it is always theoretically possible to decompile the bytecode into something resembling the original source code. However, various defensive techniques can be deployed to cause the decompiler to fail, or to output decompiled code that is very difficult to follow and interpret.

Subject to these obfuscation defenses, decompiling bytecode normally is the preferable route to understanding and attacking browser extension components. This allows you to review business logic, assess the full functionality of the client-side application, and modify its behavior in targeted ways.

Downloading the Bytecode

The first step is to download the executable bytecode for you to start working on. In general, the bytecode is loaded in a single file from a URL specified within the HTML source code for application pages that run the browser extension. Java applets generally are loaded using the `<applet>` tag, and other components generally are loaded using the `<object>` tag. For example:

```
<applet code="CheckQuantity.class" codebase="/scripts"
id="CheckQuantityApplet">
</applet>
```

In some cases, the URL that loads the bytecode may be less immediately obvious, since the component may be loaded using various wrapper scripts provided by the different browser extension frameworks. Another way to identify the URL for the bytecode is to look in your proxy history after your browser has loaded the browser extension. If you take this approach, you need to be aware of two potential obstacles:

- Some proxy tools apply filters to the proxy history to hide from view items such as images and style sheet files that you generally are less interested in. If you cannot find a request for the browser extension bytecode, you should modify the proxy history display filter so that all items are visible.
- Browsers usually cache the downloaded bytecode for extension components more aggressively than they do for other static resources such as images. If your browser has already loaded the bytecode for a component, even doing a full refresh for a page that uses the component may not cause the browser to request the component again. In this eventuality, you may need to fully clear your browser's cache, shut down every instance of the browser, and then start a fresh browser session to force your browser to request the bytecode again.

When you have identified the URL for the browser extension's bytecode, usually you can just paste this URL into your browser's address bar. Your browser then prompts you to save the bytecode file on your local filesystem.

TIP If you have identified the request for the bytecode in your Burp Proxy history, and the server's response contains the full bytecode (and not a reference to an earlier cached copy), you can save the bytecode directly to file

from within Burp. The most reliable way to do this is to select the Headers tab within the response viewer, right-click the lower pane containing the response body, and select Copy to File from the context menu.

Decompiling the Bytecode

Bytecode usually is distributed in a single-file package, which may need to be unpacked to obtain the individual bytecode files for decompilation into source code.

Java applets normally are packaged as `.jar` (Java archive) files, and Silverlight objects are packaged as `.xap` files. Both of these file types use the zip archive format, so you can easily unpack them by renaming the files with the `.zip` extension and then using any zip reader to unpack them into the individual files they contain. The Java bytecode is contained in `.class` files, and the Silverlight bytecode is contained in `.dll` files. After unpacking the relevant file package, you need to decompile these files to obtain source code.

Flash objects are packaged as `.swf` files and don't require any unpacking before you use a decompiler.

To perform the actual bytecode decompilation, you need to use some specific tools, depending on the type of browser extension technology that is being used, as described in the following sections.

Java Tools

Java bytecode can be decompiled to into Java source code using a tool called Jad (the Java decompiler), which is available from:

www.varanekas.com/jad

Flash Tools

Flash bytecode can be decompiled into ActionScript source code. An alternative approach, which is often more effective, is to disassemble the bytecode into a human-readable form, without actually fully decompiling it into source code.

To decompile and disassemble Flash, you can use the following tools:

- Flasm — www.nowrap.de/flasm
- Flare — www.nowrap.de/flare
- SWFScan — www.hp.com/go/swfscan (this works for Actionscript 2 and 3)

Silverlight Tools

Silverlight bytecode can be decompiled into source code using a tool called .NET Reflector, which is available from:

www.red-gate.com/products/dotnet-development/reflector/

Working on the Source Code

Having obtained the source code for the component, or something resembling it, you can take various approaches to attacking it. The first step generally is to review the source code to understand how the component works and what functionality it contains or references. Here are some items to look for:

- Input validation or other security-relevant logic and events that occur on the client side
- Obfuscation or encryption routines being used to wrap user-supplied data before it is sent to the server
- “Hidden” client-side functionality that is not visible in your user interface but that you might be able to unlock by modifying the component
- References to server-side functionality that you have not previously identified via your application mapping

Often, reviewing the source code uncovers some interesting functions within the component that you want to modify or manipulate to identify potential security vulnerabilities. This may include removing client-side input validation, submitting nonstandard data to the server, manipulating client-side state or events, or directly invoking functionality that is present within the component.

You can modify the component’s behavior in several ways, as described in the following sections.

Recompiling and Executing Within the Browser

You can modify the decompiled source code to change the component’s behavior, recompile it to bytecode, and execute the modified component within your browser. This approach is often preferred when you need to manipulate key client-side events, such as the rolling of dice in a gaming application.

To perform the recompilation, you need to use the developer tools that are relevant to the technology you are using:

- For Java, use the `javac` program in the JDK to recompile your modified source code.
- For Flash, you can use `flasm` to reassemble your modified bytecode or one of the Flash development studios from Adobe to recompile modified ActionScript source code.
- For Silverlight, use Visual Studio to recompile your modified source code.

Having recompiled your source code into one or more bytecode files, you may need to repack the distributable file if required for the technology being used. For Java and Silverlight, replace the modified bytecode files in your

unpacked archive, repackage using a zip utility, and then change the extension back to `.jar` or `.xap` as appropriate.

The final step is to load your modified component into your browser so that your changes can take effect within the application you are testing. You can achieve this in various ways:

- If you can find the physical file within your browser's on-disk cache that contains the original executable, you can replace this with your modified version and restart your browser. This approach may be difficult if your browser does not use a different individual file for each cached resource or if caching of browser extension components is implemented only in memory.
- Using your intercepting proxy, you can modify the source code of the page that loads the component and specify a different URL, pointing to either the local filesystem or a web server that you control. This approach normally is difficult because changing the domain from which the component is loaded may violate the browser's same origin policy and may require reconfiguring your browser or other methods to weaken this policy.
- You can cause your browser to reload the component from the original server (as described in the earlier section "Downloading the Bytecode"), use your proxy to intercept the response containing the executable, and replace the body of the message with your modified version. In Burp Proxy, you can use the Paste from File context menu option to achieve this. This approach usually is the easiest and least likely to run into the problems described previously.

Recompiling and Executing Outside the Browser

In some cases, it is not necessary to modify the component's behavior while it is being executed. For example, some browser extension components validate user-supplied input and then obfuscate or encrypt the result before sending it to the server. In this situation, you may be able to modify the component to perform the required obfuscation or encryption on arbitrary unvalidated input and simply output the result locally. You can then use your proxy to intercept the relevant request when the original component submits the validated input, and you can replace this with the value that was output by your modified component.

To carry out this attack, you need to change the original executable, which is designed to run within the relevant browser extension, into a standalone program that can be run on the command line. The way this is done depends on the programming language being used. For example, in Java you simply need to implement a `main` method. The section "Java Applets: A Worked Example" gives an example of how to do this.

Manipulating the Original Component Using JavaScript

In some cases, it is not necessary to modify the component's bytecode. Instead, you may be able to achieve your objectives by modifying the JavaScript within the HTML page that interacts with the component.

Having reviewed the component's source code, you can identify all its public methods that can be invoked directly from JavaScript, and the way in which parameters to those methods are handled. Often, more methods are available than are ever called from within application pages, and you may also discover more about the purpose and handling of parameters to these methods.

For example, a component may expose a method that can be invoked to enable or disable parts of the visible user interface. Using your intercepting proxy, you may be able to edit the HTML page that loads the component and modify or add some JavaScript to unlock parts of the interface that are hidden.

HACK STEPS

1. Use the techniques described to download the component's bytecode, unpack it, and decompile it into source code.
2. Review the relevant source code to understand what processing is being performed.
3. If the component contains any public methods that can be manipulated to achieve your objective, intercept an HTML response that interacts with the component, and add some JavaScript to invoke the appropriate methods using your input.
4. If not, modify the component's source code to achieve your objective, and then recompile it and execute it, either in your browser or as a standalone program.
5. If the component is being used to submit obfuscated or encrypted data to the server, use your modified version of the component to submit various suitably obfuscated attack strings to the server to probe for vulnerabilities, as you would for any other parameter.

Coping with Bytecode Obfuscation

Because of the ease with which bytecode can be decompiled to recover its source, various techniques have been developed to obfuscate the bytecode itself. Applying these techniques results in bytecode that is harder to decompile or that decompiles to misleading or invalid source code that may be very difficult to understand and impossible to recompile without substantial effort. For example, consider the following obfuscated Java source:

```
package myapp.interface;  
  
import myapp.class.public;  
import myapp.throw.throw;
```

```

import if.if.if.if.else;
import java.awt.event.KeyEvent;

public class double extends public implements strict
{
    public double(j j1)
    {
        _mthif();
        _fldif = j1;
    }
    private void _mthif(ActionEvent actionevent)
    {
        _mthif(((KeyEvent) (null)));
        switch(_fldif._mthnew()._fldif)
        {
            case 0:
                _fldfloat.setEnabled(false);
                _fldboolean.setEnabled(false);
                _fldinstanceof.setEnabled(false);
                _fldint.setEnabled(false);
                break;
        }
    }
    ...
}

```

The obfuscation techniques commonly employed are as follows:

- Meaningful class, method, and member variable names are replaced with meaningless expressions such as `a`, `b`, and `c`. This forces the reader of decompiled code to identify the purpose of each item by studying how it is used. This can make it difficult to keep track of different items while tracing them through the source code.
- Going further, some obfuscators replace item names with keywords reserved for the language, such as `new` and `int`. Although this technically renders the bytecode illegal, most virtual machines (VMs) tolerate the illegal code, and it executes normally. However, even if a decompiler can handle the illegal bytecode, the resulting source code is even less readable than that just described. More importantly, the source cannot be recompiled without extensive reworking to consistently rename illegally named items.
- Many obfuscators strip unnecessary debug and meta-information from the bytecode, including source filenames and line numbers (which makes stack traces less informative), local variable names (which frustrates debugging), and inner class information (which stops reflection from working properly).
- Redundant code may be added that creates and manipulates various kinds of data in significant-looking ways but that is autonomous from the real data actually being used by the application's functionality.

- The path of execution through code can be modified in convoluted ways, through the use of jump instructions, so that the logical sequence of execution is hard to discern when reading through the decompiled source.
- Illegal programming constructs may be introduced, such as unreachable statements and code paths with missing `return` statements. Most VMs tolerate these phenomena in bytecode, but the decompiled source cannot be recompiled without correcting the illegal code.

HACK STEPS

Effective tactics for coping with bytecode obfuscation depend on the techniques used and the purpose for which you are analyzing the source. Here are some suggestions:

1. You can review a component for public methods without fully understanding the source. It should be obvious which methods can be invoked from JavaScript, and what their signatures are, enabling you to test the behavior of the methods by passing in various inputs.
2. If class, method, and member variable names have been replaced with meaningless expressions (but not special words reserved by the programming language), you can use the refactoring functionality built into many IDEs to help yourself understand the code. By studying how items are used, you can start to assign them meaningful names. If you use the `rename` tool within the IDE, it does a lot of work for you, tracing the item's use throughout the codebase and renaming it everywhere.
3. You can actually undo a lot of obfuscation by running the obfuscated bytecode through an obfuscator a second time and choosing suitable options. A useful obfuscator for Java is Jode. It can remove redundant code paths added by another obfuscator and facilitate the process of understanding obfuscated names by assigning globally unique names to items.

Java Applets: A Worked Example

We will now consider a brief example of decompiling browser extensions by looking at a shopping application that performs input validation within a Java applet.

In this example, the form that submits the user's requested order quantity looks like this:

```
<form method="post" action="Shop.aspx?prod=2" onsubmit="return
validateForm(this)">
<input type="hidden" name="obfpad"
value="klGSB8X9x0WFv9KGqilePdQaxHISU5RnojwPdBRgZuiXSB3TgkupaFigj
UQm8CIP5HJxpidrPOuQPw63ogZ2vbyiOevPrkxFiuUxA8Gn301ep2Lax6IyuyEU
```

```

D9SmG7c">
<script>
function validateForm(theForm)
{
    var obfquantity =
    document.CheckQuantityApplet.doCheck(
    theForm.quantity.value, theForm.obfpad.value);
    if (obfquantity == undefined)
    {
        alert('Please enter a valid quantity.');
```

```

        return false;
    }
    theForm.quantity.value = obfquantity;
    return true;
}
</script>
<applet code="CheckQuantity.class" codebase="/scripts" width="0"
height="0"
    id="CheckQuantityApplet"></applet>
Product: Samsung Multiverse <br/>
Price: 399 <br/>
Quantity: <input type="text" name="quantity"> (Maximum quantity is 50)
<br/>
<input type="submit" value="Buy">
</form>

```

When the form is submitted with a quantity of 2, the following request is made:

```

POST /shop/154/Shop.aspx?prod=2 HTTP/1.1
Host: mdsec.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 77

obfpad=klGSB8X9x0WFv9KGqilePdQaxHIsU5RnojwPdBRgZuiXSB3TgkupaFigjUQm8CIP5
HJxpidrPOuQ
Pw63ogZ2vbyiOevPrkxFiuUxA8Gn30olep2Lax6IyuyEUD9SmG7c&quantity=4b282c510f
776a405f465
877090058575f445b536545401e4268475e105b2d15055c5d5204161000

```

As you can see from the HTML code, when the form is submitted, the validation script passes the user's supplied quantity, and the value of the obfpad parameter, to a Java applet called `CheckQuantity`. The applet apparently performs the necessary input validation and returns to the script an obfuscated version of the quantity, which is then submitted to the server.

Since the server-side application confirms our order for two units, it is clear that the `quantity` parameter somehow contains the value we have requested. However, if we try to modify this parameter without knowledge of the obfuscation algorithm, the attack fails, presumably because the server fails to unpack our obfuscated value correctly.

In this situation, we can use the methodology already described to decompile the Java applet and understand how it functions. First, we need to download the bytecode for the applet from the URL specified in the `applet` tag of the HTML page:

```
/scripts/CheckQuantity.class
```

Since the executable is not packaged as a `.jar` file, there is no need to unpack it, and we can run `Jad` directly on the downloaded `.class` file:

```
C:\tmp>jad CheckQuantity.class
Parsing CheckQuantity.class...The class file version is 50.0 (only 45.3,
46.0 and 47.0 are supported)
Generating CheckQuantity.jad
Couldn't fully decompile method doCheck
Couldn't resolve all exception handlers in method doCheck
```

`Jad` outputs the decompiled source code as a `.jad` file, which we can view in any text editor:

```
// Decompiled by Jad v1.5.8f. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   CheckQuantity.java

import java.applet.Applet;

public class CheckQuantity extends Applet
{
    public CheckQuantity()
    {
    }

    public String doCheck(String s, String s1)
    {
        int i = 0;
        i = Integer.parseInt(s);
        if(i <= 0 || i > 50)
            return null;
        break MISSING_BLOCK_LABEL_26;
        Exception exception;
        exception;
        return null;
        String s2 = (new StringBuilder()).append("rand=").append
(Math.random()).append("&q=").append(Integer.toString(i)).append
("&checked=true").toString();
        StringBuilder stringbuilder = new StringBuilder();
        for(int j = 0; j < s2.length(); j++)
        {
            String s3 = (new StringBuilder()).append('0').append
(Integer.toHexString((byte)s1.charAt((j * 19 + 7) % s1.length()) ^
s2.charAt(j))).toString();
```



```

        int k = s3.length();
        if(k > 2)
            s3 = s3.substring(k - 2, k);
        stringBuilder.append(s3);
    }

    return stringBuilder.toString();
}
}

```

As you can see from the decompiled source, Jad has done a reasonable job of decompiling, and the source code for the applet is simple. When the `doCheck` method is called with the user-supplied `quantity` and application-supplied `obfpad` parameters, the applet first validates that the quantity is a valid number and is between 1 and 50. If so, it builds a string of name/value pairs using the URL querystring format, which includes the validated quantity. Finally, it obfuscates this string by performing XOR operations against characters with the `obfpad` string that the application supplied. This is a fairly easy and common way of adding some superficial obfuscation to data to prevent trivial tampering.

We have described various approaches you can take when you have decompiled and analyzed the source code for a browser extension component. In this case, the easiest way to subvert the applet is as follows:

1. Modify the `doCheck` method to remove the input validation, allowing you to supply an arbitrary string as your quantity.
2. Add a `main` method, allowing you to execute the modified component from the command line. This method simply calls the modified `doCheck` method and prints the obfuscated result to the console.

When you have made these changes, the modified source code is as follows:

```

public class CheckQuantity
{
    public static void main(String[] a)
    {
        System.out.println(doCheck("999",
            "klGSB8X9x0WFv9KGgilePdQaxHIsU5RnoJwPdBRgZuiXSB3TgkupaFigjUQm8CIP5HJxpi
            drPOuQPw63ogZ2vbyi0evPrkxFiuUxA8Gn30o1ep2Lax6IyuyEUD9 SmG7c"));
    }

    public static String doCheck(String s, String s1)
    {
        String s2 = (new StringBuilder()).append("rand=").append
            (Math.random()).append("&q=").append(s).append
            ("&checked=true").toString();
        StringBuilder stringBuilder = new StringBuilder();
        for(int j = 0; j < s2.length(); j++)
        {
            String s3 = (new StringBuilder()).append('0').append

```

```

(Integer.toHexString((byte)s1.charAt((j * 19 + 7) % s1.length()) ^
s2.charAt(j))).toString();
    int k = s3.length();
    if(k > 2)
        s3 = s3.substring(k - 2, k);
    stringBuilder.append(s3);
}
return stringBuilder.toString();
}
}

```

This version of the modified component provides a valid obfuscated string for the arbitrary quantity of 999. Note that you could use nonnumeric input here, allowing you to probe the application for various kinds of input-based vulnerabilities.

TIP The Jad program saves its decompiled source code with the `.jad` extension. However, if you want to modify and recompile the source code, you need to rename each source file with the `.java` extension.

All that remains is to recompile the source code using the `javac` compiler that comes with the Java SDK, and then execute the component from the command line:

```

C:\tmp>javac CheckQuantity.java
C:\tmp>java CheckQuantity
4b282c510f776a455d425a7808015c555f42585460464d1e42684c414a152b1e0b5a520a
145911171609

```

Our modified component has now performed the necessary obfuscation on our arbitrary quantity of 999. To deliver the attack to the server, we simply need to submit the order form in the normal way using valid input, intercept the resulting request using our proxy, and substitute the obfuscated quantity with the one provided by our modified component. Note that if the application issues a new obfuscation pad each time the order form is loaded, you need to ensure that the obfuscation pad being submitted back to the server matches the one that was used to obfuscate the quantity also being submitted.

TRY IT!

These examples demonstrate the attack just described and the corresponding attacks using Silverlight and Flash technologies:

```

http://mdsec.net/shop/154/
http://mdsec.net/shop/167/
http://mdsec.net/shop/179/

```

Attaching a Debugger

Decompilation is the most complete method of understanding and compromising a browser extension. However, in large and complex components containing tens of thousands of lines of code, it is nearly always much quicker to observe the component during execution, correlating methods and classes with key actions within the interface. This approach also avoids difficulties that may arise with interpreting and recompiling obfuscated bytecode. Often, achieving a specific objective is as simple as executing a key function and altering its behavior to circumvent the controls implemented within the component.

Because the debugger is working at the bytecode level, it can be easily used to control and understand the flow of execution. In particular, if source code can be obtained through decompilation, breakpoints can be set on specific lines of code, allowing the understanding gained through decompilation to be supported by practical observation of the code path taken during execution.

Although efficient debuggers are not fully matured for all the browser extension technologies, debugging is well supported for Java applets. By far the best resource for this is JavaSnoop, a Java debugger that can integrate Jad to decompile source code, trace variables through an application, and set breakpoints on methods to view and modify parameters. Figure 5-6 shows JavaSnoop being used to hook directly into a Java applet running in the browser. Figure 5-7 shows JavaSnoop being used to tamper with the return value from a method.

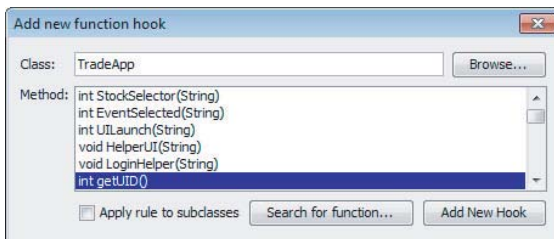


Figure 5-6: JavaSnoop can hook directly into an applet running in the browser

NOTE It's best to run JavaSnoop before the target applet is loaded.

JavaSnoop turns off the restrictions set by your Java security policy so that it can operate on the target. In Windows, it does this by granting all permissions to all Java programs on your system, so ensure that JavaSnoop shuts down cleanly and that permissions are restored when you are finished working.

An alternative tool for debugging Java is JSwat, which is highly configurable. In large projects containing many class files, it is sometimes preferable

to decompile, modify, and recompile a key class file and then use JSwat to hot-swap it into the running application. To use JSwat, you need to launch an applet using the `appletviewer` tool included in the JDK and then connect JSwat to it. For example, you could use this command:

```
appletviewer -J-Xdebug -J-Djava.compiler=NONE -J-
Xrunjdwp:transport=dt_socket,
server=y,suspend=n,address=5000 appletpage.htm
```

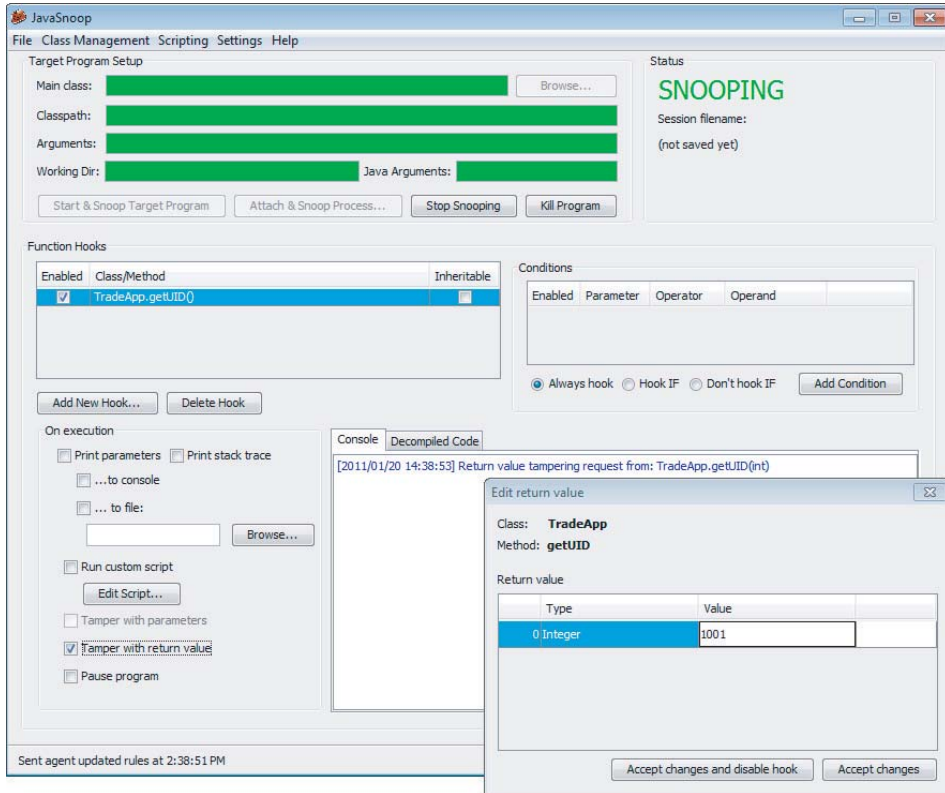


Figure 5-7: Once a suitable method has been identified, JavaSnoop can be used to tamper with the return value from the method

When you're working on Silverlight objects, you can use the Silverlight Spy tool to monitor the component's execution at runtime. This can greatly help correlate relevant code paths to events that occur within the user interface. Silverlight Spy is available from the following URL:

<http://firstfloorsoftware.com/SilverlightSpy/>

Native Client Components

Some applications need to perform actions within the user's computer that cannot be conducted from inside a browser-based VM sandbox. In terms of client-side security controls, here are some examples of this functionality:

- Verifying that a user has an up-to-date virus scanner
- Verifying that proxy settings and other corporate configuration are in force
- Integrating with a smartcard reader

Typically, these kinds of actions require the use of native code components, which integrate local application functionality with web application functionality. Native client components are often delivered via ActiveX controls. These are custom browser extensions that run outside the browser sandbox.

Native client components may be significantly harder to decipher than other browser extensions, because there is no equivalent to intermediate bytecode. However, the principles of bypassing client-side controls still apply, even if this requires a different toolset. Here are some examples of popular tools used for this task:

- OllyDbg is a Windows debugger that can be used to step through native executable code, set breakpoints, and apply patches to executables, either on disk or at runtime.
- IDA Pro is a disassembler that can produce human-readable assembly code from native executable code on a wide variety of platforms.

Although a full-blown description is outside the scope of this book, the following are some useful resources if you want to know more about reverse engineering of native code components and related topics:

- *Reversing: Secrets of Reverse Engineering* by Eldad Eilam
- *Hacker Disassembling Uncovered* by Kris Kaspersky
- *The Art of Software Security Assessment* by Mark Dowd, John McDonald, and Justin Schuh
- *Fuzzing for Software Security Testing and Quality Assurance (Artech House Information Security and Privacy)* by Ari Takanen, Jared DeMott, and Charlie Miller
- *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler* by Chris Eagle
- www.acm.uiuc.edu/sigmil/RevEng
- www.uninformed.org/?v=1&a=7

Handling Client-Side Data Securely

As you have seen, the core security problem with web applications arises because client-side components and user input are outside the server's direct control. The client, and all the data received from it, is inherently untrustworthy.

Transmitting Data Via the Client

Many applications leave themselves exposed because they transmit critical data such as product prices and discount rates via the client in an unsafe manner.

If possible, applications should avoid transmitting this kind of data via the client. In virtually any conceivable scenario, it is possible to hold such data on the server and reference it directly from server-side logic when needed. For example, an application that receives users' orders for various products should allow users to submit a product code and quantity and look up the price of each requested product in a server-side database. There is no need for users to submit the prices of items back to the server. Even where an application offers different prices or discounts to different users, there is no need to depart from this model. Prices can be held within the database on a per-user basis, and discount rates can be stored in user profiles or even session objects. The application already possesses, server-side, all the information it needs to calculate the price of a specific product for a specific user. It must. Otherwise, it would be unable, on the insecure model, to store this price in a hidden form field.

If developers decide they have no alternative but to transmit critical data via the client, the data should be signed and/or encrypted to prevent user tampering. If this course of action is taken, there are two important pitfalls to avoid:

- Some ways of using signed or encrypted data may be vulnerable to replay attacks. For example, if the product price is encrypted before being stored in a hidden field, it may be possible to copy the encrypted price of a cheaper product and submit it in place of the original price. To prevent this attack, the application needs to include sufficient context within the encrypted data to prevent it from being replayed in a different context. For example, the application could concatenate the product code and price, encrypt the result as a single item, and then validate that the encrypted string submitted with an order actually matches the product being ordered.
- If users know and/or control the plaintext value of encrypted strings that are sent to them, they may be able to mount various cryptographic attacks to discover the encryption key the server is using. Having done this, they can encrypt arbitrary values and fully circumvent the protection offered by the solution.

In applications running on the ASP.NET platform, it is advisable never to store any customized data within the `ViewState` — especially anything sensitive that you would not want to be displayed on-screen to users. The option to enable the `ViewState` MAC should always be activated.

Validating Client-Generated Data

Data generated on the client and transmitted to the server cannot in principle be validated securely on the client:

- Lightweight client-side controls such as HTML form fields and JavaScript can be circumvented easily and provide no assurance about the input that the server receives.
- Controls implemented in browser extension components are sometimes more difficult to circumvent, but this may merely slow down an attacker for a short period.
- Using heavily obfuscated or packed client-side code provides additional obstacles; however, a determined attacker can always overcome these. (A point of comparison in other areas is the use of DRM technologies to prevent users from copying digital media files. Many companies have invested heavily in these client-side controls, and each new solution usually is broken within a short time.)

The only secure way to validate client-generated data is on the server side of the application. Every item of data received from the client should be regarded as tainted and potentially malicious.

COMMON MYTH

It is sometimes believed that any use of client-side controls is bad. In particular, some professional penetration testers report the presence of client-side controls as a “finding” without verifying whether they are replicated on the server or whether there is any non-security explanation for their existence. In fact, despite the significant caveats arising from the various attacks described in this chapter, there are nevertheless ways to use client-side controls that do not give rise to any security vulnerabilities:

- **Client-side scripts can be used to validate input as a means of enhancing usability, avoiding the need for round-trip communication with the server. For example, if the user enters her date of birth in an incorrect format, alerting her to the problem via a client-side script provides a much more seamless experience. Of course, the application must revalidate the item submitted when it arrives at the server.**

Continued

COMMON MYTH (continued)

- Sometimes client-side data validation can be effective as a security measure — for example, as a defense against DOM-based cross-site scripting attacks. However, these are cases where the focus of the attack is another application user, rather than the server-side application, and exploiting a potential vulnerability does not necessarily depend on transmitting any malicious data to the server. See Chapters 12 and 13 for more details on this kind of scenario.
- As described previously, there are ways of transmitting encrypted data via the client that are not vulnerable to tampering or replay attacks.

Logging and Alerting

When an application employs mechanisms such as length limits and JavaScript-based validation to enhance performance and usability, these should be integrated with server-side intrusion detection defenses. The server-side logic that performs validation of client-submitted data should be aware of the validation that has already occurred on the client side. If data that would have been blocked by client-side validation is received, the application may infer that a user is actively circumventing this validation and therefore is likely to be malicious. Anomalies should be logged and, if appropriate, application administrators should be alerted in real time so that they can monitor any attempted attack and take suitable action as required. The application may also actively defend itself by terminating the user's session or even suspending his account.

NOTE In some cases where JavaScript is employed, the application still can be used by users who have disabled JavaScript within their browsers. In this situation, the browser simply skips JavaScript-based form validation code, and the raw input entered by the user is submitted. To avoid false positives, the logging and alerting mechanism should be aware of where and how this can arise.

Summary

Virtually all client/server applications must accept the fact that the client component, and all processing that occurs on it, cannot be trusted to behave as expected. As you have seen, the transparent communications methods generally employed by web applications mean that an attacker equipped with simple tools and minimal skill can easily circumvent most controls implemented on the client. Even where an application attempts to obfuscate data and processing residing on the client side, a determined attacker can compromise these defenses.

In every instance where you identify data being transmitted via the client, or validation of user-supplied input being implemented on the client, you should test how the server responds to unexpected data that bypasses those controls. Often, serious vulnerabilities lurk behind an application's assumptions about the protection afforded to it by defenses that are implemented at the client.

Questions

Answers can be found at <http://mdsec.net/wahh>.

1. How can data be transmitted via the client in a way that prevents tampering attacks?
2. An application developer wants to stop an attacker from performing brute-force attacks against the login function. Because the attacker may target multiple usernames, the developer decides to store the number of failed attempts in an encrypted cookie, blocking any request if the number of failed attempts exceeds five. How can this defense be bypassed?
3. An application contains an administrative page that is subject to rigorous access controls. It contains links to diagnostic functions located on a different web server. Access to these functions should also be restricted to administrators only. Without implementing a second authentication mechanism, which of the following client-side mechanisms (if any) could be used to safely control access to the diagnostic functionality? Do you need any more information to help choose a solution?
 - (a) The diagnostic functions could check the HTTP `Referer` header to confirm that the request originated on the main administrative page.
 - (b) The diagnostic functions could validate the supplied cookies to confirm that these contain a valid session token for the main application.
 - (c) The main application could set an authentication token in a hidden field that is included within the request. The diagnostic function could validate this to confirm that the user has a session on the main application.
4. If a form field includes the attribute `disabled=true`, it is not submitted with the rest of the form. How can you change this behavior?
5. Are there any means by which an application can ensure that a piece of input validation logic has been run on the client?