CHAPTER 8

Attacking Access Controls

Within the application's core security mechanisms, access controls are logically built on authentication and session management. So far, you have seen how an application can first verify a user's identity and then confirm that a particular sequence of requests that it receives originated from the same user. The primary reason that the application needs to do these things — in terms of security, at least — is because it needs a way to decide whether it should permit a given request to perform its attempted action or access the resources it is requesting. Access controls are a critical defense mechanism within the application because they are responsible for making these key decisions. When they are defective, an attacker can often compromise the entire application, taking control of administrative functionality and accessing sensitive data belonging to every other user.

As noted in Chapter 1, broken access controls are among the most commonly encountered categories of web application vulnerability, affecting a massive 71 percent of the applications recently tested by the authors. It is extremely common to encounter applications that go to all the trouble of implementing robust mechanisms for authentication and session management, only to squander that investment by neglecting to build effective access controls on them. One reason that these weaknesses are so prevalent is that access control checks need to be performed for every request and every operation on a resource that particular user attempts to perform, at a specific time. And unlike many other classes of control, this is a design decision that needs to be made by a human; it cannot be resolved by employing technology.

Access control vulnerabilities are conceptually simple: The application lets you do something you shouldn't be able to. The differences between separate flaws really come down to the different ways in which this core defect is manifested and the different techniques you need to employ to detect it. This chapter describes all these techniques, showing how you can exploit different kinds of behavior within an application to perform unauthorized actions and access protected data.

Common Vulnerabilities

Access controls can be divided into three broad categories: vertical, horizontal, and context-dependent.

Vertical access controls allow different types of users to access different parts of the application's functionality. In the simplest case, this typically involves a division between ordinary users and administrators. In more complex cases, vertical access controls may involve fine-grained user roles granting access to specific functions, with each user being allocated to a single role, or a combination of different roles.

Horizontal access controls allow users to access a certain subset of a wider range of resources of the same type. For example, a web mail application may allow you to read your e-mail but no one else's, an online bank may let you transfer money out of your account only, and a workflow application may allow you to update tasks assigned to you but only read tasks assigned to other people.

Context-dependent access controls ensure that users' access is restricted to what is permitted given the current application state. For example, if a user is following multiple stages within a process, context-dependent access controls may prevent the user from accessing stages out of the prescribed order.

In many cases, vertical and horizontal access controls are intertwined. For example, an enterprise resource planning application may allow each accounts payable clerk to pay invoices for a specific organizational unit and no other. The accounts payable manager, on the other hand, may be allowed to pay invoices for any unit. Similarly, clerks may be able to pay invoices for small amounts, but larger invoices must be paid by the manager. The finance director may be able to view invoice payments and receipts for every organizational unit in the company but may not be permitted to pay any invoices.

Access controls are broken if any user can access functionality or resources for which he or she is not authorized. There are three main types of attacks against access controls, corresponding to the three categories of controls:

- **Vertical privilege escalation** occurs when a user can perform functions that his assigned role does not permit him to. For example, if an ordinary user can perform administrative functions, or a clerk can pay invoices of any size, access controls are broken.

- **Horizontal privilege escalation** occurs when a user can view or modify resources to which he is not entitled. For example, if you can use a web mail application to read other people's e-mail, or if a payment clerk can process invoices for an organizational unit other than his own, access controls are broken.
- **Business logic exploitation** occurs when a user can exploit a flaw in the application's state machine to gain access to a key resource. For example, a user may be able to bypass the payment step in a shopping checkout sequence.

It is common to find cases where vulnerability in the application's horizontal separation of privileges can lead immediately to a vertical escalation attack. For example, if a user finds a way to set a different user's password, the user can attack an administrative account and take control of the application.

In the cases described so far, broken access controls enable users who have authenticated themselves to the application in a particular user context to perform actions or access data for which that context does not authorize them. However, in the most serious cases of broken access control, it may be possible for completely unauthorized users to gain access to functionality or data that is intended to be accessed only by privileged authenticated users.

Completely Unprotected Functionality

In many cases of broken access controls, sensitive functionality and resources can be accessed by anyone who knows the relevant URL. For example, with many applications, anyone who visits a specific URL can make full use of its administrative functions:

```
https://wahh-app.com/admin/
```

In this situation, the application typically enforces access control only to the following extent: users who have logged in as administrators see a link to this URL on their user interface, and other users do not. This cosmetic difference is the only mechanism in place to "protect" the sensitive functionality from unauthorized use.

Sometimes, the URL that grants access to powerful functions may be less easy to guess, and may even be quite cryptic:

```
https://wahh-app.com/menus/secure/ff457/DoAdminMenu2.jsp
```

Here, access to administrative functions is protected by the assumption that an attacker will not know or discover this URL. The application is harder for an outsider to compromise, because he is less likely to guess the URL by which he can do so.

COMMON MYTH

"No low-privileged users will know that URL. We don't reference it anywhere within the application."

The absence of any genuine access control still constitutes a serious vulnerability, regardless of how easy it would be to guess the URL. URLs do not have the status of secrets, either within the application itself or in the hands of its users. They are displayed on-screen, and they appear in browser histories and the logs of web servers and proxy servers. Users may write them down, bookmark them, or e-mail them. They are not usually changed periodically, as passwords should be. When users change job roles, and their access to administrative functionality needs to be withdrawn, there is no way to delete their knowledge of a particular URL.

In some applications where sensitive functionality is hidden behind URLs that are not easy to guess, an attacker may often be able to identify these via close inspection of client-side code. Many applications use JavaScript to build the user interface dynamically within the client. This typically works by setting various flags regarding the user's status and then adding individual elements to the UI on the basis of these:

```
var isAdmin = false;
...
if (isAdmin)
{
    adminMenu.addItem("/menus/secure/ff457/addNewPortalUser2.jsp",
        "create a new user");
}
```

Here, an attacker can simply review the JavaScript to identify URLs for administrative functionality and attempt to access these. In other cases, HTML comments may contain references to or clues about URLs that are not linked from on-screen content. Chapter 4 discusses the various techniques by which an attacker can gather information about hidden content within the application.

Direct Access to Methods

A specific case of unprotected functionality can arise when applications expose URLs or parameters that are actually remote invocations of API methods, normally those exposed by a Java interface. This often occurs when server-side code is moved to a browser extension component and method stubs are created so that the code can still call the server-side methods it requires to function. Outside of this situation, some instances of direct access to methods can be identified where URLs or parameters use the standard Java naming conventions, such as `getBalance` and `isExpired`.

In principle, requests specifying a server-side API to be executed need be no less secure than those specifying a server-side script or other resource. In practice, however, this type of mechanism frequently contains vulnerabilities. Often, the client interacts directly with server-side API methods and bypasses the application's normal controls over access or unexpected input vectors. There is also a chance that other functionality exists that can be invoked in this way and is not protected by any controls, on the assumption that it could never be directly invoked by web application clients. Often, there is a need to provide users with access to certain specific methods, but they are instead given access to all methods. This is either because the developer is not fully aware of which subset of methods to proxy and provides access to all methods, or because the API used to map them to the HTTP server provides access to all methods by default.

The following example shows the `getCurrentUserRoles` method being invoked from within the interface `securityCheck`:

```
http://wahh-app.com/public/securityCheck/getCurrentUserRoles
```

In this example, in addition to testing the access controls over the `getCurrentUserRoles` method, you should check for the existence of other similarly named methods such as `getAllUserRoles`, `getAllRoles`, `getAllUsers`, and `getCurrentUserPermissions`. Further considerations specific to the testing of direct access to methods are described later in this chapter.

Identifier-Based Functions

When a function of an application is used to gain access to a specific resource, it is common to see an identifier for the requested resource being passed to the server in a request parameter, within either the URL query string or the body of a `POST` request. For example, an application may use the following URL to display a specific document belonging to a particular user:

```
https://wahh-app.com/ViewDocument.php?docid=1280149120
```

When the user who owns the document is logged in, a link to this URL is displayed on the user's My Documents page. Other users do not see the link. However, if access controls are broken, any user who requests the relevant URL may be able to view the document in exactly the same way as the authorized user.

TIP **This type of vulnerability often arises when the main application interfaces with an external system or back-end component. It can be difficult to share a session-based security model between different systems that may be based on diverse technologies. Faced with this problem, developers frequently take a shortcut and move away from that model, using client-submitted parameters to make access control decisions.**

In this example, an attacker seeking to gain unauthorized access needs to know not only the name of the application page (`ViewDocument.php`) but also the identifier of the document he wants to view. Sometimes, resource identifiers are generated in a highly unpredictable manner; for example, they may be randomly chosen GUIDs. In other cases, they may be easily guessed; for example, they may be sequentially generated numbers. However, the application is vulnerable in both cases. As described previously, URLs do not have the status of secrets, and the same applies to resource identifiers. Often, an attacker who wants to discover the identifiers of other users' resources can find some location within the application that discloses these, such as access logs. Even where an application's resource identifiers cannot be easily guessed, the application is still vulnerable if it fails to properly control access to those resources. In cases where the identifiers are easily predicted, the problem is even more serious and more easily exploited.

TIP **Application logs are often a gold mine of information. They may contain numerous items of data that can be used as identifiers to probe functionality that is accessed in this way. Identifiers commonly found within application logs include usernames, user ID numbers, account numbers, document IDs, user groups and roles, and e-mail addresses.**

NOTE **In addition to being used as references to data-based resources within the application, this kind of identifier is often used to refer to functions of the application itself. As you saw in Chapter 4, an application may deliver different functions via a single page, which accepts a function name or identifier as a parameter. Again in this situation, access controls may run no deeper than the presence or absence of specific URLs within the interfaces of different types of users. If an attacker can determine the identifier for a sensitive function, he may be able to access it in the same way as a more privileged user.**

Multistage Functions

Many kinds of functions within an application are implemented across several stages, involving multiple requests being sent from the client to the server. For example, a function to add a new user may involve choosing this option from a user maintenance menu, selecting the department and user role from drop-down lists, and then entering the new username, initial password, and other information.

It is common to encounter applications in which efforts have been made to protect this kind of sensitive functionality from unauthorized access but where the access controls employed are broken because of flawed assumptions about how the functionality will be used.

In the previous example, when a user attempts to load the user maintenance menu and chooses the option to add a new user, the application may verify that the user has the required privileges and block access if the user does not. However, if an attacker proceeds directly to the stage of specifying the user's department and other details, there may be no effective access control. The developers unconsciously assumed that any user who reaches the later stages of the process must have the relevant privileges because this was verified at the earlier stages. The result is that any user of the application can add a new administrative user account and thereby take full control of the application, gaining access to many other functions whose access control is intrinsically robust.

The authors have encountered this type of vulnerability even in the most security-critical web applications — those deployed by online banks. Making a funds transfer in a banking application typically involves multiple stages, partly to prevent users from accidentally making mistakes when requesting a transfer. This multistage process involves capturing different items of data from the user at each stage. This data is checked thoroughly when first submitted and then usually is passed to each subsequent stage, using hidden fields in HTML form. However, if the application does not revalidate all this data at the final stage, an attacker can potentially bypass the server's checks. For example, the application might verify that the source account selected for the transfer belongs to the current user and then ask for details about the destination account and the amount of the transfer. If a user intercepts the final `POST` request of this process and modifies the source account number, she can execute a horizontal privilege escalation and transfer funds out of an account belonging to a different user.

Static Files

In the majority of cases, users gain access to protected functionality and resources by issuing requests to dynamic pages that execute on the server. It is the responsibility of each such page to perform suitable access control checks and confirm that the user has the relevant privileges to perform the action he or she is attempting.

However, in some cases, requests for protected resources are made directly to the static resources themselves, which are located within the server's web root. For example, an online publisher may allow users to browse its book catalog and purchase ebooks for download. Once payment has been made, the user is directed to a download URL like the following:

```
https://wahh-books.com/download/9780636628104.pdf
```

Because this is a completely static resource, if it is hosted on a traditional web server, its contents are simply returned directly by the server, and no application-level code is executed. Hence, the resource cannot implement any logic to verify

that the requesting user has the required privileges. When static resources are accessed in this way, it is highly likely that no effective access controls are protecting them and that anyone who knows the URL naming scheme can exploit this to access any resources he wants. In the present case, the document name looks suspiciously like an ISBN, which would enable an attacker to quickly download every ebook produced by the publisher!

Certain types of functionality are particularly prone to this kind of problem, including financial websites providing access to static documents about companies such as annual reports, software vendors that provide downloadable binaries, and administrative functionality that provides access to static log files and other sensitive data collected within the application.

Platform Misconfiguration

Some applications use controls at the web server or application platform layer to control access. Typically, access to specified URL paths is restricted based on the user's role within the application. For example, access to the `/admin` path may be denied to users who are not in the Administrators group. In principle, this is an entirely legitimate means of controlling access. However, mistakes made in the configuration of the platform-level controls can often allow unauthorized access to occur.

The platform-level configuration normally takes the form of rules that are akin to firewall policy rules, which allow or deny access based on the following:

- HTTP request method
- URL path
- User role

As described in Chapter 3, the original purpose of the `GET` method is of retrieving information, and the purpose of the `POST` method is performing actions that change the application's data or state.

If care is not taken to devise rules that accurately allow access based on the correct HTTP methods and URL paths, this may lead to unauthorized access. For example, if an administrative function to create a new user uses the `POST` method, the platform may have a deny rule that disallows the `POST` method and allows all other methods. However, if the application-level code does not verify that all requests for this function are in fact using the `POST` method, an attacker may be able to circumvent the control by submitting the same request using the `GET` method. Since most application-level APIs for retrieving request parameters are agnostic as to the request method, the attacker can simply supply the required parameters within the URL query string of the `GET` request to make unauthorized use of the function.

What is more surprising, on the face of it, is that applications can still be vulnerable even if the platform-level rule denies access to both the GET and POST methods. This happens because requests using other HTTP methods may ultimately be handled by the same application code that handles GET and POST requests. One example of this is the HEAD method. According to specifications, servers should respond to a HEAD request with the same headers they would use to respond to the corresponding GET request, but with no message body. Hence, most platforms correctly service HEAD requests by executing the corresponding GET handler and just return the HTTP headers that are generated. GET requests can often be used to perform sensitive actions, either because the application itself uses GET requests for this purpose (contrary to specifications) or because it does not verify that the POST method is being used. If an attacker can use a HEAD request to add an administrative user account, he or she can live without receiving any message body in the response.

In some cases, platforms handle requests that use unrecognized HTTP methods by simply passing them to the GET request handler. In this situation, platform-level controls that just deny certain specified HTTP methods can be bypassed by specifying an arbitrary invalid HTTP method in the request.

Chapter 18 contains a specific example of this type of vulnerability arising in a web application platform product.

Insecure Access Control Methods

Some applications employ a fundamentally insecure access control model in which access control decisions are made on the basis of request parameters submitted by the client, or other conditions that are within an attacker's control.

Parameter-Based Access Control

In some versions of this model, the application determines a user's role or access level at the time of login and from this point onward transmits this information via the client in a hidden form field, cookie, or preset query string parameter (see Chapter 5). When each subsequent request is processed, the application reads this request parameter and decides what access to grant the user accordingly.

For example, an administrator using the application may see URLs like the following:

```
https://wahh-app.com/login/home.jsp?admin=true
```

The URLs seen by ordinary users contain a different parameter, or none at all. Any user who is aware of the parameter assigned to administrators can simply set it in his own requests and thereby gain access to administrative functions.

This type of access control may sometimes be difficult to detect without actually using the application as a high-privileged user and identifying what requests are made. The techniques described in Chapter 4 for discovering hidden request parameters may be successful in discovering the mechanism when working only as an ordinary user.

Referer-Based Access Control

In other unsafe access control models, the application uses the HTTP `Referer` header as the basis for making access control decisions. For example, an application may strictly control access to the main administrative menu based on a user's privileges. But when a user makes a request for an individual administrative function, the application may simply check whether this request was referred from the administrative menu page. It might assume that the user must have accessed that page and therefore has the required privileges. This model is fundamentally broken, of course, because the `Referer` header is completely under the user's control and can be set to any value.

Location-Based Access Control

Many businesses have a regulatory or business requirement to restrict access to resources depending on the user's geographic location. These are not limited to the financial sector but include news services and others. In these situations, a company may employ various methods to locate the user, the most common of which is geolocation of the user's current IP address.

Location-based access controls are relatively easy for an attacker to circumvent. Here are some common methods of bypassing them:

- Using a web proxy that is based in the required location
- Using a VPN that terminates in the required location
- Using a mobile device that supports data roaming
- Direct manipulation of client-side mechanisms for geolocation

Attacking Access Controls

Before starting to probe the application to detect any actual access control vulnerabilities, you should take a moment to review the results of your application mapping exercises (see Chapter 4). You need to understand what the application's actual requirements are in terms of access control, and therefore where it will probably be most fruitful to focus your attention.

HACK STEPS

Here are some questions to consider when examining an application's access controls:

1. **Do application functions give individual users access to a particular subset of data that belongs to them?**
2. **Are there different levels of user, such as managers, supervisors, guests, and so on, who are granted access to different functions?**
3. **Do administrators use functionality that is built into the same application to configure and monitor it?**
4. **What functions or data resources within the application have you identified that would most likely enable you to escalate your current privileges?**
5. **Are there any identifiers (by way of URL parameters of `POST` body message) that signal a parameter is being used to track access levels?**

Testing with Different User Accounts

The easiest and most effective way to test the effectiveness of an application's access controls is to access the application using different accounts. That way you can determine whether resources and functionality that can be accessed legitimately by one account can be accessed illegitimately by another.

HACK STEPS

1. **If the application segregates user access to different levels of functionality, first use a powerful account to locate all the available functionality. Then attempt to access this using a lower-privileged account to test for vertical privilege escalation.**
2. **If the application segregates user access to different resources (such as documents), use two different user-level accounts to test whether access controls are effective or whether horizontal privilege escalation is possible. For example, find a document that can be legitimately accessed by one user but not by another, and attempt to access it using the second user's account — either by requesting the relevant URL or by submitting the same `POST` parameters from within the second user's session.**

Testing an application's access controls thoroughly is a time-consuming process. Fortunately, some tools can help you automate some of the work involved, to make your testing quicker and more reliable. This will allow you to focus on the parts of the task that require human intelligence to perform effectively.

Burp Suite lets you map the contents of an application using two different user contexts. Then you can compare the results to see exactly where the content accessed by each user is the same or different.

HACK STEPS

1. **With Burp configured as your proxy and interception disabled, browse all the application's content within one user context. If you are testing vertical access controls, use the higher-privilege account for this.**
2. **Review the contents of Burp's site map to ensure that you have identified all the functionality you want to test. Then use the context menu to select the "compare site maps" feature.**
3. **To select the second site map to be compared, you can either load this from a Burp state file or have Burp dynamically rerequest the first site map in a new session context. To test horizontal access controls between users of the same type, you can simply load a state file you saved earlier, having mapped the application as a different user. For testing vertical access controls, it is preferable to rerequest the high-privilege site map as a low-privileged user, because this ensures complete coverage of the relevant functionality.**
4. **To rerequest the first site map in a different session, you need to configure Burp's session-handling functionality with the details of the low-privilege user session (for example, by recording a login macro or providing a specific cookie to be used in requests). This feature is described in more detail in Chapter 14. You may also need to define suitable scope rules to prevent Burp from requesting any logout function.**

Figure 8-1 shows the results of a simple site map comparison. Its colorized analysis of the differences between the site maps shows items that have been added, removed, or modified between the two maps. For modified items, the table includes a "diff count" column, which is the number of edits required to modify the item in the first map into the item in the second map. Also, when an item is selected, the responses are also colorized to show the locations of those edits within the responses.

Interpreting the results of the site map comparison requires human intelligence and an understanding of the meaning and context of specific application functions. For example, Figure 8-1 shows the responses that are returned to each user when they view their home page. The two responses show a different description of the logged-in user, and the administrative user has an additional menu item. These differences are to be expected, and they are neutral as to the effectiveness of the application's access controls, since they concern only the user interface.

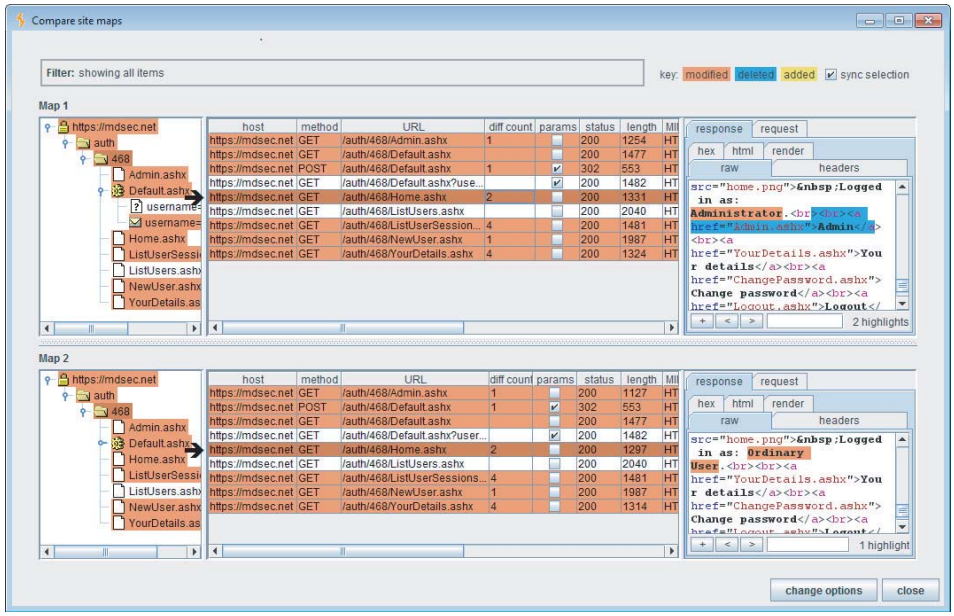

Figure 8-1: A site map comparison showing the differences between content that was accessed in different user contexts

Figure 8-2 shows the response returned when each user requests the top-level admin page. Here, the administrative user sees a menu of available options, while the ordinary user sees a "not authorized" message. These differences indicate that access controls are being applied correctly. Figure 8-3 shows the response returned when each user requests the "list users" admin function. Here, the responses are identical, indicating that the application is vulnerable, since the ordinary user should not have access to this function and does not have any link to it in his or her user interface.

Simply exploring the site map tree and looking at the number of differences between items is insufficient to evaluate the effectiveness of the application's access controls. Two identical responses may indicate a vulnerability (for example, in an administrative function that discloses sensitive information) or may be harmless (for example, in an unprotected search function). Conversely, two different responses may still mean that a vulnerability exists (for example, in an administrative function that returns different content each time it is accessed) or may be harmless (for example, in a page showing profile information about the currently logged-in user). For these reasons, fully automated tools generally are ineffective at identifying access control vulnerabilities. Using Burp's functionality to compare site maps, you can automate as much of the process as possible, giving you all the information you need in a ready form, and letting you apply your knowledge of the application's functionality to identify any actual vulnerabilities.

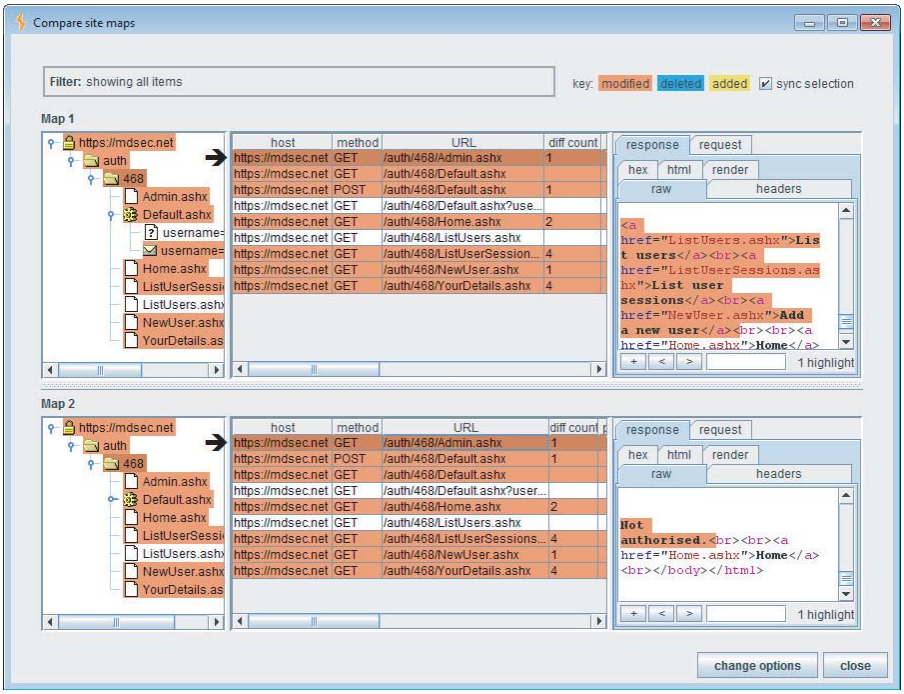

Figure 8-2: The low-privileged user is denied access to the top-level admin page

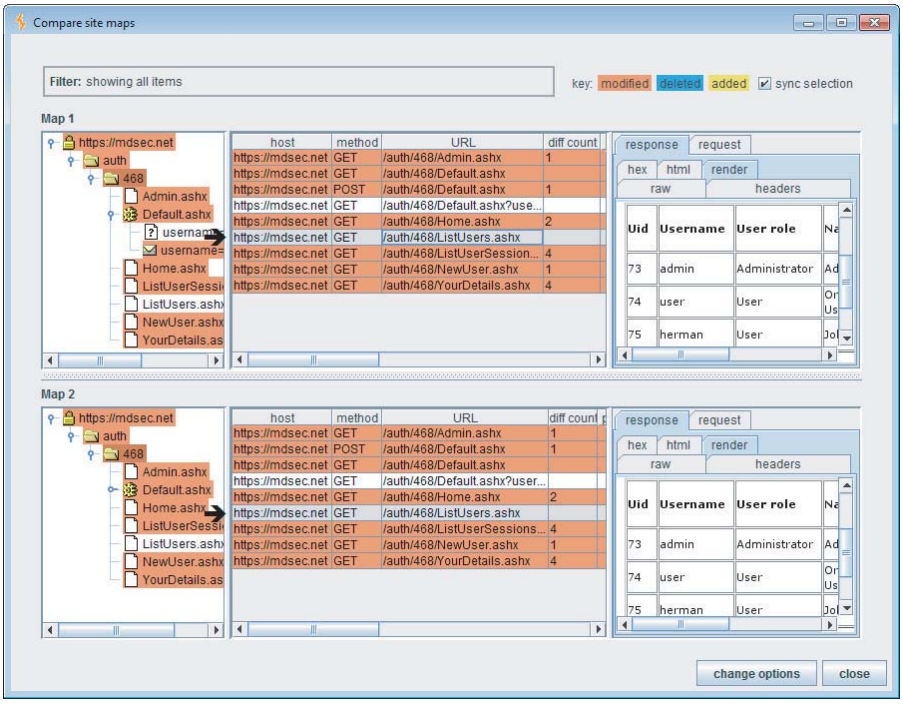

Figure 8-3: The low-privileged user can access the administrative function to list application users

TRY IT!

```
http://mdsec.net/auth/462/
http://mdsec.net/auth/468/
```

Testing Multistage Processes

The approach described in the preceding section — comparing the application's contents when accessed in different user contexts — is ineffective when testing some multistage processes. Here, to perform an action, the user typically must make several requests in the correct sequence, with the application building some state about the user's actions as he or she does so. Simply rerequesting each of the items in a site map may fail to replicate the process correctly, so the attempted action may fail for reasons other than the use of access controls.

For example, consider an administrative function to add a new application user. This may involve several steps, including loading the form to add a user, submitting the form with details of the new user, reviewing these details, and confirming the action. In some cases, the application may protect access to the initial form but fail to protect the page that handles the form submission or the confirmation page. The overall process may involve numerous requests, including redirections, with parameters submitted at earlier stages being retransmitted later via the client side. Every step of this process needs to be tested individually, to confirm whether access controls are being applied correctly.

TRY IT!

```
http://mdsec.net/auth/471/
```

HACK STEPS

1. **When an action is carried out in a multistep way, involving several different requests from client to server, test each request individually to determine whether access controls have been applied to it. Be sure to include every request, including form submissions, the following of redirections, and any unparameterized requests.**
2. **Try to find any locations where the application effectively assumes that if you have reached a particular point, you must have arrived via legitimate means. Try to reach that point in other ways using a lower-privileged account to detect if any privilege escalation attacks are possible.**

Continued

HACK STEPS *(CONTINUED)*

3. One way to perform this testing manually is to walk through a protected multistage process several times in your browser and use your proxy to switch the session token supplied in different requests to that of a less-privileged user.
4. You can often dramatically speed up this process by using the "request in browser" feature of Burp Suite:
   a. Use the higher-privileged account to walk through the entire multi-stage process.
   b. Log in to the application using the lower-privileged account (or none at all).
   c. In the Burp Proxy history, find the sequence of requests that were made when the multistage process was performed as a more privileged user. For each request in the sequence, select the context menu item "request in browser in current browser session," as shown in Figure 8-4. Paste the provided URL into your browser that is logged in as the lower-privileged user.
   d. If the application lets you, follow through the remainder of the multi-stage process in the normal way, using your browser.
   e. View the result within both the browser and the proxy history to determine whether it successfully performed the privileged action.

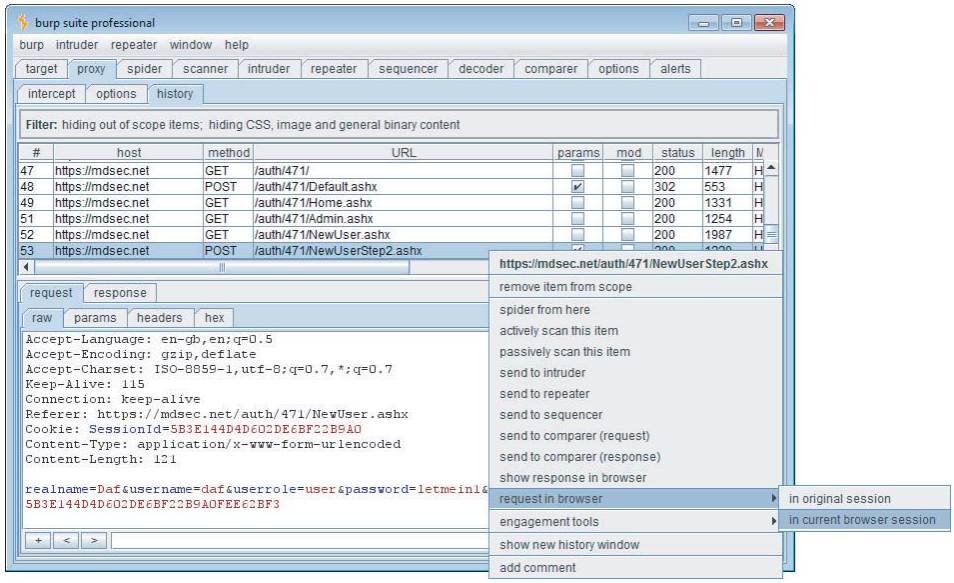

Figure 8-4: Using Burp to request a given item within the current browser session

When you select Burp's "request in browser in current browser session" feature for a specified request, Burp gives you a unique URL targeting Burp's internal web server, which you paste into your browser's address bar. When your browser requests this URL, Burp returns a redirection to the originally specified URL. When your browser follows the redirection, Burp replaces the request with the one you originally specified, while leaving the `Cookie` header intact. If you are testing different user contexts, you can speed up this process. Log in to several different browsers as different users, and paste the URL into each browser to see how the request is handled for the user who is logged in using that browser. (Note that because cookies generally are shared between different windows of the same browser, you normally will need to use different browser products, or browsers on different machines, to perform this test.)

TIP **When you are testing multistage processes in different user contexts, it is sometimes helpful to review the sequences of requests that are made by different users side-by-side to identify subtle differences that may merit further investigation.**

If you are using separate browsers to access the application as different users, you can create a different proxy listener in Burp for use by each browser (you need to update your proxy configuration in each browser to point to the relevant listener). Then, for each browser, use the context menu on the proxy history to open a new history window, and set a display filter to show only requests from the relevant proxy listener.

Testing with Limited Access

If you have only one user-level account with which to access the application (or none at all), additional work needs to be done to test the effectiveness of access controls. In fact, to perform a fully comprehensive test, further work needs to be done in any case. Poorly protected functionality may exist that is not explicitly linked from the interface of any application user. For example, perhaps old functionality has not yet been removed, or new functionality has been deployed but has not yet been published to users.

HACK STEPS

1. **Use the content discovery techniques described in Chapter 4 to identify as much of the application's functionality as possible. Performing this exercise as a low-privileged user is often sufficient to both enumerate and gain direct access to sensitive functionality.**

Continued

HACK STEPS *(CONTINUED)*

2. **Where application pages are identified that are likely to present different functionality or links to ordinary and administrative users (for example, Control Panel or My Home Page), try adding parameters such as `admin=true` to the URL query string and the body of `POST` requests. This will help you determine whether this uncovers or gives access to any additional functionality than your user context has normal access to.**
3. **Test whether the application uses the `Referer` header as the basis for making access control decisions. For key application functions that you are authorized to access, try removing or modifying the `Referer` header, and determine whether your request is still successful. If not, the application may be trusting the `Referer` header in an unsafe way. If you scan requests using Burp's active scanner, Burp tries to remove the `Referer` header from each request and informs you if this appears to make a systematic and relevant difference to the application's response.**
4. **Review all client-side HTML and scripts to find references to hidden functionality or functionality that can be manipulated on the client side, such as script-based user interfaces. Also, decompile all browser extension components as described in Chapter 5 to discover any references to server-side functionality.**

TRY IT!

```
http://mdsec.net/auth/477/
http://mdsec.net/auth/472/
http://mdsec.net/auth/466/
```

When all accessible functionality has been enumerated, you need to test whether per-user segregation of access to resources is being correctly enforced. In every instance where the application grants users access to a subset of a wider range of resources of the same type (such as documents, orders, e-mails, and personal details), there may be opportunities for one user to gain unauthorized access to other resources.

HACK STEPS

1. **Where the application uses identifiers of any kind (document IDs, account numbers, order references) to specify which resource a user is requesting, attempt to discover the identifiers for resources to which you do not have authorized access.**

2. If it is possible to generate a series of such identifiers in quick succession (for example, by creating multiple new documents or orders), use the techniques described in Chapter 7 for session tokens to try to discover any predictable sequences in the identifiers the application produces.
3. If it is not possible to generate any new identifiers, you are restricted to analyzing the identifiers you have already discovered, or even using plain guesswork. If the identifier has the form of a GUID, it is unlikely that any attempts based on guessing will be successful. However, if it is a relatively small number, try other numbers in close range, or random numbers with the same number of digits.
4. If access controls are found to be broken, and resource identifiers are found to be predictable, you can mount an automated attack to harvest sensitive resources and information from the application. Use the techniques described in Chapter 14 to design a bespoke automated attack to retrieve the data you require.

A catastrophic vulnerability of this kind occurs where an Account Information page displays a user's personal details together with his username and password. Although the password typically is masked on-screen, it is nevertheless transmitted in full to the browser. Here, you can often quickly iterate through the full range of account identifiers to harvest the login credentials of all users, including administrators. Figure 8-5 shows Burp Intruder being used to carry out a successful attack of this kind.

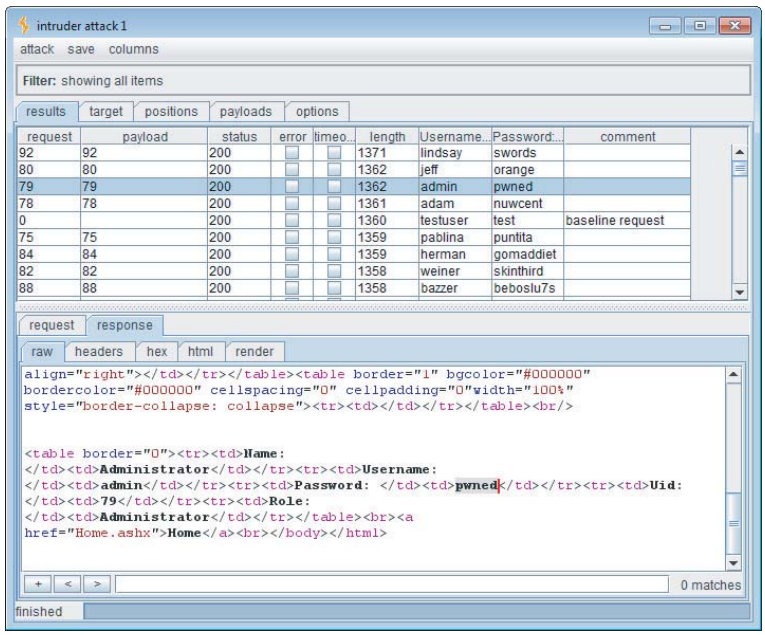

Figure 8-5: A successful attack to harvest usernames and passwords via an access control vulnerability

TRY IT!

```
http://mdsec.net/auth/488/
http://mdsec.net/auth/494/
```

TIP When you detect an access control vulnerability, an immediate attack to follow up with is to attempt to escalate your privileges further by compromising a user account that has administrative privileges. You can use various tricks to locate an administrative account. Using an access control flaw like the one illustrated, you may harvest hundreds of user credentials and not relish the task of logging in manually as every user until you find an administrator. However, when accounts are identified by a sequential numeric ID, it is common to find that the lowest account numbers are assigned to administrators. Logging in as the first few users who were registered with the application often identifies an administrator. If this approach fails, an effective method is to find a function within the application where access is properly segregated horizontally, such as the main home page presented to each user. Write a script to log in using each set of captured credentials, and then try to access your own home page. It is likely that administrative users can view every user's home page, so you will immediately detect when an administrative account is being used.

Testing Direct Access to Methods

Where an application uses requests that give direct access to server-side API methods, any access control weaknesses within those methods normally are identified using the methodology already described. However, you should also test for the existence of additional APIs that may not be properly protected.

For example, a servlet may be invoked using the following request:

```
POST /svc HTTP/1.1
Accept-Encoding: gzip, deflate
Host: wahh-app
Content-Length: 37

servlet=com.ibm.ws.webcontainer.httpsession.IBMTrackerDebug
```

Since this is a well-known servlet, perhaps you can access other servlets to perform unauthorized actions.

HACK STEPS

1. Identify any parameters that follow Java naming conventions (for example, `get`, `set`, `add`, `update`, `is`, or `has` followed by a capitalized word), or explicitly specify a package structure (for example, `com.`*`companyname`*`.`*`xxx`*`.`*`yyy`*`.`*`ClassName`*). Make a note of all referenced methods you can find.
2. Look out for a method that lists the available interfaces or methods. Check through your proxy history to see if it has been called as part of the application's normal communication. If not, try to guess it using the observed naming convention.
3. Consult public resources such as search engines and forum sites to determine any other methods that might be accessible.
4. Use the techniques described in Chapter 4 to guess other method names.
5. Attempt to access all methods gathered using a variety of user account types, including unauthenticated access.
6. If you do not know the number or types of arguments expected by some methods, look for methods that are less likely to take arguments, such as `listInterfaces` and `getAllUsersInRoles`.

Testing Controls Over Static Resources

In cases where static resources that the application is protecting are ultimately accessed directly via URLs to the resource files themselves, you should test whether it is possible for unauthorized users to simply request these URLs directly.

HACK STEPS

1. Step through the normal process for gaining access to a protected static resource to obtain an example of the URL by which it is ultimately retrieved.
2. Using a different user context (for example, a less-privileged user or an account that has not made a required purchase), attempt to access the resource directly using the URL you have identified.
3. If this attack succeeds, try to understand the naming scheme being used for protected static files. If possible, construct an automated attack to trawl for content that may be useful or that may contain sensitive data (see Chapter 14).

Testing Restrictions on HTTP Methods

Although there may not be a ready means of detecting whether an application's access controls make use of platform-level controls over HTTP methods, you can take some simple steps to identify any vulnerabilities.

HACK STEPS

1. **Using a high-privileged account, identify some privileged requests that perform sensitive actions, such as adding a new user or changing a user's security role.**
2. **If these requests are not protected by any anti-CSRF tokens or similar features (see Chapter 13), use the high-privileged account to determine whether the application still carries out the requested action if the HTTP method is modified. Test the following HTTP methods:**
   - **`POST`**
   - **`GET`**
   - **`HEAD`**
   - **An arbitrary invalid HTTP method**
3. **If the application honors any requests using different HTTP methods than the original method, test the access controls over those requests using the standard methodology already described, using accounts with lower privileges.**

Securing Access Controls

Access controls are one of the easiest areas of web application security to understand, although you must carefully apply a well-informed, thorough methodology when implementing them.

First, you should avoid several obvious pitfalls. These usually arise from ignorance about the essential requirements of effective access control or flawed assumptions about the kinds of requests that users will make and against which the application needs to defend itself:

- Do not rely on users' ignorance of application URLs or the identifiers used to specify application resources, such as account numbers and document IDs. Assume that users know every application URL and identifier, and ensure that the application's access controls alone are sufficient to prevent unauthorized access.

- Do not trust any user-submitted parameters to signify access rights (such as `admin=true`).
- Do not assume that users will access application pages in the intended sequence. Do not assume that because users cannot access the Edit Users page, they cannot reach the Edit User *X* page that is linked from it.
- Do not trust the user not to tamper with any data that is transmitted via the client. If some user-submitted data has been validated and then is transmitted via the client, do not rely on the retransmitted value without revalidation.

The following represents a best-practice approach to implementing effective access controls within web applications:

- Explicitly evaluate and document the access control requirements for every unit of application functionality. This needs to include both who can legitimately use the function and what resources individual users may access via the function.
- Drive all access control decisions from the user's session.
- Use a central application component to check access controls.
- Process every client request via this component to validate that the user making the request is permitted to access the functionality and resources being requested.
- Use programmatic techniques to ensure that there are no exceptions to the previous point. An effective approach is to mandate that every application page must implement an interface that is queried by the central access control mechanism. If you force developers to explicitly code access control logic into every page, there can be no excuse for omissions.
- For particularly sensitive functionality, such as administrative pages, you can further restrict access by IP address to ensure that only users from a specific network range can access the functionality, regardless of their login status.
- If static content needs to be protected, there are two methods of providing access control. First, static files can be accessed indirectly by passing a filename to a dynamic server-side page that implements relevant access control logic. Second, direct access to static files can be controlled using HTTP authentication or other features of the application server to wrap the incoming request and check the resource's permissions before access is granted.
- Identifiers specifying which resource a user wants to access are vulnerable to tampering whenever they are transmitted via the client. The server

should trust only the integrity of server-side data. Any time these identifiers are transmitted via the client, they need to be revalidated to ensure that the user is authorized to access the requested resource.

- For security-critical application functions such as the creation of a new bill payee in a banking application, consider implementing per-transaction reauthentication and dual authorization to provide additional assurance that the function is not being used by an unauthorized party. This also mitigates the consequences of other possible attacks, such as session hijacking.
- Log every event where sensitive data is accessed or a sensitive action is performed. These logs will enable potential access control breaches to be detected and investigated.

Web application developers often implement access control functions on a piecemeal basis. They add code to individual pages in cases where some access control is required, and they often cut and paste the same code between pages to implement similar requirements. This approach carries an inherent risk of defects in the resulting access control mechanism. Many cases are overlooked where controls are required, controls designed for one area may not operate in the intended way in another area, and modifications made elsewhere within the application may break existing controls by violating assumptions made by them.

In contrast to this approach, the previously described method of using a central application component to enforce access controls has many benefits:

- It increases the clarity of access controls within the application, enabling different developers to quickly understand the controls implemented by others.
- It makes maintainability more efficient and reliable. Most changes need to be applied only once, to a single shared component, and do not need to be cut and pasted to multiple locations.
- It improves adaptability. Where new access control requirements arise, they can be easily reflected within an existing API implemented by each application page.
- It results in fewer mistakes and omissions than if access control code is implemented piecemeal throughout the application.

A Multilayered Privilege Model

Issues relating to access apply not only to the web application itself but also to the other infrastructure tiers that lie beneath it — in particular, the application server, the database, and the operating system. Taking a defense-in-depth approach to security entails implementing access controls at each of these layers

to create several layers of protection. This provides greater assurance against threats of unauthorized access, because if an attacker succeeds at compromising defenses at one layer, the attack may yet be blocked by defenses at another layer.

In addition to implementing effective access controls within the web application itself, as already described, a multilayered approach can be applied in various ways to the components that underlie the application:

- The application server can be used to control access to entire URL paths on the basis of user roles that are defined at the application server tier.
- The application can employ a different database account when carrying out the actions of different users. For users who should only be querying data (not updating it), an account with read-only privileges should be used.
- Fine-grained control over access to different database tables can be implemented within the database itself, using a table of privileges.
- The operating system accounts used to run each component in the infrastructure can be restricted to the least powerful privileges that the component actually requires.

In a complex, security-critical application, layered defenses of this kind can be devised with the help of a matrix defining the different user roles within the application and the different privileges, at each tier, that should be assigned to each role. Figure 8-6 is a partial example of a privilege matrix for a complex application.

	Application Server	Application Roles	Database Privileges													
User type	**URL path**	**User role**	Search	Create Application	Edit Application	Purge Application	View Applications	Policy Updates	Rate Adjustment	View User Accounts	Create Users	View Company Ac	Edit Company Ac	Create Company	View Audit Log	Delegate privilege
Administrator	/*	Site Administrator	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		Support	✓		✓		✓	✓		✓	✓	✓	✓	✓		
Site Supervisor	/admin/* /myQuotes/* /help/*	Back Office – New business		✓			✓									
		Back Office – Referrals		✓	✓	✓		✓	✓							
		Back Office – Helpdesk	✓				✓			✓		✓			✓	✓
Company Administrator	/myQuotes/* /help/*	Customer – Administrator		✓	✓	✓	✓				✓	✓	✓			✓
		Customer – New Business		✓		✓	✓									
		Customer – Support	✓				✓			✓						
Normal User	/myQuotes/dash.jsp /myQuotes/apply.jsp /myQuotes/search.jsp /help/*	User – Applications	✓	✓			✓									
		User – Referrals														
		User – Helpdesk														
		Unregistered (Read Only)	✓				✓									
Audit	(none)	Syslog Server Account													✓	

Figure 8-6: A privilege matrix for a complex application

Within a security model of this kind, you can see how various useful access control concepts can be applied:

- Programmatic control — The matrix of individual database privileges is stored in a table within the database and is applied programmatically to enforce access control decisions. The classification of user roles provides a shortcut for applying certain access control checks, and this is also applied programmatically. Programmatic controls can be extremely fine-grained and can build arbitrarily complex logic into the process of carrying out access control decisions within the application.
- Discretionary access control (DAC) — Administrators can delegate their privileges to other users in relation to specific resources they own, employing discretionary access control. This is a *closed DAC* model, in which access is denied unless explicitly granted. Administrators also can lock or expire individual user accounts. This is an *open DAC* model, in which access is permitted unless explicitly withdrawn. Various application users have privileges to create user accounts, again applying discretionary access control.
- Role-based access control (RBAC) — Named roles contain different sets of specific privileges, and each user is assigned to one of these roles. This serves as a shortcut for assigning and enforcing different privileges and is necessary to help manage access control in complex applications. Using roles to perform up-front access checks on user requests enables many unauthorized requests to be quickly rejected with a minimum amount of processing being performed. An example of this approach is protecting the URL paths that specific types of users may access.

  When designing role-based access control mechanisms, you must balance the number of roles so that they remain a useful tool to help manage privileges within the application. If too many fine-grained roles are created, the number of different roles becomes unwieldy, and they are difficult to manage accurately. If too few roles are created, the resulting roles will be a coarse instrument for managing access. It is likely that individual users will be assigned privileges that are not strictly necessary to perform their function.

  If platform-level controls are used to restrict access to different application roles based on HTTP method and URL, these should be designed using a default-deny model, as is best practice for firewall rules. This should include various specific rules that assign certain HTTP methods and URLs to certain roles, and the final rule should deny any request that does not match a previous rule.
- Declarative control — The application uses restricted database accounts when accessing the database. It employs different accounts for different groups of users, with each account having the least level of privilege

necessary to carry out the actions that group is permitted to perform. Declarative controls of this kind are declared from outside the application. This is a useful application of defense-in-depth principles, because privileges are imposed on the application by a different component. Even if a user finds a way to breach the access controls implemented within the application tier in order to perform a sensitive action, such as adding a new user, he is prevented from doing so. The database account that he is using does not have the required privileges within the database.

A different means of applying declarative access control exists at the application server level, via deployment descriptor files, which are applied during application deployment. However, these can be relatively blunt instruments and do not always scale well to manage fine-grained privileges in a large application.

HACK STEPS

If you are attacking an application that employs a multilayered privilege model of this kind, it is likely that many of the most obvious mistakes that are commonly made in applying access controls will be defended against. You may find that circumventing the controls implemented within the application does not get you very far, because of protection in place at other layers. With this in mind, several potential lines of attack are still available to you. Most importantly, understanding the limitations of each type of control, in terms of the protection it does not offer, will help you identify the vulnerabilities that are most likely to affect it:

- **Programmatic checks within the application layer may be susceptible to injection-based attacks.**
- **Roles defined at the application server layer are often coarsely defined and may be incomplete.**
- **Where application components run using low-privileged operating system accounts, typically they can read many kinds of potentially sensitive data within the host file system. Any vulnerabilities granting arbitrary file access may still be usefully exploited, even if only to read sensitive data.**
- **Vulnerabilities within the application server software itself typically enable you to defeat all access controls implemented within the application layer, but you may still have limited access to the database and operating system.**
- **A single exploitable access control vulnerability in the right location may still provide a starting point for serious privilege escalation. For example, if you discover a way to modify the role associated with your account, you may find that logging in again with that account gives you enhanced access at both the application and database layers.**

Summary

Access control defects can manifest themselves in various ways. In some cases, they may be uninteresting, allowing illegitimate access to a harmless function that cannot be leveraged to escalate privileges any further. In other cases, finding a weakness in access controls can quickly lead to a complete compromise of the application.

Flaws in access control can arise from various sources. A poor application design may make it difficult or impossible to check for unauthorized access, a simple oversight may leave only one or two functions unprotected, or defective assumptions about how users will behave can leave the application undefended when those assumptions are violated.

In many cases, finding a break in access controls is almost trivial. You simply request a common administrative URL and gain direct access to the functionality. In other cases, it may be very hard, and subtle defects may lurk deep within application logic, particularly in complex, high-security applications. The most important lesson when attacking access controls is to look everywhere. If you are struggling to make progress, be patient, and test every step of every application function. A bug that allows you to own the entire application may be just around the corner.

Questions

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

1. An application may use the HTTP `Referer` header to control access without any overt indication of this in its normal behavior. How can you test for this weakness?
2. You log in to an application and are redirected to the following URL:

   ```
   https://wahh-app.com/MyAccount.php?uid=1241126841
   ```

   The application appears to be passing a user identifier to the `MyAccount.php` page. The only identifier you are aware of is your own. How can you test whether the application is using this parameter to enforce access controls in an unsafe way?
3. A web application on the Internet enforces access controls by examining users' source IP addresses. Why is this behavior potentially flawed?

4. An application's sole purpose is to provide a searchable repository of information for use by members of the public. There are no authentication or session-handling mechanisms. What access controls should be implemented within the application?
5. When browsing an application, you encounter several sensitive resources that need to be protected from unauthorized access and that have the `.xls` file extension. Why should these immediately catch your attention?