1 Access Control

One of the most basic forms of protection that any web application must utilize is the enforcement of an authentication and authorization policy.

Authentication deals with identifying users to the application; in APEX this is provided by a number of default authentication schemes and can be extended using a custom authentication scheme. Authorization is the process of assessing whether the authenticated user is privileged to access certain data or perform a particular action.

The term *access control* covers both aspects, and access-control vulnerabilities arise when either authentication can be abused to allow access to an application without valid credentials, or when authorization is incorrectly applied, allowing valid users to access parts of the application for which they should not have privileges.

One of the great things about APEX is the capability to apply authorization schemes to a wide range of components. At a simple level, pages within an APEX application can be protected by your authorization scheme to prevent access to certain sets of users. The applicability of authorization schemes is a lot more granular: reports, buttons, and processes can all also be protected. Users with different privileges can then only view or access specific components on a page. While APEX provides a great access control model, there are some common mistakes that are made where data and functionality do not get protected as you might expect. This chapter will guide you through the various access control features and show how they can be used securely in your applications.

THE PROBLEM

When authentication or authorization is not applied correctly, an unauthenticated user with no access to the application may be able to view and interact with the data it is intended to protect. Valid (but malicious) users of the application may also be able to invoke operations that should be restricted to a limited subset of users.

In our experience performing security assessments of APEX applications, we can say that although APEX provides fantastic flexibility and granularity with authorization, in many cases such protection is not defined or applied correctly. As an APEX application grows and matures, we often see newer pages and components that do not have the protection they require. In one (extreme!) case, we analyzed an application where the Create Admin User page was not protected, and could be accessed by any authenticated user of the application.

THE SOLUTION

By ensuring that the authentication scheme used by your APEX application is robust and conforms to best practice, you can be confident that only legitimate users of the application should have access. Of course, other attacks against an APEX application can allow those malicious attackers to get in even when authentication is defined correctly, but these attacks (such as using Cross-Site Scripting to steal a valid user's credentials, or SQL Injection to access arbitrary data within the database) can be mitigated in other ways and are discussed later in this book.

Authorization should be applied to those areas within an application that need to be protected from subsets of valid authenticated users. Only very simple applications are designed with one generic user level; most have at least some notion of "role" with base-level users, and administrative functionality for a specific group of users.

We're not going to cover designing and documenting an application's access-control model, as this is very dependent on the specific requirements of the application. However, this is a crucial step when developing any system. Such requirements should be captured when the system is planned, and then once implemented, the access-control structure can be compared with the initial intentions.

Instead, we present some common access-control mishaps that we've observed across a number of APEX applications, and discuss how the simple addition of access-control settings can secure the APEX application.

AUTHENTICATION

The first stage is to define a reasonable authentication scheme for the application. In general, any authentication scheme should be capable of identifying users based on some description of who they are (their username) and a secret that nobody except the user should know (such as a password).

Depending on the requirements of the APEX application, you define authentication using one of the built-in methods or via a custom scheme, as shown in Figure 1-1.

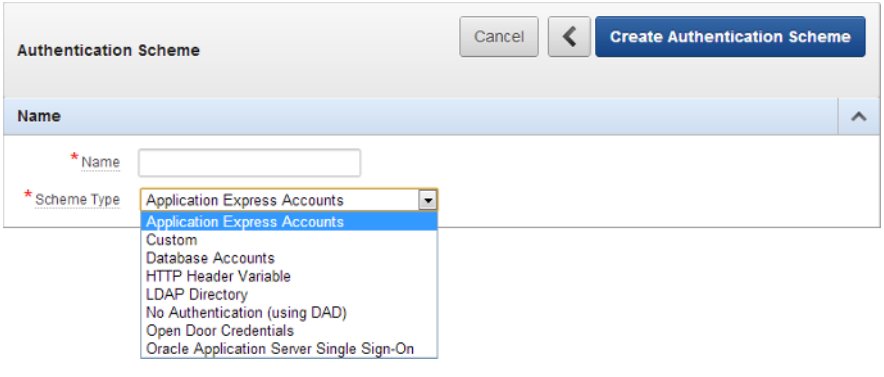

FIGURE 1-1: Available authentication schemes

No rules exist for which of these schemes to use or avoid (although choosing Open Door Credentials would require confidence that the data and operations of the application were truly intended for everybody).

When authenticating users based on the traditional credentials of username and password, here is some "best practice" guidance that you should consider:

- **Account lockout:** If a user attempts authentication with an invalid password a number of times, consider rejecting future access for a certain period (the chosen threshold and timeout depends on the sensitivity of the application and the corporate security policy).
- **Password complexity:** Users invariably choose the simplest password they can, so an application should enforce a level of complexity so attackers cannot guess valid user credentials (again, the chosen policy depends on the application).
- **Password reset:** Where an application allows users to reset their password if they forget, it should either require some additional confirmation or send a reset link with a unique token to their configured e-mail address. The application should not allow a reset based on some publicly available information (for example, birth date or mother's maiden name), and should never e-mail users their actual password.
- **Password storage:** The application should not store user credentials in clear text, but instead should store passwords that are cryptographically "hashed" and preferably "salted" with a unique value. This limits the damage of the worst-case-scenario of your account information being compromised, because an attacker would still not be able to authenticate as other users without "cracking" the password hashes. Storing passwords that are encrypted, rather than hashed, is not considered good practice because they can be decrypted should the key be discovered.

With authentication defined and adhering to these guidelines and applied to an APEX application, any non-public page should be protected so that only legitimate users have access. This is the first part of the story of access control; the next stage is applying authorization to provide more granular control over the functionality available to users.

Application Authentication

You can define the authentication scheme in the Security section of an APEX application's properties, as shown in Figure 1-2. This scheme is used whenever a page that requires authentication is requested by a user who is not logged in. It is possible to specify No Authentication, effectively making all pages publicly accessible; needless to say, you should not use this without very careful consideration about the data and features within an application.

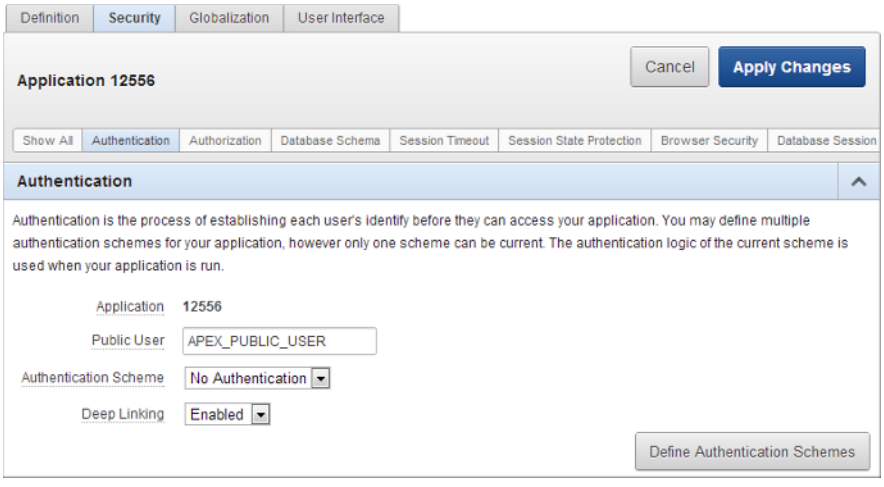

FIGURE 1-2: Application authentication settings

Page Authentication

You can apply authentication to pages within an APEX application via the Security section of the page properties, as shown in Figure 1-3.

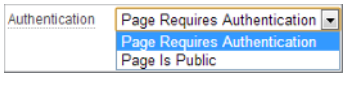

FIGURE 1-3: Setting page authentication

This setting dictates simply whether a user needs to be authenticated to access the page. If a page doesn't require authentication, it is considered a *public page*.

Generally, an application requires only a single public page: the login page. Having more public pages is not a security problem as long as those pages contain only information and functions that are really intended for access by anyone whose browser can reach the application.

Given page numbering is generally sequential in APEX applications, public pages can be trivially enumerated by a simple attack that iterates through the pages of the application. Do not assume that because some public page is not immediately obvious to visitors of the site that it will not be found by a more investigative user!

Reducing the public pages in an application serves to reduce the attack surface that is available to hackers looking to break into the site; as always, unless it really has to be public, make sure that the page requires authentication.

AUTHORIZATION

Once users are identified through authentication, the application can continue to make access-control decisions, to limit access to certain sections or functionality. This is what *authorization schemes* are used for within an APEX application. As a developer, you can define a number of schemes based on the complexity of the required access-control model. Generally, an authorization scheme would check the groups that a user is a member of, or query some privileges table to ascertain the roles and permissions for the user.

For the purposes of this chapter, define a dummy authorization scheme (see Figure 1-4) called ***`IS_USER_AN_ADMIN`*** that you can apply to various areas to observe the effect of this form of access control. This scheme will always return false (because the `ISADMIN` item is not defined), but demonstrates certain attacks that could occur when authorization is not applied correctly.

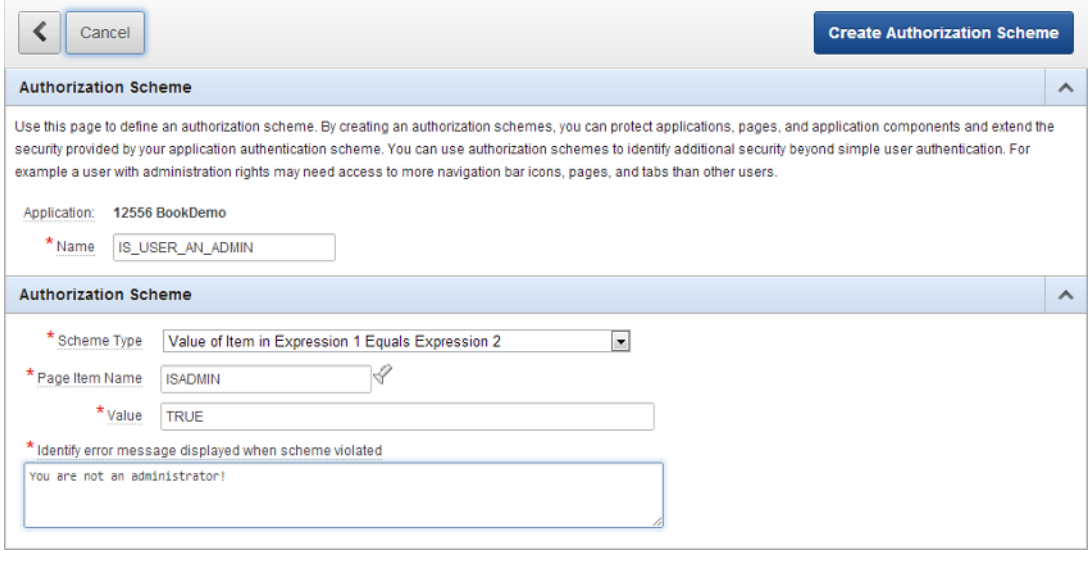

FIGURE 1-4: Create a dummy authorization scheme to experiment with

Application Authorization

You can apply an authorization scheme at an application level to be enforced across all (non-public) pages. Optionally, this can cover public pages also, although that somewhat defeats the purpose of marking them as public.

With minimal public pages and an application-level authorization scheme, the APEX application is well protected against unauthenticated (anonymous) users. Applying authorization at this level can also defend against the accidental creation of pages that are not configured to require authentication.

Page Authorization

The authorization scheme setting by default has two options: either the page does not require authorization, or only non-public users can access the page. With authentication defined (Page Requires Authentication), these two settings are equal.

TIP *There is a reason to specifically choose Must Not Be Public User even in this case. Because the default authorization is No Page Authorization Required, if you explicitly set it to Must Not Be Public User, it shows you have considered the security of the page and have made the conscious decision that this page is accessible to every authenticated user of the application.*

Though not strictly required, this step assists the security review process because pages with No Page Authorization Required are most likely pages where the developer has not considered the authorization requirements, and potentially the content needs to be protected further.

Authorization gets more interesting when you add a custom authorization scheme (see Figure 1-5).

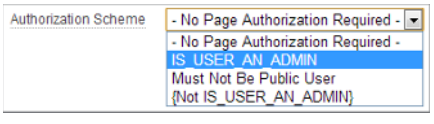

FIGURE 1-5: Applying authorization to a page

With the `IS_USER_AN_ADMIN` authorization scheme defined in the application, you can now specify that the page should be available only to users who pass the checks implemented by the scheme.

The access-control structure of a general application would therefore be as follows:

- **Login page:** Public
- **All other pages:** Must Not Be Public User
- **Administrative pages:** Defined with a custom authorization scheme

More complex APEX applications implementing many user roles would apply the relevant more granular authorization scheme to pages.

You can define the same authorization scheme attribute for regions of a page, so that the displayed page differs based on user privileges.

It is also possible to use Conditions on page components as a form of authorization. See Figure 1-6.

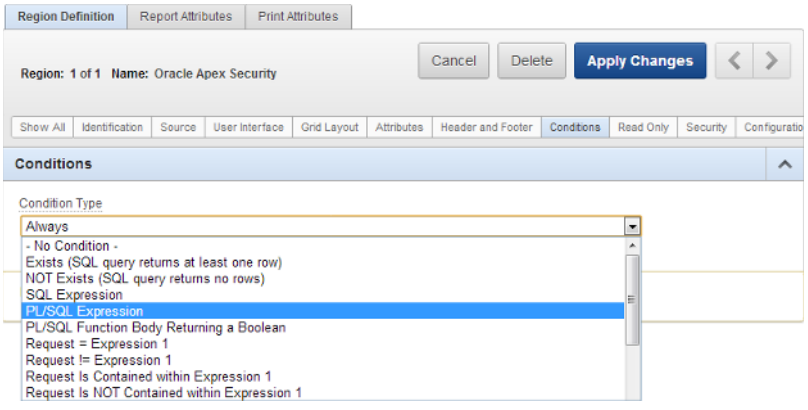

FIGURE 1-6: Conditions

There is nothing necessarily incorrect about using Conditions in this way, except perhaps that it is not immediately obvious that the Condition is acting as part of the APEX application's security boundary.

TIP *Unless there is a valid reason not to, the application's access-control model should be enforced using the security attributes rather than as a condition.*

Button and Process Authorization

In APEX, a process can be defined on a page that operates On Submit, when the HTML form contained on the page is submitted. These processes execute whenever the page is submitted, unless linked to a specific button using the When Button Pressed attribute.

Imagine a page that is accessible to two different levels of user (say, any authenticated user and also an administrator). You might have a button that has access control so only the higher-privilege user can access some functionality (such as deleting some data). The page has a Delete Row process that occurs On Submit and is linked to the Delete button (using When Button Pressed).

By applying an admin-only authorization scheme to the button, APEX renders the button only when the user passes the authorization test.

This situation occurs often, and actually contains an access-control vulnerability. The crux of the problem is that the process is not protected by an authorization scheme. It is technically possible to invoke the imaginary Delete Row process without actually clicking the Delete button, through a JavaScript call.

WARNING *When applying security to a button, remember to also apply equal security constraints to the process that is invoked when the button is clicked.*

To demonstrate, create a blank page (20) with an HTML region, and two items: a button (`P20_BUTTON`) and a display-only item (`P20_STRING`).

Now create a process (`APPEND_STRING`) with a process point of "On Submit – After Computations and Validations," with the following process source:

```
begin
 select :P20_STRING || 'Recx!' into :P20_STRING from dual;
end;
```

Select *`P20_BUTTON`* for the When Button Pressed attribute to link this process's execution to occur when the button is clicked. See Figure 1-7.

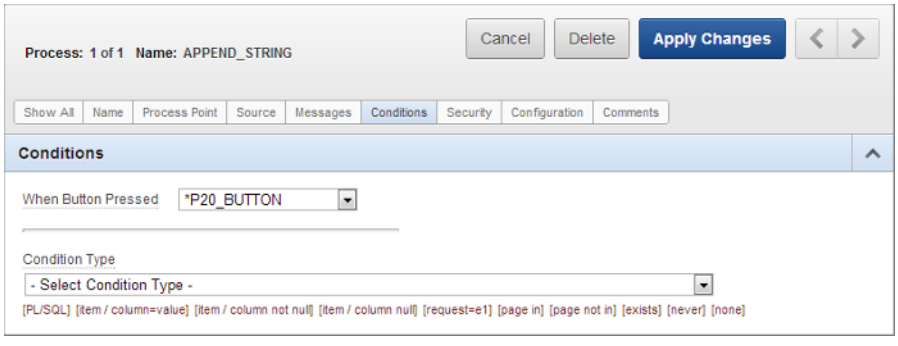

FIGURE 1-7: Setting to invoke the process when the button is clicked

The page should contain the button, the display-only item, and the process that is executed when the button is clicked, as shown in Figure 1-8.

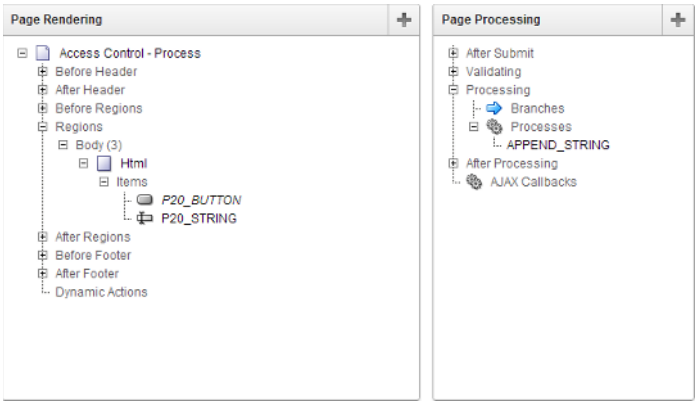

FIGURE 1-8: Page structure

The resulting page should now display a button and the empty *`P20_STRING`* item. When the button is clicked, the string is modified so your text is appended. We're using the simplest possible example here to get the core access-control issue across — in real-world applications we've seen this same structure implementing actions such as deleting data, modifying site content, and even disabling user accounts.

The button within the page is defined by the following HTML:

```
<input type="button" value="Button" onclick="apex.submit('P20_BUTTON');"
id="P20_BUTTON"/>
```

This means when the button is clicked, the JavaScript `apex.submit()` method is called.

If you now apply the *`IS_USER_AN_ADMIN`* dummy authorization scheme to this button and then run the page, the button is no longer displayed. See Figure 1-9.

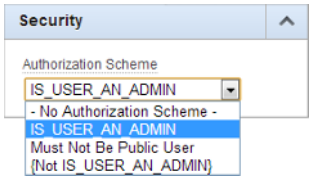

FIGURE 1-9: Apply an authorization scheme to the button

At first, it appears that this means the process can no longer be executed by non-administrative users. But, what an attacker can do is simulate a button click by executing the JavaScript in the browser's JavaScript console (even when the actual button definition does not appear in the HTML!). Figure 1-10 shows the simple JavaScript command that an attacker can enter into their browser.

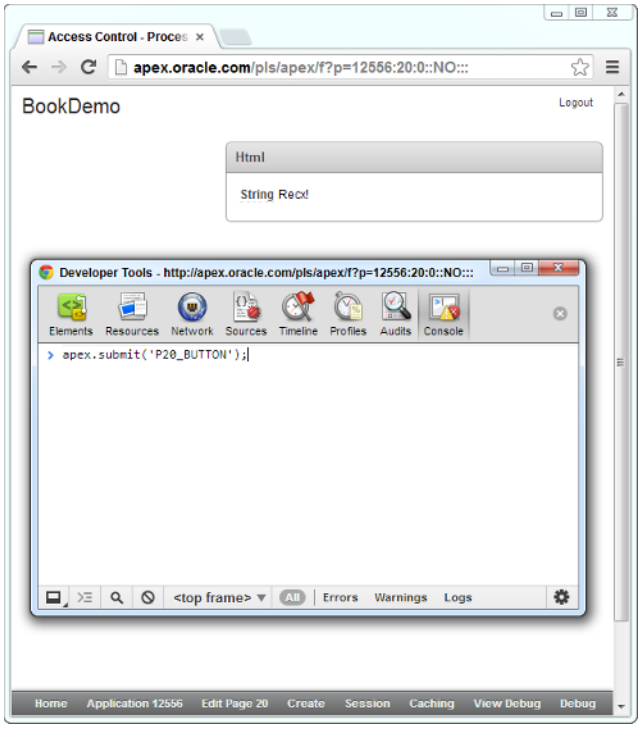

FIGURE 1-10: Forcing a button click using JavaScript

If you enter this JavaScript and press enter you will notice that the page refreshes. The displayed string is also now longer than before, indicating that the process has executed a second time, even without the physical click of the button.

The only caveat here is that an attacker would need to know in advance the name of the button. The access-control model of the APEX application should not rely simply on the unpredictability of a button name, and it would certainly be possible for an attacker to iterate through a list of likely button names.

To resolve the access-control vulnerability here, the process should have an authorization scheme that matches the button. When set, the preceding JavaScript still refreshes the page but the string output does not change, because the process is no longer executing.

NOTE *The same applies to the Validations and Branches that are linked to button presses, although generally there is less of a security impact if Validations or Branches can be executed by unprivileged users.*

NOTE *A similar attack against Dynamic Actions that execute server-side PL/SQL code is theoretically possible, but cannot realistically be performed by an attacker. A Dynamic Action that is protected with an authorization scheme means the JavaScript to invoke the action is not included in the page displayed in the browser, much like with the button in the preceding example. Although the code to hook up a dynamic action could be specified manually by an attacker in the JavaScript console as before, there is a complex* `ajaxIdentifier` *component that uniquely represents the Dynamic Action:*

```
"ajaxIdentifier":"D22C8577EE8C8066BA70874E0B814467D23F5CD274C23A349148DCB
297EF7295"
```

This value is actually encrypted with the widely used Advanced Encryption Standard (AES) algorithm, using a server-side secret as the key. Therefore this value cannot be determined by an attacker. Without this identifier the attacker cannot invoke the dynamic action, so the server-side PL/SQL code cannot be executed.

Process Authorization — On-Demand

Within the Shared Components section of an APEX application's definition are application processes (Figure 1-11). These application-wide processes can have access-control security concerns when they are defined as having a Process Point of On-Demand.

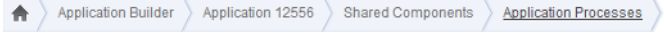

FIGURE 1-11: Application-level On-Demand processes

Create an application process called PrintHello that executes on-demand, and runs some PL/SQL to simply display a message as shown in Figure 1-12.

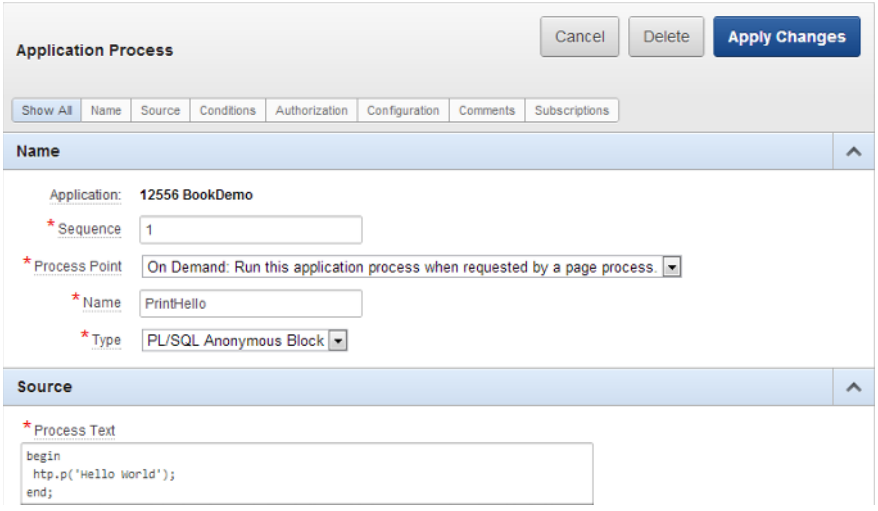

FIGURE 1-12: An example on-demand process

In APEX 4.2, a default authorization scheme is applied which requires users to be authenticated ("Must Not Be Public User").

For this example, edit the process and change the authorization scheme to No Authorization Required. This was the default for any application created in APEX prior to version 4.2, and the scheme will not be changed when these applications are imported and upgraded to APEX 4.2.

You can invoke the On-demand process via the URL on any accessible page:

```
f?p=12556:101:0:APPLICATION_PROCESS=PrintHello:::
```

You can also invoke it via an Ajax call in the browser's JavaScript console:

```
var get = new htmldb_Get(null,
 $x('pFlowId').value,
 'APPLICATION_PROCESS=PrintHello',
 101);
get.get();
```

Either way, the response is a simple HTML page with the "Hello World" message.

When no authorization scheme is applied, any on-demand application process can be invoked by an attacker, prior to authentication. All that is required is the name of the process, and one publicly accessible page (the login page 101 can generally be used). Again, the security of the APEX application should not only depend on the complexity of the name used.

The security threat posed by processes defined in this way depends on the implementation details of the PL/SQL within the process. Some APEX applications have had unprotected on-demand processes that list user accounts, send e-mails to users, and even contain SQL Injection vulnerabilities, giving unauthenticated attackers control over the data within the database!

The new default setting of Must Not Be Public User in APEX 4.2 reduces, but does not remove, this threat. This scheme applies to any authenticated user, and again depending on the implementation details of the process PL/SQL code, this could still represent an access-control vulnerability where the process performs some privileged action.

Resolving the issue is simply a matter of ensuring that all application processes that execute on-demand have appropriate authorization schemes applied, so they do not expose privileged functionality to unprivileged users.

TIP *When creating a process (under Shared Components, Application Processes), APEX 4.2 even suggests that on-demand processes should be created on pages, rather than as application shared items. When declared on a page, the On-Demand Process is accessible by users only if they can access the page, and this simplifies the access-control model by grouping similarly privileged actions together.*

Creating on-demand processes at a page level limits the chance that a process may be unintentionally accessible to some users.

File Upload

In APEX before version 4.0, any uploaded file content (received using the File Browse item type) was inserted into the *WWV_FLOW_FILES* table (also referred to as *APEX_APPLICATION_FILES*). File content was accessed using the p function on the URL with a single parameter representing the ID of the file:

```
p?n=2928618714505864969
```

In later versions of APEX, this method of receiving and accessing uploaded files is still possible, although you have another option of allowing storage in a custom table, as show in Figure 1-13.

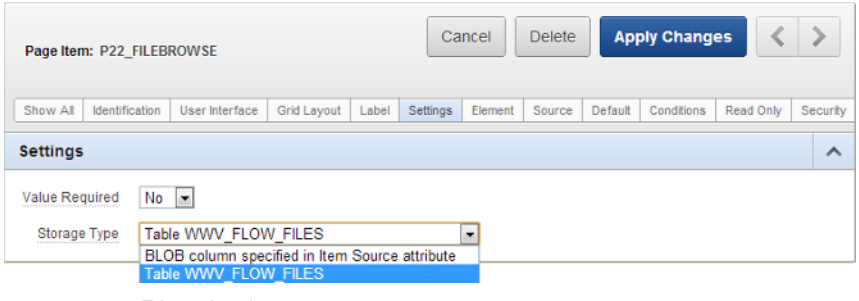

FIGURE 1-13: File upload storage options

Applications that use the *WWV_FLOW_FILES* table can exhibit access-control security issues.

There is a pattern to the values for the n parameter that represents the ID of the file that was uploaded. The following three values were captured by uploading files in quick succession:

```
p?n=2931268814589196184
p?n=2931268914935196284
p?n=2931269015281196367
```

There are three blocks that are incrementing within this identified: from left-to-right in the first example there are: 29312688, 14589, and 196184.

There was more of a delay between the first two requests than the second, and the distance between the final six digits is greater, suggesting a time-based sequence.

This suggests that the identifier for uploaded files is made of up three components:

- A incrementing counter
- A 5-digit number that increases in value
- A 6-digit number that increases in value

For an attacker this is interesting because he could keep uploading files until the value he expects in the first block is skipped, indicating that another user of the APEX environment has uploaded a file. For example, between the two following identifiers, the initial counter value 29316565 has been skipped, indicating that someone has uploaded content between the two upload requests:

```
p?n=2931656420176226210
p?n=2931656620523226290
```

A number of possible values for the full identifier of the upload exist, calculated as follows:

Total possibilities = (20523 − 20176 − 1) × (226290 − 226210 − 1) = 346 × 79 = 27,334

If the attacker makes a request for each possible identifier, he would (eventually) be able to access the file uploaded by the other user.

You have basically two concerns here:

- The unique identifier for uploaded files is sequential and potentially predictable.
- The method of accessing uploaded content (via a request to the `p` function in the URL) offers no mechanism of requiring authentication or enforcing authorization.

The Oracle documentation for older versions of APEX indicates that files uploaded to the *`WWV_FLOW_FILES`* table should not be left there:

"Note: Don't forget to delete the record in *`WWV_FLOW_FILES`* after you have copied it into another table."

The newer documentation recommends that the alternate binary large object (BLOB) storage mechanism is used, because otherwise unauthenticated access to uploaded files may be possible.

For an APEX application that deals with files uploaded by users, ensure the content has correct access control:

- For APEX version before 4.0, ensure that a page process copies the content from the *`WWV_FLOW_FILES`* table to another location and deletes the original row.
- For newer APEX versions, 4.0 and above, use the alternative BLOB storage mechanism.

SUMMARY

Access control is critical to any application's security, and APEX provides simple mechanisms to apply authentication and authorization to your applications.

For authentication, whichever mechanism you use, consider the following:

- Limit password guessing and dictionary attacks on user credentials (account lockout).
- Ensure users choose suitably complex passwords (password complexity).
- Users who forget their passwords should be able to regain access securely (password reset).
- Stored user credentials should not be immediately usable if they are compromised (password storage).

As well as authentication, an APEX application should apply authorization to protect areas so that only a subset of users has access. The authorization schemes should be designed to identify users based on their privilege or role. The schemes should then be applied throughout the application, to each page and component that requires access control.

Remember the following:

- Apply the authorization scheme Must Not Be Public User to any page that is really intended to be accessible to any authenticated user; this allows the security review process to quickly pick up pages that may require protection but have no authorization policy applied.
- Where possible, use the APEX authorization scheme attributes to protect pages and components, rather than using conditions, to ensure the security enforcement policy is clearly indicated.
- Ensure that processes linked to button clicks have matching authorization schemes, to prevent attacks from initiating processes even when the button is not displayed.
- Check all application-level on-demand processes to ensure they are protected with an authorization scheme to prevent unauthenticated users (or all authenticated users) from executing the process.
- When handling file uploads, avoid the `WWV_FLOW_FILES` table where possible; for older versions of APEX, remove content immediately after upload.