Assessing Oracle e-Business Suite 11i

David Litchfield
dlitchfield@google.com

25th November 2015

Oracle e-Business Suite 11i has been around for over 15 years and is still widely used (though admittedly, it's not as widespread as R12). One would think over that time all the security bugs would have been found and solved. Indeed, according to the "Secure Configuration Guide for Oracle e-Business 11i" [1] written by Oracle we are told on page 42:

"Of the many 'potential SQL Injections' we have seen reported we have yet to find a single confirmed example"

With that in mind the author set out to investigate whether this and other claims about EBS' security stood up to scrutiny. There are two key functional areas the author pegged as potential targets because they have a large attack surface, namely, the PL/SQL Gateway and default JSPs.

The PL/SQL Gateway accepts the name of a PL/SQL package and procedure and passes it to the database for execution - for example:

https://example.com/pls/EBS/foo.bar

where foo is the name of the PL/SQL package and bar is the name of the procedure. For e-Business suite, the package needs to be "enabled". This is done by adding the package name to the APPS.FND_ENABLED_PLSQL table and setting its ENABLED column to "Y". By default, there are around 800 enabled PL/SQL packages and procedures.

For JSPs there are around 15,000 JSP files.

For my investigation, I did not look at every enabled PL/SQL package nor did I review every single JSP file; I took a sampling of each. All told, I spent roughly 80 hours looking at the code and developing proof of concept exploits for each issue found. Some issues were easy to exploit whereas others were slightly harder requiring multiple steps.

My findings are as follows:

21 SQL injection flaws (all confirmed and exploitable - incidentally)
26 Cross site scripting issues
1 Open Redirect
2 Denial of Service issues

These were all reported to secalert_us@oracle.com as and when they were found so that a patch can be developed and delivered to customers. The patches were released in January 2016[2]. It is further hoped that my investigation has shown Oracle that their products are not as secure as they think and will give them the motivation to spend some time and resources doing their own review to weed out all the issues I did not find.

Exploiting the SQL Injection Flaw in ORACLESSWA

The EXECUTE procedure on ORACLESSWA takes a parameter E. The E parameter is decrypted using icx_call.decrypt and resolves to a function id (and responsibility ID etc). Consider the following URL:

https://example.com/pls/EBS/OracleSSWA.Execute?E=%7B!38FC0AD8B864E9292DA4180C5B96CE7534B905551F9EB138

Here, the value for the E parameter is "{!38FC0AD8B864E9292DA4180C5B96CE7534B905551F9EB138" which decrypts to "178*20873*0*2633**]". 2633 is the FUNCTION_ID for which the WEB_HTML_CALL is "ICX_CHANGE_LANGUAGE.show_languages". The function ID is passed to the ORACLEAPPS.RUNFUNCTION procedure.

The EXECUTE procedure on ORACLESSWA can also take another parameter, P. If P is not null, then it too is decrypted and passed to ORACLEAPPS.RUNFUNCTION along with the function ID.

The ORACLEAPPS.RUNFUNCTION procedure looks up the WEB_HTML_CALL in APPS.FND_FORM_FUNCTIONS and, if it's a PL/SQL call, executes it via DBMS_SQL in an anonymous block. First, however, if there are any parameters, i.e. the decrypted value for P, then they are unpacked and concatenated to the anonymous block. The decrypted value for P must be of the form X=Y. If an attacker passes an encrypted string via P that decrypts to ");htp.p(user);END;--=A" then this will be concatenated to the anonymous block. This is how the block appears before execution:

```
begin ICX_CHANGE_LANGUAGE.show_languages();htp.p(user);END;--=>'A') ; end;
```

This will execute, executing the attacker supplied htp.p(user) statement.

So if we encrypt ");htp.p(user);end;--=A" we get "{!76EF7B870B1E380618ED818959DC37F6FB9E6C44A14AC3D7". Setting the value for P as this and requesting it we can see "APPS" - the return value for the USER() function.

https://example.com/pls/EBSPROD/OracleSSWA.Execute?E=%7B!38FC0AD8B864E9292DA4180C5B96CE7534B905551F9EB138&P={!76EF7B870B1E380618ED818959DC37F6FB9E6C44A14AC3D7

To see the output “APPS” you may need to right click and view source. It’s down the bottom, the last entry on the page.

As another example, let's execute DBMS_AW.INTERP('SLEEP 10') and cause the application to hang for ten seconds:

https://example.com/pls/EBS/OracleSSWA.Execute?E=%7B!38FC0AD8B864E9292DA4180C5B96CE7534B905551F9EB138&P={!76EF7B870B1E38064AA6A73B965E2A75FC9703B291FF84F2CFB2999D647F9AB32E6C48A2B78C427C1A33A56A2EBAD348FC33BDC46F16DE24EECDD8A46E290840

Exploiting the SQL injection flaw in HR_UTIL_DISP_WEB

If we look at the source for the DEXL procedure on HR_UTIL_DISP_WEB we see it takes a parameter P_URL which is then decrypted and passed to parameter to HR_GENERAL_UTILITIES.Execute_Dynamic_SQL where it is executed as a block of anonymous PL/SQL:

As it happens, the P_URL parameter is simply a number. This number is a TEXT_ID record in the APPS.ICX_TEXT table and so when the icx_call.decrypt2 function executes, it simply returns the text column for the corresponding TEXT_ID.

If an attacker can add their own row into the APPS.ICX_TEXT table then they can use the DEXL procedure to execute arbitrary SQL as the APPS user. So that begs the question how does an attacker do this? The answer is of course find somewhere in the code that calls icx_call.encrypt2 on user supplied input. This will insert a row into APPS.ICX_TEXT with the attacker supplied SQL. Looking through the code further we see:

In the code above, if an attacker executes the display_fatal_errors procedure with the value for P_MESSAGE as their arbitrary SQL then it will be "encrypted" with the icx_call.encrypt2 function - which adds the attacker supplied SQL to the ICX_TEXT table. Further, the TEXT_ID number is returned to the user in the browser redirect via "window.location"

So, by chaining a few requests together an attacker can execute arbitrary SQL as the highly privileged APPS user.

First the attacker requests:

https://example.com/pls/ebs/HR_UTIL_DISP_WEB.display_fatal_errors?p_message=htp.p(dbms_aw.interp(%27sleep%2010%27))

This will cause the arbitrary SQL, in this case “htp.p(dbms_aw.interp(‘sleep 10’))”, to be added to the ICX_TEXT table. The browser is redirected to

https://example.com/pls/ebs/hr_util_disp_web.display_fatal_error_form?p_message=8595383

The attacker then takes the text_id number - in this case: 8595383 - and passes it to the DEXL procedure:

https://example.com/pls/ebs/hr_util_disp_web.dexl?p_url=8595383

and the SQL executes - in this case it simply hangs the application by calling dbms_aw.sleep for 10 seconds.

There are other ways to get “encrypted” data into ICX_CALL. For example, the ICX_ADMIN_SIG.TOOLBAR procedure:

https://example.com/pls/EBS/icx_admin_sig.toolbar?DISP_EXPORT=FOOBAR

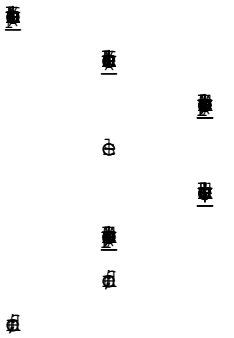
```
<TD>
<TABLE  border=0 cellspacing=0 cellpadding=0>
<TR ALIGN="CENTER">
<FORM ACTION="OracleON.csv" METHOD="POST" NAME="exportON">
<INPUT TYPE="hidden" NAME="S" VALUE="8619644">
</FORM>
<TD><A HREF="javascript:document.exportON.submit()" onMouseOver="w
<TD WIDTH=50></TD>
<TD><A HREF="javascript:help_window()" onMouseOver="window.status=
</TR>
<TR ALIGN="CENTER" VALIGN="TOP">
<TD ALIGN="CENTER"><A HREF="javascript:document.exportON.submit()"
```

8619644 is the TEXT_ID returned after inserting “FOOBAR” into APPS.ICX_TEXT via the ICX_CALL.ENCRYPT2() function.

Exploiting the SQL Injection flaw in JTF_BISUTILITY_PUB

Exploiting the SQL injection flaw in JTF_BISUTILITY_PUB is similar to exploiting the SQL injection flaw in HR_UTIL_DISP_WEB. The LOV_VALUES procedure of the JTF_BISUTILITY_PUB PL/SQL package takes a number of parameters, one being

P_WHERE_CLAUSE. Already its name should be ringing alarm bells. Here is the parameter's treatment in the procedure:



We can see that the P_WHERE_CLAUSE parameter is first "decrypted" using a call to the ICX_CALL.DECRYPT2() function. The DECRYPT2 function takes a number as an argument and this number is checked against the TEXT_ID column of the APPS.ICX_TEXT table and the associated TEXT column is returned. So, in the case of the P_WHERE_CLAUSE parameter of the LOV_VALUES procedure, we pass it a number and the corresponding SQL in the APPS.ICX_TEXT table is used as a where clause. As in exploiting HR_UTIL_DISP_WEB, if we, as an attacker, can get our own SQL into that table remotely then we can execute arbitrary SQL via the where clause.

As it happens the LOV procedure on JTF_BISUTILITY_PUB will do this for us. If we look at the code we see the following:

Here, if we pass NULL for the P_WHERE_STRING parameter but pass a value for the P_JS_WHERE_CLAUSE parameter then it is "encrypted" using the ICX_CALL.ENCRYPT2 function. The resulting TEXT_ID is returned and sent back to the user:

In the screenshot above we can see we've set our P_JS_WHERE_CLAUSE parameter to JTF_BISUTILITY_PUB.LOV as "length(dbms_aw.interp('sleep 10')) is not null". This SQL snippet, which will cause the application to sleep for 10 seconds and thus prove we're executing SQL, is added to the APPS.ICX_TEXT table and the corresponding TEXT_ID returned; in this case "8619584".

With the attack primed, we simply pass this number as the value for the P_WHERE_CLAUSE parameter to the LOV_VALUES procedure:

https://example.com/pls/EBS/jtf_bisutility_pub.lov_values?p_form_name=xxx&p_x=1&p_LOV_foreign_key_name=BIS_PRODUCT_CATEGORY_FK1&p_LOV_region_id=191&p_LOV_region=BIS_PRODUCT_CATEGORY_LOV&p_attribute_app_id=191&p_attribute_code=P_ITEM&p_region_app_id=191&p_region_code=BIS_PRODUCT_CATEGORY&p_where_clause=8619584

The application hangs for 10 seconds.

Additionally, the JTF_BIS_UTIL PL/SQL package is a wrapper for JTF_BISUTILITY_PUB so in blocking access to JTF_BISUTILITY_PUB you must also block access to JTF_BIS_UTIL. Lastly, the same vulnerability exists in ICX_UTIL.LOV.

Reference

[1] http://www.scribd.com/doc/284414206/EBS-Suite-Security
[2] http://www.oracle.com/technetwork/topics/security/cpujan2016-2367955.html