A4 Broken Access Control Lab

Sometimes attackers will try to use a value like an ID where they shouldn't. If our site responds, we can expose data. This is called an insecure direct object reference. Let's attack and harden against them.

Mounting the attack

1. Open the website. Make sure you're logged out.
2. Register a new user called "OtherGuy". Set all of his account info like company, address, title and so forth.
3. Log out of OtherGuy's account.
4. Log back in as yourself.
5. Go to your ViewAccountInfo.aspx page. Click on the link to change your info.
6. Did you notice how the QueryString in the address bar has your account name? Change it to read "OtherGuy".

You're logged in as yourself but you're now seeing OtherGuy's address and other account information. You can not only see it, but worse, you can change it! This is an insecure direct object reference. Let's fix it now.

Hardening the site

7. Stop the site running.
8. Edit ChangeAccountInfo.aspx.cs. Notice how in the page load event we are pulling the id from the query string. That has to go away.
9. Remove those lines and only read the user's name from Session like so:

```
_userName = (Session["UserId"] ?? string.Empty).ToString();
```

10. You'll also have to go back to the ViewAccountInfo.aspx page and as soon as the page identifies the user, have it save the username to session as "UserId".

```
Session["UserId"] = customer.ContactName;
```

11. Run it and attack the site again. Now, no matter what is in the QueryString, you can only change your own user info.

When you can no longer change the querystring to access someone's info, you can move on.

Hardening in .Net

Our solution works and works well, but .Net provides an easier way. With .Net, you can read the user's identity from the User object.

12. Comment out that line you added in the last section; the one where you read username from session.
13. Change it to pull the username from the built-in security model. You'll find the username in User.Identity.Name.
14. Run and test the site as yourself legitimately. Then try to get to OtherGuy's information as yourself.

When you can legitimately change your data and still not get to anyone else's information you can move on.

MIssing Function Level Access Control

Let's say that as one of our website administrators you want to maintain a product; raise the price, change the description, discontinue it … whatever. Here's how that is done.

15. Go to the website. Look around. Do you see anywhere that you can maintain products? No? Good. You should only see them if you are logged on as an administrator.
16. Go to http://localhost:7777/admin. Can you get there? ____ Again, permissions should prevent that.
17. Attempt #3: Log on as an administrator. Use 'jeffortson' as the username and 'cahiking' as the password. Now look for the admin link. Click it. Ah! Success!
18. Now that you can get to the protected admin page, try to edit a product. Make any change you want.
19. Go back to that product and make sure your changes stuck.
20. Finally, log out so you're a regular user, not an administrator. Are you out? ____ Are you sure? _____ Good. Moving on …

Mounting the attack

21. Go to the website and click on search. Search for any product. Once found, click on its link.
22. Notice the product page's URL. Write it here:
    _____

This is an SEO-friendly or RESTful url. URLs like this are written to be predictable and logical. Fortunately (or unfortunately) that makes them changeable and hackable. Let's do that now.

23. If you were a hacker and saw that URL to view a product, you might try to insert "Edit" or "Modify" or "Update" into the url. Add those in in varying ways until you hit on a URL that works. What is the URL that worked? _____
    (Hint: if you try and try and still can't figure it out, ask your instructor to help.)

Notice that this URL can be used by anyone. We thought that anyone using this page would surely go through the admin page which is protected through authentication. We thought that since we have its access behind the admin page only, nobody would know that it even existed. We neglected to anticipate that an attacker might navigate directly to it.

Hardening the site

24. In the main site, open web.config.
25. Find the </system.web> ending tag. Add this after it:

```
<location path="Product/Edit">
  <system.web>
    <authorization>
      <allow roles="admin" />
      <deny users="*" />
    </authorization>
  </system.web>
</location>
```

26. Try running the site again. Navigate to the page. Can you get there as a regular user? You should be redirected to the login page.
27. Now log in as an administrator. After logging in, you'll be able to edit products again.

Another hole

That URL is actually a route which was set up in global.asax. Feel free to look in there to see how it was set up. Unfortunately that route allows access directly.

28. Go to http://localhost:7777/Admin/EditProduct.aspx?ID=12 (or some other product). Can you get there? _____ Let's fix that.
29. Take a look at the site in Solution Explorer.

See how all of the admin pages are in a subdirectory? This is a good practice. We can put all pages that only administrators can access in this directory and protect the directory as a whole.

30. First, protect the directory by adding a section to web.config. Add another <location path="admin"> section like you did above. Once again, protect it from unauthorized access but this time make sure the user is in the role of "admin".
31. It's now allowing only logged-in users in the admin role to access it. But anonymous users can still get to it. Fix that by putting a <deny user="*" /> tag after the <allow role="admin" />.

Testing

32. Run and test your site as an anonymous user. You should not be able to edit any products regardless of how you get there.
33. Run and test your site as an administrative user. You should be able to edit any products again regardless of how you get there.

When you've corrected the direct URL access problems you can be finished.