All this text wasn’t written by me (Tiago a.k.a TigaxMT).

All of it was transcribed by me from HackerOne video lessons.

All credits and thanks go to them.

XSS and Authorization

XSS REVIEW

All of you have seen XSS in action at this point, but let's review the types of XSS that we're going to discuss today:

- **Reflected XSS** – Input from a user is directly returned to the browser, permitting injection of arbitrary content
- **Stored XSS** – Input from a user is stored on the server (often in a database) and returned later without proper escaping
- **DOM XSS –** Input from a user is inserted into the page's DOM without proper handling, enabling insertion of arbitrary nodes

RECOGNITION

The first step to exploiting or mitigating XSS is simple: find XSS.
But as you guys saw in your coursework, that's not always easy.
So what can we do to figure out if, say, a reflected input is vulnerable?

I follow a pretty straightforward mental checklist for each input:

1. Figure out where it goes: Does it get embedded in a tag attribute? Does it get embedded into a string in a script tag?
2. Figure out any special handling: Do URLs get turned into links, like posts in level 1?
3. Figure out how special characters are handled: A good way is to input something like **'<>:;"**

From those 3 steps, you'll know whether or not a given input is vulnerable to XSS.
At this point , one of the differences between stored and reflected XSS becomes apparent: rXSS vulnerabilities are inherently dependent on CSRF vulnerabilities to be exploitable, in the case of POSTs. If your rXSS exists just in a GET, you're fine, but you're dependent on CSRF otherwise.

EXPLOITATION CASE 1

During the special character test, you notice that angle brackets are passed through without encoding, and your input is being shown in a text node of the document.
In this case, a simple payload like <script>alert(1);</script> will almost definitely work.
In very rare cases, a WAF (Web Application Firewall) or other filtering may detect the script tag and prevent execution.

EXPLOITATION CASE 2

A closely related variant of the first case is when your input is being reflected in a tag attribute.
In this case, your first priority is to break out of the attribute, but in most cases you don't need to leave the tag at all – meaning no need for angle brackets.

Now how do we execute code? Well, there are a multitude of DOM events that can be triggered.
A good one in this case is onmouseover, e.g. http://"onmouseover="alert(1);
Giving you <a href="http://"onmouseover="alter(1);"> …
And now when the victim hovers over that link, you have JavaScript executing.

At that point , you're out of the attribute! But you can't leave the tag and spaces would end the "URL", so what can we do?
One important thing to realize is that there's no need for whitespace after the ending quote of an attribute. The tag <a href="http://foo.com"style="..."> is perfectly valid.

EXPLOITATION CASE 3

If you see your input being reflected in a script tag, there are a number of ways in which this can go wrong. Let's use the following example:
<script>var token = 'user input here';</script>
Normal HTML encoding does not properly mitigate this case, for two reasons:

1. HTML entities won't be parsed on JavaScript, meaning the input will simply be wrong.
2. Single quotes are rarely encoded as HTML entities.

With our example in mind, let's took at what happens with certain payloads under HTML encoding and simple string escaping.

- HTML encoded payload: ';alert(1);'
  - Gives us a final script of: <script>var token = '';alert(1);'';</script>
  - Meaning we have complete control over execution!
- JS string-escaped payload: </script><script>alert(1);</script>
  - Gives us a final script of: <script>var token='</script><script>alert(1);</script>';</script>
  - Again giving us complete control.

MITIGATION

We've now seen 3 or 4 different cases of stored/reflected XSS and how we can exploit them.
So how do we mitigate them? Well, that's a bit more complex than it seems.
People generally say "just escape/encode!", but don't recognize that context matters, as we've seen.

In the third case, it’s enough to string escape angle brackets in addition to quotes and backslashes. I.E. replace < with \x3c and > with \x3e.

But there are a multitude of cases where that’s not enough, e.g. when you’re passing an integer value into a DOM event attribute or a variable in a script tag.

MITIGATION THROUGH DESIGN

Unless there is absolutely no other option, user-controlled input should not end up in a script tag or inside of an attribute for a DOM event. While it is possible to mitigate it (as we just discussed), the likelihood of proper mitigation is next to nil.
You’re going to see a multitude of different ways in which XSS is mitigated. In almost every JS-related case, it’s going to be wrong.

DOM-BASED XSS

What is it?

DOM-based XSS (DOM XSS) differs from rXSS/sXSS in that it doesn’t depend on a server-side flaw to get attacker input into a page.
This means that trough vulnerable JavaScript on the client side, it’s possible for an attacker to inject arbitrary content.

Let’s take a look at a simple page that includes a flag in the page based on the locale specified in the hash, e.g. http://example.com/#en-us

```
<div class="flag">
    <script>
    var locale = 'en-us';
    if(location.hash.length > 2)
        locale = location.hash.substring(1);
    document.write('<img src="flags/' + locale + '.png"');
    </script>
</div>
```

Some of you might have noticed that it looks just like an rXSS vulnerability, just on the client side.
The string that comes from the hash in the URL is directly inserted into an image tag, allowing an attacker to pass anything, e.g. http://example.com/#”><script>alert(1);</script>

PROBLEM

The core problem with DOM XSS is that there are effectively an infinite number of ways in wich it can come about, each of which requiring different mitigations:

- Embedding attacker data into eval/setTimeout/setInterval requires string escaping/filtering
- Embedding attacker data into tags and attributes requires HTML encoding
- Same goes for innerHTML

MITIGATION

So given all that, how can we generalize about the mitigation?
Don't put user-controlled data on the page! It seems strict, but it's the way to go. Whitelist very specific things, e.g. a list of valid locales for the flag example.
If you must put user data into a page, you have to escape/encode for the specific context.

FORCED BROWSING / IMPROPER AUTHORIZATION

Well, they're pretty much both the same thing.

In both cases, you have a failure properly authorize access to a resource, e.g. an admin area is left unprotected, or you're able to directly enumerate values in a request to access other user's data.

WHAT'S THE DIFFERENCE?

The line is very fine between the two. Generally, forced browsing (or direct object reference) is used when you're talking about enumerable values such as post IDs and other parts of the site that are not ordinarily available to you from your privilege level.
Don't worry too much about which you use; some people combine them just under "authorization bugs" (or auth-z, to differentiate from auth-n, authentication).

A simple example of this can be found in level 1.
The permalink functionality for posts made it simple for an attacker to enumerate IDs and access every post in the system, not just their own.
For example, one of mine is: http://h101levels.appspot.com/levels/1/post?id=465
Changing id=465 to id=464 gives me a post from another user. This is an example of forced browsing.

PRIVILEGES

Aside from just changing IDs that we see, how else can we find auth-z bugs?
One of the best techniques when testing an application is to perform every action you can as the highest-privileged user, then switch to a lower-privileged user and replay those requests, changing session Ids/CSRF token as needed.

This is a great way to find admin-level functionality that as improper authorization checks. 99% of the time, applications don't generalize their access levels in any meaningful way, so make sure to test every entrypoint you see. You never know when one will be vulnerable.

REVIEW

- XSS
  - There are 3 kinds:
    - Reflected
    - Stored
    - DOM-based
  - rXSS and sXSS are very close in terms of exploitation and mitigation.
  - DOM-based XSS can’t be mitigated by any core strategy and exploitation differs greatly between cases.
- Forced browsing and improper authorization
  - Basically the same thing
  - Differences are how the bugs are found/come about

XSS CHEAT SHEET

Here are a couple quick things to try when testing for XSS:

- “><h1>test</h1>
- ‘+alert(1)+’
- “onmouseover=”alert(1)
- http://”onmouseover=”alert(1)