Introduction to Software Security

Chapter 3.9.4: Web Session Management

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

DRAFT — Revision 0.4, October 2023.

Objectives

- Motivate the need for session web sessions.
- Learn about the three basic components of a secure web session: secure connections, hard-to-guess session IDs, and nonces. These are often called “tokens”.
- Understand how these three components create a secure session.

Background

In the previous chapter, we have seen how an attack might cause your web browser to execute web requests that you did not intend, resulting in a cross site request forgery attack. This is one example of an insecure web session. In addition, when web sessions are not properly secured, attackers can try a variety of different attacks, including:

- Snoop on an unsecure (unencrypted) web session, obtaining passwords and other credentials to enable unauthorized access to the web site.
- Snoop on an unsecure connection to obtain private user data.
- Use information from the snooping to generate malicious web requests.
- Conduct unauthorized operations on this website.

To prevent such malicious behaviors, we can construct secure web sessions. Three key elements to such a session are:

1. Secure connections: Connections should be based on authenticating the server and using strong encryption for the web session.
2. Effective session IDs: Each session with a web application should be identified by a hard-to-guess session ID.
3. Use of nonces on each web request: This is another hard-to-guess token that is unique to each request and not stored in a cookie.

Note that if you use a web framework , it will often handle many of these issues for you. However, it is good to understand the fundamentals of how these frameworks should work. We now explain each of the three elements in more detail.

Secure Connections

This element is simple: always encrypt web sessions. In other words, always use HTTPS and never use HTTP. Never.

We see this advice coming from many authoritative sources, such as `cio.gov`, the resource website for U.S. government agency Chief Information Officers[1]:

> *All browser activity should be considered private and sensitive.*

White House OMB (Office of Management and Budget) memorandum M-153-13 from June 2015 requires that U.S. government servers all adhere to an HTTPS-only standard. This advice is 100% correct and provides an example that we should all follow.

Apple promotes a similar direction in their App Transport Security (ATS) policy. Since iOS 9.0 (September 2015), iOS enforces strong HTTPS connections based on TLS 1.2 or later and requires a specific exemption if you want to use HTTP for some purpose.

Android also tries to encourage the use of HTTPS but is somewhat more permissive about the use of HTTP. When you create an Android App Link[2], the default is to use HTTPS. You can only override that default if you provide explicit permission in the application manifest:

```
<application android:usesCleartextTraffic="true"></application>
```

When the user installs the application on their device, they will be asked if use of HTTP is acceptable. Of course, if they say "no", then the application will not be loaded, so users tend to just ignore the implication of such messages and say "yes". Otherwise, they may not be able to complete their current task at hand. In general, asking users for such permissions is not an effective form of security as they often do not understand the issue any further than "If I don't say yes, I won't get my work done."

The most effective approach to connection security is simply to disable HTTP in your web server. This is a simple configuration option in most servers and frameworks. We are long past the need to save power and time by avoiding encryption. Processors and networks are now fast enough, except in very specialized embedded devices.

Note that you need to be sure that all the subdomains in your network domain use HTTPS. For example, if your bank's domain, `bank.com` is secure, it might not be the case that the separate web server at `creditreport.bank.com` is also secure. This server might have to be separately configured in the same way as the bank.com server.

You can also configure your web server to use HTTP Strict Transport Security (HSTS)[3]. This option will add the `Strict-Transport-Security` header to your server responses. HSTS informs browsers that

[1] `https://https.cio.gov/`
[2] `https://developer.android.com/training/app-links`
[3] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security`

the site should only be accessed using HTTPS, and that any future attempts to access it using HTTP should automatically be converted to HTTPS.

Google maintains an HSTS Preload List web application[4] where you can register your website as an HTTPS-only site. All the major browsers, including Chrome, Firefox, Opera, Edge, and IE11 subscribe to this list and will enforce this limitation. It is a relatively simple task to request that your domain name be added to the HSTS Preload List.

There are other mechanisms that you can use to prevent information from being disclosed over an insecure connection. For example, in your web responses, you can add two attributes to the HTTP headers, `Secure` and `HttpOnly`:

1. `Secure`: prevents cookies sent with this option from being sent to the server over insecure (HTTP) connections.
2. `HttpOnly`: prevents access to the cookies by Javascript running in the browser. This means that Javascript cannot reference `document.cookie`, as we saw in the cross site scripting chapter (Chapter 3.9.2).

Session IDs

A session ID is used to label a series of web requests belonging to the same session. For example, each time that you authenticate (log in) to a web service, you would be issued a new session ID. In addition, for many web sites, when you start anonymous (not logged in) access to a server, you will also be issued a session ID.

Session IDs (and nonces, described later) are both often called "tokens" in various web frameworks.

Possession of a valid session ID serves as proof of right to access a particular service or resources. So the session ID is a form of capability based access control[5].

It is important that the session ID be difficult to guess so that imposters cannot create malicious web requests. While session IDs like session0001, session0002, session0003, …, are easy to generate, they are also easy to guess. The zoom teleconferencing service was vulnerable to a widely publicized attack based on easy-to-guess session IDs. This attack, called "zoom bombing"[6], allowed malicious users to join zoom sessions to which they were not invited. The result of this attack was loss of privacy and occasional (and sometimes offensive) disruption of the conference session.

Some common characteristics session IDs generated by web frameworks include:

1. Hard to guess: the essential characteristic of a session ID.
2. Digitally signed: an easy way to remove patterns and detect tampering. A signed session ID acts as a capability, so possession alone is evidence of permission to make requests. In large

[4] `https://hstspreload.org/`
[5] `https://en.wikipedia.org/wiki/Capability-based_security`
[6] `https://nvd.nist.gov/vuln/detail/CVE-2022-22780`

distributed environments, this can reduce requests to the central server because the front-ends can check the authenticity of the request.

3. Include a timestamp: provides an easy way for the server to detect for the expiration of the session that might require re-authentication.

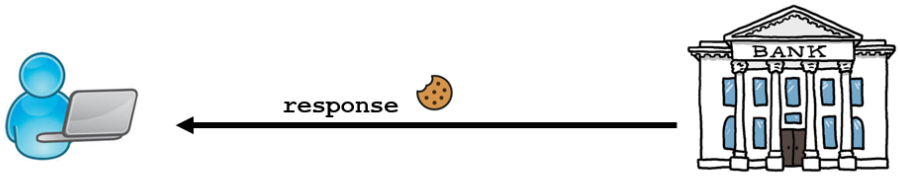
Some web frameworks will encrypt session IDs. This has the advantage that it automatically gives the first two benefits above but provides only limited additional security. In addition, encrypted session IDs are usually avoided in very large distributed environments because there may be front-end processing on a web request, such as routing the request to the appropriate server. The information needed for routing (such as the user ID) needs to be unencrypted. Since we are communicating over an encrypted connection (HTTPS), encrypting session IDs has less benefit.

Session IDs can be stored in cookies or browser storage (localStorage or sessionStorage). Cookies are vulnerable to cross site request forgery attacks. Both cookies and browser storage can be exfiltrated using a cross site scripting attack, though that vulnerability for cookies can be reduced by use of the Secure and HttpOnly options mentioned above. So, while cookies are a good place to store session IDs, we need one more element to make a session secure.

Nonces

The final element of our secure session consists of a web application creating a random value that is embedded in each response (web page) sent back to the client. This random value is called a *nonce* in mathematics and commonly called a *token* in web frameworks. When the client submits a web request based on this web page, the random value in that page is returned to the web application and is checked to see if it matches the value sent. Because the random value is contained in the web page and not in a cookie, it can be sent to the server if the client submits that page. As a result, a request generated by a cross site request forgery attack cannot contain a nonce sent from the application.

A common way to embed the nonce and have it returned to the application is by the use of a hidden initialized value in a web form. This kind of entry might look like:




```
<form>
 …
 <input type="hidden" name="token" value="a2vM9jT1w">
 …
</form>
```

The embedded value is stored in the `token` field of the form, where the web application provides an initial value for it. Of course, the "..."s in the form are the actual fields that will be submitted with this

web request and would be visible for the user to fill in. When the form is submitted, the nonce value is submitted with it and then checked on the server side.

Bringing It All Together

While there are many different variations on how session IDs and nonces are delivered and stored, in this chapter, we have outlined a few variations and showed our favorite version.

A secure connection, based on HTTPS, is an essential foundation for a secure session. Without it, all other measures will be ineffective.

The session ID stored in a cookie gives an easy-to-use mechanism, provided that the session ID string is chosen in a way that is difficult to guess. Adding the Secure and HttpOnly attributes makes this feature, without considering nonces, more resistant to XSS attacks.

Creating a nonce, delivering it in the message body, and returning it in a form, helps to defeat CSRF attacks and puts the nonce out of the reach of exfiltration by an XSS attack. Storing the nonce in browser localStorage or sessionStorage requires more care.

Web Frame Example: Session IDs and Nonces in Django

As we mentioned at the start of this chapter, if you are programming your application in a web framework, that framework will handle many of the issues related to implementing a secure web session. Here, we talk briefly about how Django[7] implements session IDs and nonces[8].

Note that a common encoding for session IDs and nonces is the JSON Web Token (JWT) based on IETF RFC 7519[9]. This encoding is popular, in part, because it can be easily incorporated into the URL.

In Django, when a client authenticates itself, a “Refresh Token” is returned in an HttpOnly cookie, effectively acting as a session ID. In addition, an “Access Token” is returned in the body of the response, acting as a none.

Summary

In this chapter we discussed the risks and consequences of not having a secure web session. We then described three critical elements needed to build a secure web session: (1) secure connections, (2) hard to guess session IDs, and (3) nonces stored out of the reach of an XSS attack. We then briefly described how the popular Django web framework implements these concepts.

Exercises

1. Choose a web framework and research the details of how it handles sessions. Evaluate how this framework tries to defeat XSS and CSRF attacks.

[7] `https://www.djangoproject.com/`
[8] `https://old-monk.medium.com/token-authentication-in-django-216de56d9d57`
[9] `https://datatracker.ietf.org/doc/html/rfc7519`

2. Check out the HSTS Preload List[10] website. Is your organization on this list? Is your bank? If you have a personal website, work to add it to the list (making sure that you really do only support HTTPS).
3. Try out the techniques presented in this chapter using the exercise based on our virtual machine image. Note that you will need the Virtual Box virtual machine system and an x86-based processor.

   `https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Exercises/3.9.4_Web_Attacks_Session_Management_Exercise.html`

[10] https://hstspreload.org/