Lock Picking PHP Sessions

Introduction (the skippable part)

If you are an experienced PHP developer, you know that in order to optimize your application, eventually you need to peek "behind the curtains". This gives you better understanding of what your PHP and/or your web server are actually doing, when you think that they are working hard on producing your web page. And you find yourself in Profiling, in Zend Server Code Tracing, looking at Z-Ray or even at the logs.

There is a whole universe of beautiful and exciting things you can find in these places. In this article we'll talk about one particular topic – PHP sessions. When all obvious threats to your application's performance have been eliminated and that spot of light at the end of the tunnel is starting to take shape, you'll suddenly notice that a very simple thing – session_start() – takes too much time.

Sometimes it does, sometimes it doesn't. To make things worse, you run this one particular script and it works just fine. But the minute you load your whole web page with nice pictures, JavaScript widgets, elaborate CSS designs, things go south.

The offender is well known, although rarely mentioned in an average PHP study guide. You read about this on php.net, but you never think this can happen to you. You see it every minute of every day, but you never really notice it. Until it hits you.

Suspense music reaches its climax…

Session Locking

Many developers don't realize or choose to ignore this behavior of sessions. Which is easy to do – PHP makes session handling convenient for developers. Unfortunately, you learn about $_SESSION and accompanying handful of functions very early in your PHP study. And you get used to how simple and transparent this magic is. You learn it so early, that even when you're a better developer, you rarely doubt this magic.

The ugly truth is that Santa is not real and there is no magic in sessions. Depending on the session handler, the session data are stored in a file, in memory, in some RDBMS / NoSQL storage etc. To identify this data, PHP sets a session cookie in user's browser. And this works perfectly fine as long as there is only one HTTP request per session.

If the user opens the same page in two browser windows or tabs, or if the page has framesets or AJAX powered widgets, there will be more than one (almost) simultaneous requests to the server. Naturally, each request transmits cookies, including the session cookie.

What happens if two requests access the same session simultaneously and the session data is modified by one of these requests? This would mean that the data read by the second request are not valid anymore. And what would happen, if both requests try to simultaneously modify the session data? This dilemma is not specific to PHP sessions and not new to computing in general. One of the proven solutions is locking.

This is how PHP session locking came to be. **Only one request can access a given session**. The session is locked for the rest of concurrent requests. Again, this is true for any session handler and not only in PHP. Why is it important to keep this in mind? Because today even a very simple web page can emit 3-4 simultaneous requests to your PHP application.

Let's take a look at an example. We'll take the worst case scenario where a session is opened by all simultaneous (and extremely slow) requests:

if request #1 takes **20 seconds** (user authentication)
and request #2 takes **30 seconds** (DB access)
and request #3 takes **25 seconds** (external web service),
request #4 can get access to this session only **75 seconds after it came in**.

blocked URL

What are the right things to do?

- Close the session as early as possible - explicitly with `session_write_close()`. Don't wait for the session to close automatically at the end of the request. This way your request may still take 30 seconds, but the session will only be locked for several milliseconds, allowing the next request to proceed.

  blocked URL
- As strange as it sounds - avoid using session_start() and $_SESSION. This will automatically lead to healthy parallel programming practices. For example, you can replace session_start() with session_get():

```
function session_get() {
        global $SessionData;
        session_start();
        session_write_close();
        $SessionData = $_SESSION;
}
```

Even if you have to write to the session at the end of the request, you should close it early and reopen at the end of the request. This doesn't make sense in linear paradigm, but in parallel programming it looks like this:

Your first request needs to go to the database and fetch the user's shopping cart. This is an expensive operation and will take about 30 seconds. At the end this request needs to save the timestamp to the session. You also have 6 small widgets on the site, which don't depend on expensive operations but depend on session data, for example, they need to say things like

Hello **Sven**, what a beautiful ***søndag*** today!

You won't be able to display these widgets until you close the session in the first request. If you close the session at the beginning and open again at the end of the first request, you will entertain your user with widgets while he's waiting for the shopping cart to load.

For example, you can introduce session_set():

```
function session_set() {
        global $SessionData;
        session_start();
        $_SESSION = $SessionData;
        session_write_close();
}
```

- Don't write to the session too often. In general, don't consider the session a variable that keeps information between requests. In many cases, it is cheaper to get the information again than to write it to the session.
- To the previous point, but also to performance in general - make sure you include caching in your architecture.

  For example, if you go to an external web service to fetch the weather, there is no reason to do this for each user from Berlin - do this once every 30 minutes, when the cached information expires.

  In the same way, maybe you really don't need to save anything in the session, except the user's ID - the rest you can save in cache or in the database. If you use Zend Server, Zend Data Cache would be a sound choice, anyway there is no lack of caching solutions for PHP.

Disclaimer

Please do not take the recommendations and especially the code samples in this article literally. Although the theory behind this article is correct, real life is usually different from theory and we often need to settle for "good practices" instead of implementing "best practices".