PHP Session and Password Security

Elliott Post

Loyola University Chicago

Author Note

Thank you to Dr. Paul Rose at SIUE for providing this APA template. To aid in his request for tracking template usage, I have left in the below note for anyone else who stumbles across this template:

…if you find [this template] helpful, I don't want your money, but I would be grateful if you would click on this URL: http://goo.gl/DGHoZ. It directs to a harmless Department of Psychology web page at SIUE, but what is more important is that it records click-through data that give me an idea of how many people have found this document helpful. Thanks!

Abstract

Any website that has the ability to “remember” or “recognize” a visitor uses some form of *session* management. These sessions, which have a one-to-one correspondence with a unique site visitor, often contain sensitive information including a unique session ID. Because these sessions contain sensitive data and are used to identify a specific user the sessions become a target for attackers. Most server-side languages such as PHP have some form of session management programmed into its base functionality or an official library. Unfortunately, the default session management is often insecure on its own and there are a series of steps which must be followed to protect against session hijacking, session fixation, and session predicting. Additionally, PHP recently revised the way it handles password management and this change is related to session management. This paper discusses how to improve session security and utilize the new password management features.

Keywords: PHP, Session Management, Session Hijacking, Session Fixation, Password Hashing

PHP Session and Password Security

The programming language PHP (Hypertext Pre-Processor) powers approximately 81% of all websites where the server-side programming language is known (W3 Techs, 2015). The prevalence of PHP alone is cause enough to require PHP web developers to learn about securely accepting, storing, and manipulating passwords and user sessions. On its own, PHP provides the base functionality for managing sessions and working with passwords; however, these prebuilt functions work only to provide capabilities, by themselves alone they are insecure. PHP developers *must* extend the default functionality and security that PHP provides for the applications they build if the application handles any session or user data.

According to the PHP manual, "[s]ession support in PHP consists of a way to preserve certain data across subsequent accesses" ("Sessions – Manual"). I offer a more specific variation of this definition, one that this paper follows: A session in PHP is a server-managed, trackable set of specific user data (collected with or without the user's consent) that extends over a specific period of time, such that the developer determines both what data is collected and what the lifespan of a session is. On a non-application-specific level, the session data collected ranges from usernames and passwords, to IP address and browser data, to settings and other user variables, and more. Application-specific data collected in sessions may include shopping cart data (to be later stored in a database), pages visited, game character and level information, etc.

Sessions matter because they are critical with respect to how the web is used. Sessions allow users to be uniquely identified by the server. Sessions allow servers to "remember" users such that they do not need to provide login credentials anytime a secure resource is being accessed. And, most importantly, sessions matter because they *protect* user data when used correctly. If you have ever used Facebook, Twitter, Google, or any other dynamic website you

have most likely encountered sessions. You may not have been aware that your activity was being recorded and tracked through sessions because sessions are designed to be transparent. Any website that can uniquely identify a user or remembers a user between pages or visits uses *some* form of session management (although these sessions may not necessarily be server-side but because server-side is most common this paper analyzes only PHP sessions).

In order to discuss security vulnerabilities and procedures with sessions, one must have a general understanding of how sessions work. Sessions are saved both on the server and often on the client's machine as well. (I will discuss more the difference between storing session data purely on the webserver versus relying on the user to supply session data when discussing vulnerabilities.). Session data is serialized and saved into a file on the server. Each session has a randomly assigned identifier (unless otherwise specified) known as the *session ID*. The filename is generated by the session ID when the session is created or regenerated. Session files are read when *session_start()* is called and the *$_SESSION* global variable is populated with session data from the session file. As previously mentioned, this data typically includes sensitive user information such as username, password, and other user-identifying features such as IP address and browser data. When *session_start()* is called, the session data stored on the client's computer—usually in the form of a cookie—is read by PHP native behavior and the session ID is used to locate the specific aforementioned session file. In the case that the client does not have session data stored locally, the session ID will be provided through different means, most commonly as a *$_GET* variable (HTTP query variable) in the URI. Sessions are able to "remember" users, then, because the data is saved to a flat file and can be quickly retrieved for re-authentication and re-authorization.

Because sessions are responsible for both uniquely identifying a user and for storing sensitive user data, they make tempting targets for malicious users. There are two common approaches to breaking sessions, *session fixation* and *session hijacking,* and one less common approach called *session predicting*. (Kolšek, 2002; PHP Security Consortium, 2005).

I mentioned earlier that the session ID is a randomly assigned ID. *Session predicting* (or *session prediction*) is the process of guessing the ID that PHP will randomly assign the session. This approach is the least effective of the three mentioned because PHP's random ID generator is actually quite secure in itself, and also because the other two methods are far more effective at corrupting a session. The other methods exemplify exactly what can happen if a session ID becomes insecure.

Session fixation is similar to phishing and it is the process of a malicious user specifying a (fixed) session ID such that when *session_start()* is called by PHP, a session ID is found (as it was specified), and then a session file is created by PHP that corresponds to the session ID. Of course, this session is still invalid because it represents only an unauthorized session (I assume, then, that all sessions created are marked as "insecure" automatically by the developer's application until a user has been both authenticated and authorized[1]. In order for session fixation to actually be useful, it requires a legitimate application user to use the fixed session ID created by the malicious user. A malicious user can trick the legitimate user into using the fixed session ID through a variety of means.

The most common approaches for malicious users to trick legitimate users are: 1) cross-site JavaScript cookie injections (though most modern browsers refuse cross-site JavaScript interactions now for reasons like this and other security concerns), 2) providing the session ID through the URI, and 3) using HTML's hidden form fields through an illegitimate login-like

page. In the first case, the malicious user will write some JavaScript that sets a cookie for a foreign domain (the target site) that specifies the field *sessionid* with the fixed session ID from earlier. Then, the malicious user just needs to wait for the legitimate user to connect to the target site, provide credentials, and the malicious user has access. Again, most modern browsers refuse cross-site scripting through JavaScript as there are many related security concerns surrounding it. In the second case, PHP by default can accept a query variable *sessionid* in place of a user cookie and then all the malicious user needs to do is send the legitimate user to a link such as "http://mysite.org/login.php?sessionid=555" where 555 is the fixed session ID. The malicious user now has authorized access once the legitimate user logs in. Furthermore, this approach can be masked by using URI shorteners such as http://bit.ly. In the third case—and this is the most like phishing—the malicious user creates a copy of the target site's login page, but adds a hidden field called "sessionid" with a value corresponding to the fixed session ID. The form does submit eventually to the actual target site's login page, but with problems. The problems are two-fold, the malicious user now has the user's login credentials, IP address, browser information, etc., but they also have a valid session and a notification from the legitimate user that the fixed session is now authorized. The first two cases *might* not have a method to send a notification back to the malicious user.

The other, and generally most-common approach to breaking sessions, is *session hijacking*. Session hijacking is the process of a malicious user becoming aware of an existing session ID (versus explicitly defining the session ID like in session fixation) and then using the known, authorized session ID to impersonate a user (Teja, 2015).

There are multiple ways that a malicious user can learn of a valid and authorized session ID. The most popular ways are: 1) Network Eavesdropping, 2) Unwitting Exposure, 3)

Forwarding, Proxies and Phishing, and 4) Reverse Proxies. All of these topics are worthy of their own papers and as such I overgeneralize their descriptions in the interest of being concise. In the first case, the malicious user listens in on all communications to a certain network and monitors incoming and outgoing data transmissions. Because the session ID will either be passed in the HTTP request as a cookie or as a query variable, the attacker needs only to look at these two places in any HTTP requests and can quickly identify the authorized session ID. In the second case, a server which relies on the session ID to be passed through URIs will usually automatically append the session ID as a query variable to all internal hyperlinks (this is called transparent session IDs). The user needs only to share a link and unwittingly shares his session ID. The third case is very similar to what is discussed for the final case in session fixation. The fourth case is an extension of the first case, where an attacker listening in on a network not only listens but modifies network data before allowing it to pass on to its intended destination.

It is relieving to know, however, that there exists ways to protect sessions from the aforementioned vulnerabilities. There are additionally other improvements a server administrator can make to PHP's global settings to further improve security[2]. These other improvements all revolve around modifying default PHP session settings such as session file prefixes, session file save locations, session lifetimes, etc.

The very first thing a PHP developer utilizing PHP's native sessions should do is understand and utilize *session_regenerate_id()*. This function will copy the existing session data to a new session with a new, randomly generated, session ID. It is important also to note that this function accepts an optional Boolean parameter specifying whether the existing session before regeneration (herein the "old" session) should be destroyed after it has been regenerated. The problem with this parameter is that if one opts against utilizing it, it defaults to false, meaning the

old session is still valid and authorized, but if one opts to delete the old session, then pending processes which use that session ID may fail. Hafner with Team Treehouse suggests an excellent method of destroying old sessions via AJAX after a very short period (~10 seconds) to allow for database queries and other pending processes which are utilizing the old session ID to finish before invalidating/deleting the old session (2009). The PHP Security Consortium suggests regenerating the ID each time "there is change in privilege level" (2005). The effect of regenerating the session ID (and deleting the old session afterward) is that all existing sessions which have been compromised through any of the listed vulnerabilities and various cases are no longer compromised, and with the exception of phishing, all session fixation is blocked. This is a huge improvement, but it is still insufficient. Attackers can still use session hijacking techniques (or, possibly, session predicting but again this is *very* unlikely to begin with) to corrupt a regenerated session.

In addition to using *session_regenerate_id()*, PHP developer's should utilize other known user data. For instance, if the HTTP request provides a value for *user-agent*, record that value in the session. Because sessions *should* be limited to one browser, if the user-agent changes mid-session, then the change in browser should be seen as a security concern and policies and procedures should be in place to handle this change. (E.g.: if a user logs in with Chrome then visits their home page, and suddenly Firefox appears as the user-agent with the same session ID and is attempting to change profile password, the application may wish to destroy or pause that session until identity can be confirmed.) A simple way to do this with PHP is to save the user-agent to the session when the session is created, and during any sub-sequent access request to that session the user-agent should be verified. However, because the attacker may be using the same user-agent as the legitimate user, or because HTTP 1.1 does not require user-agent to be

supplied, this technique is insufficient in itself. Chris Shiflett recommends to concatenate the user-agent with a string known only to the application and generate a hash from it, then to save this hash to the session and compare, rather than simply comparing the user-agent only (2004). Along with user-agent, the user's IP address should be saved and verified subsequently (though IPs may change over time and many IPs are shared in a network so one should not rely solely on the IP).

Now, applications that utilize sessions which regenerate IDs, and match more than a single component (session ID) to validate a session, are secured from session predicting and session fixation entirely, and can also *detect* forms of session hijacking and phishing attempts. Again, policies and procedures should exist for how to handle these detections, as the tactics listed so far may not prevent session hijacking entirely. With the exception of phishing and unwitting exposure, an application can further thwart session hijacking by utilizing SSL. Sites which are secured using SSL will encrypt network traffic and prevent network sniffing and modification. The method to prevent unwitting exposure in session hijacking is especially simple: require site users to use cookies, and disable PHP's ability to use sessions through query variables (URIs). To complete this action, adjust the applications PHP settings such that *session.use_only_cookies* has a value of 1, and *session.use_trans_sid* has a value of 0. Once a developer has made all these changes, the sessions on the site are secured from all listed forms of attacks that do not utilize phishing.

Phishing is a very dangerous form of attack and it is also very difficult to prevent because the attacks occur off the application server[3]. Usually an attacker will build some sort of application which resembles or pretends to be the legitimate application and a legitimate application user will unknowingly fall prey to the attacker by providing login credentials or other

sensitive data[2]. Although phishing attempts cannot be completely avoided, mitigating damage from phishing attempts is the responsibility of the application developer. The PHP developer can limit compromised data by combining sessions with databases.

Sessions can hold a great deal of data because they can are simply flat files; however, just because they can does not mean they should. Sessions should be tracked *and* managed in the application's database system. By tracking sessions and unique identifiers such as user-agent, IP address, etc., a wise developer can build a system that detects unusual patterns and notifies the legitimate user of a potential account compromise, as well as disables the threatened account until identity can be verified. Again, this is a process that a set of policies and procedures (as well as a pattern-recognition system) should be constructed to determine how to handle recognized potential attacks. In addition to tracking sessions and detecting potential threats, combining sessions with database management allows the application developer (and potentially the application user) to view all active sessions, view session history, and remotely terminate sessions, instead of waiting for the sessions to expire or to manually erase a session flat file. In short, a session ID should be verified using the aforementioned processes, and in addition it should be compared to a table in the database that tracks sessions to further verify the authenticity of that session.

Furthermore, a developer can utilize third party software which has been vouched for by the open-source community as "secure" to mitigate the need to develop her own session system. In many cases, this is a good option; however, it is important to understand *why* and *how* the existing software secures sessions and user data before implementing this software. Many popular frameworks such as PHP's Laravel use security approaches similar to what is listed

above to help secure sessions (Otwell, “Session – Laravel – The PHP Framework For Web Artisans”).

Password management is just as critical as session management to the security of an application. The best way to secure passwords is with modern hashing algorithms. If the application fails to implement good practices with managing user passwords, the security of the sessions may not matter at all. Applications which do not use hashes, or rely on hashing algorithms like *md5* or *sha1*, and even *sha512* for hashing user passwords are vulnerable to attack (Ferrara, 2012a).

In the past, functions like md5 and sha1 were commonly used for hashing user passwords. Before discussing modern approaches to hashing, allow me to provide much generalized terminology for background on the subject. It has been my experience that many developers do not understand the difference between encoding, encryption, and hashing (Post, 2014). Encoding is the simplest to understand. In programming, encoding typically defines *how* something should be rendered or displayed. The most simple (and perhaps most common) example of this is character encoding such as Latin, UTF-8, UTF-16, etc. Encryption is commonly mistaken for encoding, but encryption is *not* encoding. Encryption modifies the text (often called mangling) such that it no longer resembles the original message (or at least, a *good* encryption algorithm will do this). A good example of encryption is the Enigma machines used by the Nazis in WWII. Most encryption algorithms are reversible, but an algorithm does not require that the text can be decrypted. Lastly, then, hashing is based on encryption. Hashing is a *digest* of the encrypted text. Hashes have a one-to-many correspondence with plain text strings. Hashes generally are irreversible, and have a feature such that if a single bit changes in the encrypted text, a completely different hash is generated. By concatenating the plain text before

encryption with a randomly generated or static string (called a *salt*) hashes become exceedingly difficult to crack.

I said earlier that md5 and sha1 are vulnerable to attack, but I also stated hashes are difficult to crack. Generally speaking, hashes are difficult to crack, but certain algorithms have known security vulnerabilities and this has partially led to the creation of *rainbow tables* ("How Rainbow Tables work"). On a low level, rainbow tables are lookup tables that map a hash to a set of plain-text strings. In other words, within a matter of seconds, anyone with access to a rainbow table can search the table for the hash and determine a subset of all strings that generate that specific hash. The reason that md5 and sha1 should not be used for security, then, is because both algorithms have widely published lookup tables and lookup time on a modern computer is almost instant. There are other vulnerable algorithms, but the secure, standard encryption algorithm now is usually Blowfish Crypt (Bcrypt), though other secure algorithms do exist and can be used (Ferrara, 2012b).

Of course, if the application does not use any hashing and stores passwords in plain text in a database, then there are fatal security flaws. Additionally, applications where the developer has created her own encryption algorithm are often vulnerable because the developer is commonly not a security expert. If an attacker gains entry to the database where passwords are not hashed or not hashed securely, then the attacker can deduce plain text values for each user password. By the time this occurs, session security will be completely irrelevant as the entire security of the application has been compromised.

Luckily for PHP developers, using modern hashing algorithms has never been easier. In 2012, Anthony Ferrara wrote a plugin for PHP that utilizes Bcrypt and several other modern encryption algorithms to make it extremely simple for PHP developers to work with encryption.

PHP has supported Bcrypt and other algorithms since version 5, but until 5.3.7 there were several security issues which existed in PHP's implementation (Ferrara, 2012b). Although PHP had support, the encryption algorithms were very complex to use *correctly*. Ferrara's plugin was the result of an official PHP developers RFC (see Ferrara, 2012a) that included a very short list of functions to encapsulate the security and complexity of using the native PHP encryption implementation. This plugin is available for PHP 5.3.7-5.4.x and in PHP 5.5 the plugin became an official part of the PHP core (Ferrara, 2012b; "Password Hashing – Manual"). By using Ferrara's functions (*password_hash*, *password_verify*, *password_needs_rehash*, *password-_get_info*), a developer can greatly improve password management security in her application, and in the event of a database breach the developer can rest safely knowing the hashed passwords are extremely difficult to decrypt[4].

Coordinating strong password hashes with secure session management provides an application a great deal of security. Of course, adding features such as SSL, and mitigating other network and server threats to the server which runs the application are also very important to any application's security. Nonetheless, at minimum, a developer's job should include building strong application security. By expanding upon PHP's default session management, and utilizing its modern password management libraries, a developer has fulfilled her role of providing a minimal yet well designed session and password management system for her application.

References

Ferrara, A. (2012a). *ircmaxell/password_compat*. *GitHub*. Retrieved 14 June 2014, from https://github.com/ircmaxell/password_compat

Ferrara, A. (2012b). *PHP: rfc:password_hash*. *Wiki.php.net*. Retrieved 6 September 2015, from https://wiki.php.net/rfc/password_hash

Hafner, R. (2009). *How to Create Bulletproof Sessions - Treehouse Blog*. *Treehouse Blog*. Retrieved 3 September 2015, from http://blog.teamtreehouse.com/how-to-create-bulletproof-sessions

Kestas.kuliukas.com. *How Rainbow Tables work*. Retrieved 6 September 2015, from http://kestas.kuliukas.com/RainbowTables/

Kolšek, M. (2002). *Session Fixation Vulnerability in Web-based Applications*. Retrieved 3 September 2015, from http://www.acros.si/papers/session_fixation.pdf

Otwell, T. *Session - Laravel - The PHP Framework For Web Artisans*. *Laravel.com*. Retrieved 4 September 2015, from http://laravel.com/docs/4.2/session

Php.net. *PHP: Password Hashing - Manual*. Retrieved 5 September 2015, from http://php.net/manual/en/book.password.php

Php.net. (2015). *PHP: Sessions - Manual*. Retrieved 2 September 2015, from http://php.net/manual/en/features.sessions.php

Phpsec.org, (2005). *PHP Security Consortium: PHP Security Guide: Sessions*. Retrieved 2 September 2015, from http://phpsec.org/projects/guide/4.html

Post, E. (2014). *PHP Password Hashing API*. *Ellytronic.Media*. Retrieved 5 September 2015, from http://projects.ellytronic.media/homework/comp453/php-password/

Shiflett, C. (2004). *Session Hijacking, by Chris Shiflett*. *Shiflett.org*. Retrieved 4 September 2015, from http://shiflett.org/articles/session-hijacking

Teja, K. (2015). *Preventing Session Hijacking in PHP*. *Packetcode.com*. Retrieved 3 September 2015, from http://packetcode.com/article/preventing-session-hijacking-in-php

W3techs.com,. (2015). *Usage Statistics and Market Share of Server-side Programming Languages for Websites, September 2015*. Retrieved 2 September 2015, from http://w3techs.com/technologies/overview/programming_language/all

Footnotes

[1] Setting a Boolean flag in the session such as “authorized,” or “initiated” will achieve this goal. See an example (using the “initiated” flag) in the PHP Security Consortium reference just before the Session Hijacking heading: http://phpsec.org/projects/guide/4.html.

[2] In many shared hosting environments, developers can modify these settings at the account level by relying upon php.ini files or .htaccess files to modify PHP environmental variables.

[3] For more information about phishing and examples of it, see: http://www.pcworld.com/article/135293/article.html and http://www.microsoft.com/security/online-privacy/phishing-symptoms.aspx

[4] For examples and implementation details, see my blog article “PHP Password Hashing API.”