Global Scientific JOURNALS

GSJ: Volume 9, Issue 1, January 2021, Online: ISSN 2320-9186
www.globalscientificjournal.com

SECURING INTERACTIVE DATA-DRIVEN PHP WEB APPLICATIONS

Muhtar Hanif Alhassan

Department of Computer Science,
National Open University of Nigeria, Abuja
malhassan@noun.edu.ng

KeyWords
SQL Injection, Vulnerability, Threats, XSS attack, Denial-of-Service attack, DOM-based XSS

ABSTRACT

There is a rising tendency of moving away from desktop applications towards interactive web applications in seeking for robust and effective enterprise solutions. A key setback to this is the inherent security challenges that characterize the Web. This paper presents an overview of the potential risks of SQL injection and other threats faced by PHP Web applications and briefly describes simple coding techniques that help to avoid them. The work is based on experience in developing an interactive students' portal for a large university.

1. Introduction

Web applications have gained tremendous popularity over desktop software solutions for a number of reasons. Foremost is the shift in computing paradigm that increasingly favours computing in the Cloud over traditional solutions that rely on total ownership of ICT infrastructure. Web applications provide support for mobility and portability at substantially reduced costs in ICT investment. These key advantages of Web applications rely on the ubiquitous Internet technology with its inherent security challenges. In what follows we briefly present key strategies for designing and implementing secure Web applications with particular reference to PHP/MySQL development environment. It is assumed that the network and hosting environment for the Web application are fully secured and focus is on the application itself as the main channel of attack.

1. Basic Components of Application Security

An application's security status is normally determined by the degree to which it provides the following:

- **Authentication** which helps determine the identity of the application's users. Authentication uniquely identifies the clients of applications and services. These include end users, other services, processes, or computers. Technically, authenticated clients are referred to as principals.
- **Authorization** which addresses role assignment for each user. Authorization governs the resources and operations that each principal is permitted to access. Examples of resources are files, databases, tables, rows, and so on, together with system-level resources such as registry keys and configuration data. Operations include performing transactions such as purchasing a product, transferring money from one account to another, and so on.
- **Auditing and logging** which helps ensure non-repudiation.
- This guarantees that a user cannot deny performing an operation or initiating a transaction. For example, in an online examination registration system, non-repudiation mechanisms are required to make sure that a student cannot deny registering for a particular examination.
- **Confidentiality**, also referred to as privacy, is the process of making sure that data remains private and confidential, and that it cannot be viewed by unauthorized users or eavesdroppers who monitor the flow of traffic across a

network. Encryption is frequently used to enforce confidentiality. Access control lists (ACLs) are another means of enforcing confidentiality.

- **Integrity** is the guarantee that data is protected from accidental or deliberate (malicious) modification. Like privacy, integrity is a key concern, particularly for data passed across networks. Integrity for data in transit is typically provided by using hashing techniques and message authentication codes.
- **Availability** which ensures that the system remains avail- able for legitimate users. The goal for many attackers with denial-of-service attacks is to crash an application or to make sure that it is sufficiently overwhelmed so that other users cannot access the application

2. Reasons for Vulnerability

Web applications are considered the path of least resistance for cyber-attacks for a number of reasons:

1) They are constantly exposed to the Internet and are easy to probe by outside attackers using freely available tools that look for common vulnerabilities such as SQL Injection.
2) Web applications are easier to attack than traditional targets such as the network and host operating system layers which have been hardened over time. Furthermore, networks and operating systems are protected by mitigating controls such as next-generation firewalls and IDS/IPS systems.
3) New applications are driven by short development cycles that increase the probability of design and coding errors. This is much so because security is often overlooked when the key objective is rapid time-to-deploy.
4) In some cases, applications are assembled from hybrid code obtained from a mix of in-house development, outsourced code, third-party libraries and open source without verifying components for critical vulnerabilities.
5) Modern applications are likely to present a larger attack surface with Web 2.0 technologies that incorporate complex client-side logic such as JavaScript (AJAX) and Adobe Flash.

3. Stages of Attack

Attacking a Web application normally takes place in stages as follows:

- *Surveying / assessing the target:* This is an attempt to determine the characteristics of the application including supported services, protocols, potential vulnerabilities and entry points. The information gathered here is used by the attacker to plan the initial attack.
- *Exploitation/ penetration*: This is the initial attacking stage which, in the simplest cases, usually relies on the same entry point used by legitimate users such as the logon page or pages that do not require authentication.
- *Escalation of privileges*: At this stage the aim is to acquire administrative and other high-level privileges that will enable the attacker take control of vital resources.
- *Sustained access*: This is an attempt to ensure future access and may involve planting back-door codes. The attacker also tries covering tracks of the attack by clearing logs and hiding any malicious tools.
- *Denial of Service*: For some attackers this may be the last resort when they are unable to gain access. The aim is to prevent users from using the application by flooding the server with an overwhelming number of spurious requests such as SYN flood attack.

4. Threat Types

Threats to Web applications have been categorized by Microsoft using the acronym STRIDE.[4] The acronym STRIDE stands for:

- *Spoofing:* Using a fake identity to attempt gaining access to the application.
- *Tampering*: Unauthorized modification of data.
- *Repudiation*: The ability of users to deny performance of a specific transaction or action.
- *Information Disclosure*: Unwanted disclosure of private data.
- *Denial of Service*: The act of making an application unavailable to users.
- *Elevation of Privilege*: Unauthorized elevation of user privilege.

5. The PHP/MySQL Case

Most Web applications rely on the availability of a secure, efficient and robust database system as one of the core services at the server side of things. For security, users only interact with the database through carefully coded scripts that ensure authentication and role-based access to data. At the heart of such scripts lies the structured query language (SQL) which exists in various flavours depending on the database management system. In this

paper we consider the MySQL database engine which is popularly deployed in a significantly large number of Web applications today. MySQL is a fast, powerful database system which is fully capable of coping with large complex databases. It is freely available as a download and can be used on a wide range of operating systems like UNIX, Windows, and Mac OS X. Furthermore, MySQL is available as part of most Web hosting packages for a relatively small fee. The PHP programming language is suitable for building dynamic, interactive Web sites. Thus, PHP programs run on a Web server, and serve Web pages to the visitor on request. Dynamic content is easily created by embedding PHP code within HTML Web pages. This paper focusses on the challenges associated with securing a data-driven Web application that is developed using PHP and MySQL.

6. Sources of Vulnerabilities

Spicing up a Web application to make it dynamic and interactive comes with vulnerabilities that hackers would want to exploit. If not properly secured, the application becomes a target for attackers that are intent on among other things:

- obtaining usernames, passwords, sessions, and cookies in a clandestine manner;
- modifying or destroying data;
- defacing the content of the Web site;
- running external illegal programmes on the server;
- hijacking the entire server, and so on.

Next, we focus on the two main I/O related causes of vulnerabilities in a dynamic interactive Web application:

- Insecure data input into the application, and
- Poor encoding of output data displayed on the Web page

Associated with insecure input is the risk of a client-side code injection attack referred to as cross-site scripting (XSS) using URL encoding whereby a valid input value is replaced by a malicious JavaScript code that redirects the browser to a server-side script on the hacker's server. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user without validating or encoding it. The malicious injected script can access any cookies, session tokens, or other sensitive information retained by the browser and used with the attacked site. Note that an XSS vulnerability can only exist if the payload that the attacker inserts ultimately gets parsed (as HTML in this case) in the victim's browser.

Basically, there are 3 known types of cross-site scripting.[1] These are:

- Persistent (Type I) XSS: This is also known as Stored XSS and occurs when the malicious user input is stored on the target server, such as in a database without taking any precaution. A victim then retrieves the stored data from the web application without that data being made safe to render in the browser. With the advent of HTML5, and other browser technologies, we can envision the attack payload being permanently stored in the victim's browser, such as an HTML5 database, waiting to be sent to the server when an opportunity presents itself.
- Non-Persistent (Type II) XSS: In this case the injected script is reflected off the web server, may be through an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web site. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user's browser. The browser then executes the code because it came from a " trusted" server.
- Type-0 (DOM-Based) XSS As defined by Amit Klein, who published the first article about this issue[1], DOM Based XSS is a form of XSS where the entire tainted data flow from source to sink takes place in the browser, i.e., the source of the data is in the DOM, the sink is also in the DOM, and the data flow never leaves the browser. For example, the source (where malicious data is read) could be the URL of the page (e.g., document.location.href), or it could be an element of the HTML, and the sink is a sensitive method call that causes the execution of the malicious data (e.g., document.write)."

7. Ensuring Secure Logins

A hacker can capture identification and authentication codes by inserting Trojan Horse programs into the login process on a server host and by using macro facilities to record keystrokes on a client node. The weakest link in an otherwise secure system is the user. It is important to ensure that only authorized access to resources is allowed. An important nontechnical factor in information system security is human behaviour, which the potential to render just about any technical security measure useless. It is thus important to integrate information system security into the operational culture of the end-users.

Password Hashing: Two important considerations when hashing passwords are the computational expense, and

the salt. The more computationally expensive the hashing algorithm, the longer it will take to brute force its output.
The native password hashing API in PHP safely handles both hashing and verifying of passwords in a secure manner. There is also a pure PHP compatibility library available for PHP 5.3.7 and later.
The preferred algorithm to use when hashing passwords is Blowfish, which is also the default used by the password-hashing API, as it is significantly more computationally expensive than MD5 or SHA1, while still possessing scalability.
Brute force attacks consist of using powerful computers to try all possible codes to locate the correct ones. Brute force is applied for locating modem or network numbers, user IDs, and passwords. Another option is the crypt () function, which supports several hashing algorithms in PHP 5.3 and later. When using this function, the algorithm selected guaranteed to be available, as PHP contains native implementations of each supported algorithm, in case one or more are not supported by your system.

8. Encoding HTML Entities

This method protects against the common type of XSS attack that involves injecting an HTML element with an src or onload attribute that launches the attacking script. PHP has the htmlentities () function which translates all characters with HTML entity equivalents as those entities, thus rendering them harmless. This approach prevents HTML from being embedded, and therefore prevents JavaScript embedding as well. The resulting safe versions of the input is thus displayed. The htmlentities () function further converts both double and single quotation marks to entities, which will ensure safe handling for all form elements.
The ENT_QUOTES parameter should always be used with htmlentities (). It is this parameter that tells htmlentities () to convert a single quotation mark to its entity and a double quotation mark to its entity. While most browsers will render this, some older clients might not, which is why htmlentities () offers a choice of quotation mark translation schemes. The ENT_QUOTES setting is more conservative and therefore more flexible than either ENT_COMPAT or ENT_NOQUOTES.

9. Content Security Policy

Content Security Policy (CSP) is a feature introduced in HTML 5 and defined in HTTP header directives delivered from the server. Essentially, CSP is a declarative security header available to developers so they can dictate those domains the site is allowed to load content from or initiate connections to when rendered in the web browser. This provides an additional layer of security from critical vulnerabilities such as cross-site scripting, clickjacking, cross-origin access and the like, on top of input validation and whitelisting in the code. An improperly configured header, however, will fail to provide this additional layer of security. The policy is defined with the help of fifteen directives including eight that control resource access.
The directives or functions are briefly described here.
1) **script-src** directive controls a set of script-related privileges for a specific page. As an example, we may specify 'self' as one valid source of script, and https://apis.google.com as another. This is done as follows:
Content-Security-Policy: script-src ' s e l f ' https://apis.google.com
The above directive will make the browser download and execute JavaScript from apis.google.com over HTTPS, as well as from the current page's origin.
2) **base-ur**i restricts the URIs that can appear in a page's <base> element. If this value is absent, then any URI is allowed. If this directive is not declared, the user agent will use the value in the <base> element.
3) **connect-src** restricts the URLs which can be loaded using script interfaces. The APIs that are restricted are:
<a> ping,
Fetch,
XMLHttpRequest,
WebSocket, and
EventSource.
4) **font-src** specifies the origins that can serve web fonts.
It specifies valid sources for fonts loaded using @fontface. For example, Google's web fonts could be enabled via
font-src https://themes.googleusercontent.com
5) **form-action** lists valid endpoints for submission from <form> tags.
6) frame-ancestors specifies the sources that can embed the current page. This directive applies to <frame>, <iframe>, <embed>, and <applet> tags. The directive cannot be used in <meta> tags and applies only to non-HTML resources.

7) **frame-src** was deprecated in level 2 but is restored in level 3. If not present it still falls back to child-src as before.
8) **img-src** defines the origins from which images can be loaded.
9) **media-src** restricts the origins allowed to deliver video and audio.
10) **object-src** allows control over Flash and other plugins.
11) **plugin-types** limits the kinds of plugins a page may invoke.
12) **report-uri** specifies a URL where a browser will send reports when a content security policy is violated. This directive can't be used in ¡meta¿ tags.
13) **style-src** is counterpart of script-src specifically meant for for stylesheets.
14) **upgrade-insecure-requests** instructs user agents to rewrite URL schemes, changing HTTP to HTTPS. This directive is for websites with large numbers of old URL's that need to be rewritten.
15) **worker-src** is a CSP Level 3 directive that restricts the URLs that may be loaded as a worker, shared worker, or service worker. As of July 2017, this directive has limited implementations.

CSP allows application developers to set guidelines on what can be trusted by the client. This means even if the attacker succeeds in injecting some malicious script, the browser will ignore it unless it conforms to the defined CSP (or the attacker can manipulate the CSP). This works fine provided the attacker is unable to alter the stream and alter the HTTP headers to disable the CSP. One way of achieving this is through a man-in-the-middle attack (MITM). This vulnerability can be reduced by using the more secure HTTPS.
A properly enforced CSP ensures that a baseline policy that disallows any inline JavaScript from executing exists. This brings along a good architectural policy that separates data from instructions and removes inline JavaScript from the HTML resulting in a more readable and maintainable code.

10. Escape String Method

All techniques aimed at preventing these attacks are based on ensuring data input to the application gets cleaned before it is parsed. In PHP, one way of achieving this using the escape string. SQL injection is a code injection technique, to attack web applications with malicious SQL statements that are inserted mostly through form fields and intended to pull out important database information. This is a very critical security breach, it can destroy the database completely, so there is a need to secure the apps with proper code standards. Using the mysqli extension, the php function *mysqli_real_escape_string ();* or it's object-oriented version *mysqli::escape_string* can be used to filter input data. This way, all unnecessary extra characters are removed before passed parsing. Thus, the procedural version of calling the escape string function has the form:
$clean = mysqli_real_escape_string ($link, $raw);
To avoid repetition, one approach used in the *StudWare* project, is to replace say the *$_POST* array with a user-defined substitute *$clean* as follows:

```
foreach (arra y keys ( $_POST ) as $key )
{
$clean [$key]= mysqli_real_escape_string ($lnk , $_POST [ $key ] ) ;
}
```

This way all occurrences of *$_POST* are replaced by *$clean* as in:

```
$usern = $clean [ 'usname '] ;
$pwd = $clean [ ' pword ' ] ;
```

rather than

```
$usern = $_POST [ ' usname ' ] ;
$pwd = $_POST [ ' pword ' ] ;
```

In object-oriented form, escape string takes the form:

```
$ c l e a n = $mysqli−> escape_string ( $raw ) ;
```

11. Prepared Statements

In this approach, the SQL statements are constructed as parameterized queries which are sent to and parsed by the database server separately from any parameters thereby pre- venting any malicious SQL injection attack. In the first (pre- pare) stage, the database engine simply compiles the statement ready to accept the filtered parameters. At the execute stage, the parameters are combined with the already compiled statement for execution. This way the statements are compiled and hence protected from alteration before the parameters are introduced. Thus, any input to the query comes in as a parameter conforming to a predefined format ensuring safe execution.

12. PHP Data Objects (PDO) Method

PHP Data Objects (PDO) is a database access layer that provides a uniform method of access to multiple databases. PDO facilitates efficient methods for prepared statements working with objects. The main advantage here is that PDO is a wrapper class that easily connects with multiple databases, and also leads to a code which is more organised and maintained. PDO abstracts the database API and also basic operations that would otherwise be repeated hundreds of times making the application very complex. However, perhaps the most important feature of PDO is its capacity to support re-usable prepared statements. To ensure protection from SQL injection, a prepared statement is the only proper way to run a query if any variable is to be used in it. Thus, we substitute the variable with a placeholder, prepare the query, and execute it while passing the variable separately. An example using a positional placeholder is

```
$stmt =$pdo->pre pare ( 'SELECT* FROM studs WHERE stid=? ') ;
$stmt->execute ( [ $stID ] ) ;
$student = $stmt->f e t c h ( ) ;
```

Alternatively, one could use a named placeholder as follows:

```
$stmt =$pdo->pre pare ( 'SELECT * FROM studs WHERE stid =: stid ') ;
$stmt->execute ( [ 'stid '=>$stID ] ) ;
$student = $stmt->f e t c h ( ) ;
```


13. Conclusion

A brief description of potential threats facing a PHP Web application has been given. The work assumes that the network, server, and operating systems have been properly secured and considers only threats directed at the application. Various vulnerabilities associated with Web applications have been described. The risk of a client-side code injection attack referred to as cross-site scripting (XSS) using URL encoding has been briefly described. Strategies for securing entry point logins and protection against SQL and other malicious code injections have been briefly described with particular reference to PHP -the open source server-side scripting language. The salient features of Content Security Policy introduced in HTML 5 have been briefly described together with its potential for supporting application development. It has been asserted that SQL injection can be prevented by using prepared statements to ensure safe variable data input into queries. Finally, the strength of PDO as a database access layer with inherent support for prepared statements is emphasized.

References

[1] [1] Owasp.org. (2015). Cross-site Scripting (XSS) - OWASP.
[2] [2] DOYLE, Matt.: 'Beginning PHP 5.3', Wiley Publishing, Inc, Indianapolis, 2010.
[3] [3] SCLAR, David and TRACHTENBERG, Adam.: 'PHP Cookbook', O'Reilly Media, Inc, Sebastopol, 2014.
[4] [4] MEIR, J.D. et-all: 'Improving Web Application Security: Threats and Countermeasures' Microsoft Corporation, https://msdn.microsoft.com/en-us/library/ff648647(d=printer).aspx
[5] [5] WEST, Mike and MEDLEY, Joseph:" Content Security Policy"
[6] https://developers.google.com/web/fundamentals/security/csp/
[7] [6] BOSWORTH, Seymour and KABBY, M.E. (Editors):" Computer Security Handbook" John Wiley & Sons, Inc Canada 2002