Injection Prevention in PHP

Injection attacks happen when untrusted data is sent to a code interpreter through a form input or some other data submission to a web application. For example, an attacker could enter SQL database code into a form that expects a plaintext username. If that form input is not properly secured, it would accept malicious SQL code to be executed, and attacker would take over the whole DBMS and might get his hands on the server other than the information leakage.

Prevention:

1. MySQL Injection:

If user input is inserted into an SQL query without filtering, then the application becomes vulnerable to SQL injection, like in the following example:

Example:
Let's say we have file name called: sql_test.php inside it the following code:

```
<?php
	$db_username = "root";
	$db_password = "";
	$db_host = "localhost";
	$db_base = "db_test";
	$con = mysqli_connect($db_host, $db_username, $db_password);

	$unsafe_variable = $_GET['user_input'];

	mysqli_query($con, "INSERT INTO `mytable` (`mycolumn`) VALUES
	('$unsafe_variable')") or die(mysql_error());
?>
```

if user input something like `myvalue'); DROP TABLE mytable;--` as parameter in the get request url
URL would be like this: `http://localhost/sql_test.php?user_input=myvalue'); DROP TABLE mytable;--`

and the query becomes:

```
	INSERT INTO `mytable` (`mycolumn`) VALUES('myvalue'); DROP TABLE mytable;--')
```

The two hyphens -- means comment, so the remaining ') after the two hyphens will be considered as a comment and error will not be triggered and table will be dropped!.

What can be done to prevent this from happening?

Use prepared statements and parameterized queries. These are SQL statements that are sent to and parsed by the database server separately from any parameters. This way it is impossible for an attacker to inject malicious SQL.

You basically have two options to achieve this:

- Using MySQLi (for MySQL):

```
    $stmt = $dbConnection->prepare('SELECT * FROM employees WHERE name = ?');
    $stmt->bind_param('s', $name); // 's' specifies the variable type => 'string'

    $stmt->execute();
    $result = $stmt->get_result();
    while ($row = $result->fetch_assoc()) {
        // Do something with $row
    }
```

- Using PDO (for any supported database driver):

```
    $stmt = $pdo->prepare('SELECT * FROM employees WHERE name = :name');

    $stmt->execute([ 'name' => $name ]);


    foreach ($stmt as $row) {
        // Do something with $row
    }
```

- To see a fully prepared statements code in php, go here:
https://www.w3schools.com/php/php_mysql_prepared_statements.asp

- To read discussions in StackOverFlow about MySQL Injection, go here:
https://stackoverflow.com/questions/60174/how-can-i-prevent-sql-injection-in-php

- Or you can watch a youtube video explaining how to protect your code:
https://www.youtube.com/watch?v=cgwWpd4SqIM

2. LDAP Injection:

LDAP injection attacks can be prevented by sanitizing user-submitted data by escaping specific LDAP special chars.

- if you are using LDAP, you should consider escaping its injections.

Example code:

```
<?php

      // $ds is a valid link identifier for a directory server
      $unsafe_email_address = $_GET[‘user_input’];

      $base   = "o=My Company, c=US";
      $mail = "(mail=".$unsafe_email_address.")";

      $sr = ldap_search($ds, $base, $mail , array("sn", "givenname", "mail"));

      $info = ldap_get_entries($ds, $sr);

      echo $info["count"]." entries returned\n";

?>
```

Using ldap_escape() function to escape LDAP injections: code will be like:

```
<?php

      // $ds is a valid link identifier for a directory server
      $unsafe_email_address = $_GET[‘user_input’];

      $base   = "o=My Company, c=US";
```

```
    $safe_email_address = "(mail=".ldap_escape($unsafe_email_address, "", LDAP_ESCA
    PE_FILTER).")";

    $sr = ldap_search($ds, $base, $safe_email_address , array("sn", "givenname", "m
    ail"));

    $info = ldap_get_entries($ds, $sr);

    echo $info["count"]." entries returned\n";

?>
```

- To read more about LDAP injection escaping, go here:
https://www.php.net/manual/en/function.ldap-escape.php

3. SHELL Injection:

OS Commands injection attacks can be prevented by sanitizing user-submitted data by escaping specific OS Command special commands.

If you are using OS Commands with User input, it must be filtered:

- exec — Execute an external program
- passthru — Execute an external program and display raw output
- proc_close — Close a process opened by proc_open and return the exit code of that process
- proc_get_status — Get information about a process opened by proc_open
- proc_nice — Change the priority of the current process
- proc_open — Execute a command and open file pointers for input/output
- proc_terminate — Kills a process opened by proc_open
- shell_exec — Execute command via shell and return the complete output as a string
- system — Execute an external program and display the output

Source: https://www.php.net/manual/en/book.exec.php

To escape Shell Injection, we use two functions: escapeshellarg() and escapeshellcmd()

Example of protection:

```
<?php

    // We allow arbitrary number of arguments intentionally here.
    $command = './configure '.$_POST['user_input_configure_options'];

    $escaped_command = escapeshellcmd($command);

    system($escaped_command);

?>
```

Or another protection example:

```
<?php

    system('ls '.escapeshellarg($_POST['user_input_path']));
```

```
?>
```

- For more info
https://www.php.net/manual/en/function.escapeshellcmd.php
https://www.php.net/manual/en/function.escapeshellarg.php

4. XPATH Injection:

XPath injection attacks: can be prevented by escaping single and double quotations from input. Using str_replace() to remove all single and double quotations from input, or by addslashes() to add slashes before single or double quotations.

Example Protection:

```
<?php

   $unsafe_u_input = $_POST['user_input_xpath'];

   $safe_unput = str_replace($unsafe_u_input, "\"", ""); \\ removing "

   $safe_unput = str_replace($safe_unput, "'", ""); \\ removing '

   $note=<<<XML
   <note>
   <to>Tove</to>
   <from>Jani</from>
   <heading>Reminder</heading>
   <body>Do not forget me this weekend!</body>
   </note>
   XML;

   $xml = new SimpleXMLElement($note);

   $result = $xml->xpath($safe_unput);

   print_r($result);

?>
```

5. NoSQL Injection:

NoSQL Injection attacks: can be prevented by:

- Input data type casting, which will make sure that all input data are String, an array type must not be allowed.
- Escaping single and double quotations or replacing input with regular expression validation.

Simple Example:

```
$conn = new Mongo('localhost');
$db = $conn->test;
$collection = $db->items;
```

```
$userName = $_GET['username'];
$userPass = $_GET['userPassword'];

$user = $db->$collection->findOne(array('username'=> 'user1', 'password'=>
'pass1'));

foreach ($user as $obj) {

   echo 'Username' . $obj['username'];
   echo 'password: ' . $obj['password'];
   if($userName == 'user1' && $userPass == 'pass1'){

         echo 'found'
   }
   else {

         echo 'not found'
   }

}

$conn->close();
```

To prevent NoSQL Injection

- Add @strval($input) or @(string)$input function to override input type and replace the word `array` with nothing if exists with str_ireplace() function.

- Escaping single and double quotations with addslashes() function or replacing input with regular expression validation preg_replace('/[^a-z0-9]/i', '\', $input);

So in the above example, the two red lines will be like this:

```
$userName = @(string)$_GET['username'];
$userPass = @(string)$_GET['password'];

$userName = str_ireplace("array", "", $userName);
$userPass = str_ireplace("array", "", $userPass);

$userName = addslashes($userName);
$userPass = addslashes($userPass);
```

or more complex way (less lines)

```
$userName = addslashes(str_ireplace("array", "", @(string)$_GET['username']));
$userPass = addslashes(str_ireplace("array", "", @(string)$_GET['password']));
```

to watch a video of how this attack can be accomplished, go here:
https://www.youtube.com/watch?v=nuMtYsE6guM

to read more about NoSQL injection go here:
https://blog.securelayer7.net/mongodb-security-injection-attacks-with-php/

6. Log Injection:

Log Injection attacks: can be prevented by escaping new line character entry, for windows CRLF and Unix LF and Mac CR.

Whatever logging mechanism you are using, if user's input included in the log, make sure you remove break lines, before logging the data.

It can be achieved with str_replace() function

Example:

```
$unsafe_user_input = $_GET['user_input'];
$safe_log = str_replace($unsafe_user_input, "\r", ""); // removes CR
$safe_log = str_replace($safe_log, "\n", ""); // removes LF
$log_data = "User did an error: ".$safe_log; // then you can record user input
```

To learn more about Injection:
https://www.youtube.com/watch?v=rWHvp7rUka8
https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html
https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A1-Injection