Network and Web Security

Answers to Selected Questions for Tutorials 5, 6 and 7*

March 4, 2021

Tutorial 5: SQL Injection Vulnerabilities

2 Blind SQL injection vulnerabilities in DVWA

1. Which lines of the DVWA source code in each category are vulnerable to SQL injection attacks?

   In both categories, line 8 (in the low security levels), line 9 (in the medium security levels) and line 8 (in the high security levels) are vulnerable to SQL injection attacks. They allow untrusted user input (from `$_GET['id']`*) to contaminate an SQL query that is then sent to the database server for execution using PHP's* `mysqli_query()` *function.*

2. Which lines of the DVWA source code in each category attempt to fix these vulnerabilities? Why are they inadequate?

   In both categories, line 5 of the PHP code for the low security level reads the user ID to be queried from the query string in the URL (`$_GET['id']`*). In higher levels, the input is read from different locations that make the SQL injection attack harder: in the medium security levels it is read from the value of the selected option in a* `<select>` *box in a HTML form (*`$_POST['id']`*), but the attacker can submit a non-approved value for the* `id` *parameter by intercepting and modifying the outgoing HTTP request. The high security levels allow the user to change their user ID — in the* SQL Injection *category by updating the server-side session data store (*`$_SESSION['id']`*) and in the* SQL Injection (Blind) *category by modifying their cookie (*`$_COOKIE['id']`*) — but the supplied value is never validated before being concatenated with the SQL query that is eventually executed by the database server.*

   Additionally, the PHP code for the high security level in the SQL Injection (Blind) *category attempts to frustrate timing attacks by sleeping at random (for at most 4 seconds) when the database query returns an empty table (i.e., when the answer to the "yes/no" question is "no"). This is inadequate because the attack can simply cause the database server to sleep for more than 4 seconds, so that the "no" outcome can still be identified by the attacker.*

3. How could these vulnerabilities be fixed properly? Compare your answer with the source code for the *impossible* security level (which is invulnerable to SQL injection) in each category.

   The fix for all security levels in both categories is the same: parameterised statements should be used in order to build the SQL query that will be executed by the database server, rather than concatenating variables containing untrusted user-supplied input with string literals and passing the resulting string as the second parameter to `mysqli_query()`.

*Thanks to Chris Novakovic `c.novakovic@imperial.ac.uk` for preparing this material.

Tutorial 6: Client-Side Web Vulnerabilities - Part I

1 Cross-site scripting vulnerabilities in DVWA

1. Which lines of the DVWA source code in each category attempt to prevent XSS attacks? Why are they inadequate?

   In the XSS (Reflected) *category, line 6 of the PHP code in the medium security level strips all occurrences of the string* `<script>` *from the* `name` *URL query string parameter before it is inserted into the HTML; similarly, line 6 of the PHP code in the high security level strips anything matching the regular expression* `/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i` *from the* `name` *URL query string parameter. Neither are effective because there are ways of triggering script execution on a page that don't involve the use of* `<script>` *elements (e.g., the* `onerror` *attribute of an* `<img>` *elements whose source image can't be loaded).*

   In the XSS (Stored) *category, the* `message` *parameter in the POST form data is insufficiently sanitised in the PHP code in the low security level (PHP's built-in* `stripslashes()` *function won't prevent HTML injection), but is correctly sanitised in the higher security levels. The* `name` *parameter isn't correctly sanitised in any security level: the prevention attempts at each security level are identical to their counterparts in the* XSS (Reflected) *category.*

2. How could these vulnerabilities in DVWA be fixed properly? Compare your answer with the source code for the *impossible* security level (which is invulnerable to XSS) in each category.

   In both categories, in all security levels, the solution is the same: all user-supplied input, whether read from the URL query string or the database, should be appropriately sanitised with PHP's built-in `htmlspecialchars()` *function (ideally using* `ENT_QUOTES` *as the second parameter) before being inserted into the HTML.*

Tutorial 7: Client-Side Web Vulnerabilities - Part II

1 Cross-site request forgery vulnerabilities in DVWA

1. Which lines of the DVWA source code in the CSRF category attempt to prevent CSRF attacks? Why are they inadequate?

   On line 5 of the PHP code in the medium security level, the server checks that the server name is contained within the referring URL for the incoming password change request; this can easily be bypassed (e.g., with the use of an iframe), not least because the code doesn't check that the server name is the origin of the referring URL: it could in fact appear anywhere in the URL. On line 5 of the PHP code in the high security level, the server checks the validity of an anti-CSRF token, but this token is static and per-user, and can therefore be stolen (e.g., using an exploit in another part of DVWA) and reused in a CSRF exploit at any later time.

2. How could these vulnerabilities in DVWA be fixed properly? Compare your answer with the source code for the *impossible* security level, which is invulnerable to CSRF.

   Correct implementation of an anti-CSRF token, or requiring the user to correctly supply some authenticating information (e.g., their current password) before performing the password change, will fix the vulnerability.

2 A content security policy for DVWA

1. You'll eventually find that, no matter how sophisticated your content security policy is, you won't be able to mitigate all possible client-side code injection attacks without affecting DVWA's functionality. Why is this? (You may find it helpful to look at the source code of various web pages served by DVWA, to understand

what functionality is being blocked by the CSP.)

DVWA's HTML output contains inline scripts (e.g., in `onclick` *attributes). This means that the* `Content-Security-Policy` *HTTP header must contain the directive* `script-src 'unsafe-inline'` *to allow these inline scripts to execute, to avoid breaking functionality within DVWA. Unfortunately, this also permits injected scripts (e.g., those present as a result of a successful XSS attack) to execute in the browser, thus undermining the content security policy's effectiveness.*

2. How could DVWA be redesigned so that it provides the same kind of functionality while allowing a more effective content security policy to be deployed?

   All inline scripts should be removed from DVWA and replaced with code in an external script that modifies the DOM (e.g., inline event listeners for elements that have them should be replaced with calls to `addEventListener()` *on the target element). This external script should then be hosted on a separate high-security trusted origin (e.g.,* `http://10.6.66.43`*), and the content security policy can mandate that the browser should only execute scripts originating from this origin (e.g.,* `script-src http://10.6.66.43`*).*

   Ideally, the use of `eval()` *should also be eradicated from DVWA's JavaScript code after it has been refactored, so that the* `script-src` *directive need not contain* `'unsafe-eval'`.