Network and Web Security

# Tutorial 9 - Client-Side Web Vulnerabilities - Part 2*

February 26, 2024

## 1 Cross-site request forgery vulnerabilities in DVWA

In this tutorial, we'll continue using the `dvwa` VM used for Tutorials 4, 6, and 7. We have seen in module 21 that *cross-site request forgery* (*CSRF*) attacks force a victim's web browser into performing unwanted yet authorised actions. A victim's browser is coerced into initiating a HTTP request that performs an action inside a web application (possibly one that requires authentication, if the victim is logged in to the web application when the attack occurs) without the victim's intent (or, worse, even without their knowledge). This attack exploits a badly-designed web application's trust in the behaviour of the victim's web browser: it relies on the web application's inability to distinguish HTTP requests that were genuinely initiated by a user action from ones that were initiated automatically without their consent.

Let's take a look at a feature in DVWA that's vulnerable to CSRF attacks.

1. From the left-hand menu in DVWA, select **CSRF**.
2. This part of DVWA allows the currently logged-in user to change their password. (The PHP source code for this part of the page is shown when you click the **View Source** button.) Enter a new password twice, click the **Change** button, and select **Logout** from the left-hand menu. You should be able to log in with the new password.
3. Prepare a malicious web page that, when loaded by a victim logged in to DVWA, causes their DVWA password to be set to the name of the current security level (either `low`, `medium` or `high`) without their knowledge or permission. Check whether your web page is effective by loading it while logged in to DVWA in one web browser and trying to log in as that user with the new password in another web browser.

As you move through the security levels, you'll need to combine your knowledge of CSRF and XSS (and reuse a part of DVWA that you exploited earlier in this tutorial) in order to bypass the anti-CSRF mechanisms that have been put in place.

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

After constructing a web page that successfully changes the victim's password when DVWA is set to the *high* security level, you've completed this tutorial and can stop reading here. If you'd like to learn how to make DVWA's defences against client-side web attacks more robust, keep reading…

## 2 A content security policy for DVWA (Optional)

The *Content Security Policy* (*CSP*) of a web application restricts the usage and/or origins of many different types of resources, such as scripts (both inline and external), styles (again, both inline in `style` attributes and externally in stylesheet files), images, and iframes. For instance, the policy

*Based on original content by Chris Novakovic `c.novakovic@imperial.ac.uk`.

```
img-src *; child-src 'self'; media-src 'none'
```

allows images to be loaded from any origin, including `data:` URIs embedded in the page itself, but allows frames and iframes to be loaded only from the same origin as the page; HTML5 media resources (i.e., in `<video>` or `<audio>` elements) should not be loaded at all. The full policy language is defined in the W3C specification, although a much more readable introductory guide can be found at `https://content-security-policy.com`.

The CSP standard is intended to provide *defence in depth* for web applications and their users: a content security policy won't prevent all XSS attacks (not least because not all browsers understand the `Content-Security-Policy` HTTP header, or enforce it in exactly the same way as other browsers), but it provides an additional hurdle that an attacker must overcome in order to successfully compromise the web application's users. The hope is that a large enough number of users is protected by at least one of the web application's security mechanisms that the damage caused by a successful client-side code injection attack against the web application itself is extremely limited.

The DVWA developers — unsurprisingly, given that it's designed to be vulnerable to all sorts of web attacks — didn't define a content security policy for DVWA. We can make DVWA users with modern web browsers significantly less vulnerable to client-side code injection attacks by serving pages with a suitable content security policy.

1. Devise a `Content-Security-Policy` header that could be deployed in HTTP responses by DVWA. Your header should mitigate the effects of client-side code that gets injected into the HTML documents served by DVWA.
2. Test your header by injecting it into DVWA's HTTP responses using Burp's proxy. In the Burp user interface, select the **Proxy** tab, then the **Options** sub-tab. In the *Match and Replace* list, **Add** a new rule of the type **Response header**; leave the **Match** field blank, and enter `Content-Security-Policy: <your policy>` into the **Replace** field. Click **OK**, make sure your new rule is enabled, and enable Burp's proxy in your web browser.
3. Try to perform one of your earlier XSS attacks against DVWA, and check whether execution of the injected script is blocked by the browser (if it is, you should receive notification of a CSP violation in the developer console).
4. Try to use DVWA in a benign way. If any of your actions get blocked, try to tweak your policy so that benign actions are permitted while malicious actions are still blocked.

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.)

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?