CO331: Network and Web Security

Tutorial 8: Client-Side Web Vulnerabilities*

March 2, 2018

Tutorials 5 and 7 focused on server-side vulnerabilities in web applications. In this tutorial, we'll continue using the copy of the *Damn Vulnerable Web Application (DVWA)* on the `dvwa` VM to investigate two prominent client-side vulnerabilities in web applications: *cross-site scripting* and *cross-site request forgery*.

1 Cross-site scripting vulnerabilities in DVWA

We saw earlier in the course that any scripts executing on a web page (e.g., those included with the `<script>` element) run in the same origin as the including page. This gives included scripts access to a wealth of valuable or sensitive data: the origin's cookies, Web Storage and IndexedDB databases can all be read and written, and arbitrary AJAX requests can be made to URLs in the same origin.

Unless a web application is carefully designed, it may be vulnerable to *cross-site scripting* (*XSS*) attacks, in which an attacker manages to subvert the HTML document produced by the web application and served to users by injecting their own malicious client-side code into it that is then executed by the victim's web browser. Like the genuine scripts included on the page, this malicious code runs in the same origin as the web application, and may be able to exfiltrate sensitive data (such as the victim's session cookie) to a server under the control of the attacker, or perform privileged operations with the victim's authority (e.g., by initiating same-origin AJAX requests to the web application's API endpoints).

In this tutorial, we'll encounter two types of XSS vulnerability:

Reflected XSS. This occurs when a web application directly includes untrusted input from the URL's query string or a submitted form in the HTML document it outputs. If the untrusted input is not correctly sanitised before it is included in the HTML document, it may be possible to craft a malicious query string or form submission that causes the attacker's client-side code to be injected into the page and executed by the victim's browser.

Stored XSS. This occurs when a web application stores untrusted input from the URL's query string or a submitted form (e.g., in a server-side file or database table) and later uses this input in a HTML document that it outputs. As with reflected XSS, if this input is not correctly sanitised before it is read from storage and included in the output HTML document, it may be possible to inject a malicious client-side script into the HTML. Stored XSS attacks are often more damaging than reflected XSS attacks, since the malicious script is usually injected into the HTML document for *every* user who subsequently visits the page.

Two parts of DVWA are specifically designed to be vulnerable to XSS. Let's start with the reflected XSS category.

1. Start a "fake" web server on port 8000 on `kali-vm` using Ncat, a variant of Netcat created by the Nmap developers that makes it easier to listen for multiple and concurrent incoming connections:

   ```
   ncat -lkc "perl -e 'while (defined(\$x = <>)){ print STDERR \$x; last if \$x eq qq#\\r\\n
   # } print qq#HTTP/1.1 204 No Content\\r\\n\\n#'" 8000
   ```

   When a web browser connects on port 8000, Ncat will spawn a Perl program that reads each line of the

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

HTTP request, stops reading when the end of the request header is reached, and responds with a simple HTTP response (`204 No Content`).

2. Open either Chrome or Firefox. If you choose to use Chrome, you'll need to bypass its built-in XSS prevention mechanism: in Burp Suite, select the **Proxy** tab, then the **Options** tab, enable the **Disable browser XSS prevention** rule in the *Match and Replace* list, and enable Burp's proxy in Chrome. This will inject the `X-XSS-Protection: 0` header into all HTTP responses that pass through Burp, disabling Chrome's XSS Auditor (which blocks the execution of a script if its source code is found in the HTTP request). You can turn off interception in Burp: the new header will still be injected into the response.
3. Visit `http://10.6.66.42/dvwa/` in your chosen browser.
4. Log in to DVWA with the user name `admin` and the password `password`.
5. From the left-hand menu, select **XSS (Reflected)**.
6. This part of DVWA allows you to enter your name into a text box. After entering your name and clicking the **Submit** button, DVWA greets you using the name you gave. (The PHP source code for this part of the page is shown when you click the **View Source** button.)
7. Craft a malicious URL that mounts an XSS attack against logged-in visitors to this page: when the URL is accessed, the resulting page should contain an injected script that causes the victim's web browser to leak their session cookie (`PHPSESSID`) to your "fake" web server by triggering a HTTP request to it that contains the value of that cookie. You'll know if your XSS attack has succeeded by looking at the output from `ncat` in the terminal: if your attack was successful, the web browser will have made a HTTP request to your "fake" web server that will be visible in the terminal, with the session cookie encoded somewhere in the request.

If you're stuck, you may find the *OWASP XSS Filter Evasion Cheat Sheet* (`https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`) helpful: it lists some of the common ways you can work around the presence of anti-XSS blacklists in web application code. It's also useful to bear in mind that HTML parsers in web browsers are incredibly forgiving of malformed HTML documents, and that an XSS attack may work even if you can't find a way to get well-formed HTML past the blacklist.

Move through the security levels as you find attacks that succeed. This category is completed when you successfully perform a reflected XSS attack against the code for the *high* security level.

After completing the reflected XSS category, move on to the stored XSS category.

1. From the left-hand menu in DVWA, select **XSS (Stored)**.
2. This part of DVWA behaves like a guestbook, and allows you to enter your name and a message that will be displayed to all visitors to the page. Enter your name and message into the boxes, and click the **Sign Guestbook** button to save your message in the DVWA database. (The PHP source code for this part of the page is shown when you click the **View Source** button.) From now on, every time the page loads, all previously-submitted messages are displayed further down on the page.
3. Submit a malicious name and/or message that gets stored in the DVWA database and causes the web browsers of future visitors to this page to leak their session cookie to your "fake" web server, as before.

Move through the security levels as you find attacks that succeed, making sure you reset the DVWA database after you complete each level so you begin the next level with a single benign guestbook message:

1. From the left-hand menu in DVWA, select **Setup / Reset DB**.
2. Click the **Create / Reset Database** button.

When you find a stored XSS attack that succeeds against the code for the *high* security level, you've completed DVWA's XSS challenges.

1. Which lines of the DVWA source code in each category attempt to prevent XSS 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 XSS) in each category.

2 Cross-site request forgery vulnerabilities in DVWA

There are other ways to trick a victim's web browser into performing unauthorised actions other than via the injection of scripts into the victim's browser. One such attack is *cross-site request forgery* (*CSRF*), in which 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 authorisation (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…

3 A content security policy for DVWA (Optional)

In 2012, recognising the pervasiveness of client-side web attacks and their potentially enormous impact when successful, the World Wide Web Consortium (W3C) standardised a client-side web security mechanism named the *Content Security Policy* (*CSP*). The intent of the standard is to provide web developers with a means of describing

the resource types that web browsers should and shouldn't load or interpret when parsing their web pages, and where those resources may or may not be loaded from. Version 1.0 of the standard was released in 2012, and was quickly implemented in the major web browsers (replacing earlier ad-hoc implementations of precursors to the Content Security Policy); version 2 followed in 2014, defining more fine-grained resource types. Version 3 is currently being drafted.

A suitable content security policy for a web application must be crafted by the developer and delivered via the `Content-Security-Policy` HTTP response header. It may restrict 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

```
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 (and HTML5 Web Workers) 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?