Web Security 2: XSS and CSRF Attacks

Wen Zhang, Joanna Yang

November 14, 2017

1 Cross-Site Scripting (XSS)

A cross-site scripting (XSS) attack is when the browser will execute unintended script that the attacker embeds in the victim's site. There are two types of XSS: stored and reflective. The main reason XSS attacks work is because the script that gets embedded in the victim's web page takes the origin of the victim's web page. Therefore, the browser will run the unintended embedded script. Some examples of malicious script are, but not limited to, stealing a user's cookie and do actions under the user's account, such as automatically re-tweeting posts[1] or sending friend requests.[2]

1.1 Stored XSS

Here is an example of a Stored XSS attack workflow:

1. Attack makes a POST HTTP request which includes the embedded script (could be contained in a simple comment or post) to the victim server `myspace.com`
2. The victim server `myspace.com` stores this POST HTTP request in their database
3. Alice will request the victim site's web page and load the website `myspace.com`
4. Alice's browser will run the attacker's embedded script in the victim's web page

Other than posts and comments where malicious users can input script, scripts can be embedded in images, which also take the origin of the embedded page, and also in the CSS.

In conclusion, stored XSS attack targets vulnerable web sites that fail to ensure that content uploaded does not contain embedded script.

[1] TweetDeck HackedPanic (And Rickrolling) Ensues `https://www.wired.com/2014/06/tweetdeck-hacked/`.

[2] The MySpace Worm that Changed the Internet Forever `https://motherboard.vice.com/en_us/article/wnjwb4/the-myspace-worm-that-changed-the-internet-forever`.

1.2 Reflective XSS

Here is an example of a Reflective XSS attack workflow:

1. The victim Alice will visit the malicious `attacker.com`
2. Alice will receive the `attacker.com` web page and press on a well-crafted URL provided by the attacker
3. This URL link will make a request to the vulnerable server `bank.com`
4. `bank.com` will echo the user input provided in the attacker's URL to Alice's browser
5. Alice's browser will run the embedded script from the attacker via the URL and perform unintended actions.

One of the most common targets for reflected XSS attacks are search bars because most servers will just reflect the keyword in the URL. In addition relying on victims to press on URL links, attacks can also embed the URL link in images. Therefore, whenever the victim's browser loads the image, it will make HTTP requests in the same way.

1.3 Defenses against XSS

As discussed above, XSS attacks target on vulnerable websites that do not check the user inputs and blindly passes these inputs to browsers. Also, because script is embedded, the attackers are able to bypass same origin policy and have the malicious script run in the victim user's browser. The following defenses target at checking the user inputs for malicious tags.

Input Validation One method of defending against XSS is checking the user's inputs against a whitelist, or known as sanitizing user inputs. Blacklists are usually not recommended because it is difficult to encompass all tricky and corner cases. For example, a blacklist may try to remove every instance of `<script>` and `</script>`. However, it can be easy for victim server to miss strange corner cases such as inputs like `<scr<script>ipt>` and so forth. Whitelist will limit the scope of acceptable inputs, which makes the system less vulnerable to these corner cases. For example, if the server is only expecting alphanumeric characters in an input form, the server can either reject the input or automatically remove the characters that are not alphanumeric.

Output escaping Another method is escaping user inputs, which means replacing certain characters with the escape sequence, before they are passed to the HTML parser. The HTML parser will look for special characters, such as `< > & " '`. If the user's input provides these characters, you have to escape the parser by replacing these special characters with the escape sequence so that the parser will interpret these characters as the text character. An example is that the escape sequence of `<` is `&lt;`. Now, instead of having `<script>` running, the web page will display the tag as text. The same idea can be applied to escaping SQL inputs.

Content-security Policy (CSP) The Content security policy is implemented by the browser to help mitigate XSS attacks. The web servers will specify a whitelist of domains that are allowed for executable scripts. All other scripts, including in-line scripts, are not allowed.

By adding a `Content-Security-Policy:` header, the client servers can specify which script, image, and media source that allow script. In addition, servers can also globally opt out any script, but this will render an uninteresting web page.

2 Cross-Site Request Forgery (CSRF)

A cross-site request forgery (CSRF) attack executes unwanted actions on behalf of a user on a website where the user is already authenticated. This attack is ranked #5 on OWASP Top 10 in 2010 and #8 in 2013.

2.1 How it works

Here is an example CSRF attack workflow:

1. User logs onto bank website `bank.com` and establishes a session.
2. User then visits attack site `evil.com`.
3. Attack site `evil.com` returns a malicious page containing JavaScript code; the code sends a POST request to `bank.com` that transfers funds to the attacker.
4. User's browser sends the fund transfer request to `bank.com`, automatically attaching the user's authentication cookie for `bank.com`.
5. Bank website `bank.com`, after verifying user's authentication cookie, transfers funds from user to attacker.

From the bank website's perspective, the fund transfer request looks like a legitimate request from the user; after all, a valid authentication cookie is provided. The request, however, originates from the attack site `evil.com`, while legitimate requests should only result from user actions on `bank.com`. The problem is that cookie authentication is insufficient when side effects occur.

Unfortunately, many popular websites have fallen victim to CSRF exploits. For instance, a CSRF attack on YouTube in 2008 enabled an attacker to add videos to a user's "Favorites", to send arbitrary messages on a user's behalf, etc. As another example, a CSRF attack against Facebook in 2009 stole personal information from any user who visited an infected page.[3] Indeed, CSRF attacks can be quite powerful as they allow an attacker to act as another user on any site that the user is logged onto.

2.2 Defenses against CSRF

As discussed above, CSRF attacks occur when a website (`bank.com`) processes a state-changing request (fund transfer) that originates from a different website (`evil.com`). Therefore, to defend against CSRF attacks, a website must make sure that every state-changing request comes from one of its own pages. Here are three common defenses:

[3] InternetNews.com. Facebook Hit by Cross-Site Request Forgery Attack. `http://www.internetnews.com/security/article.php/3835596/Facebook+Hit+by+CrossSite+Request+Forgery+Attack.htm`.

Secret token validation. The server, for each action, demands a secret token (called a *CSRF token*), which is sent to the browser only when the user browses to the action. Concretely, the server:

- Maintains state that associates each user's CSRF token with her session ID;
- Embeds the CSRF token in every form on the site (e.g., in a hidden field on a fund transfer form);
- On receiving a request for action, checks that the supplied CSRF token indeed matches that of the user's session.

Secret token validation is effective as long as the CSRF token cannot be obtained by an adversary. Fortunately, a malicious site cannot simply read the token off of a target site due to the same-origin policy (e.g., a browser wouldn't allow `evil.com` to obtain contents from `bank.com`). And of course, the token must be hard to guess.

Because the CSRF token is only embedded in legitimate forms and cannot be obtained by an attacker, the website can be assured that each action it receives originates from a form on one of its own pages, provided that the site is willing to maintain a large table of CSRF tokens for all active sessions.

Referer validation. When a browser issues an HTTP request, it includes a `Referer` header containing the URL that initiated the request. The website can simply check, for each action, if the request originates from a legitimate URL in its domain. For example, Facebook's login service returns an error message when a login request is initiated from a page not located on `facebook.com`.

Unfortunately, the `Referer` header is not always available due to, e.g., network stripping by corporations out of privacy concerns: the header reveals the URL of the page that leads to a website and may leak confidential information about a corporation's intranet. Other sources of `Referer` header blockage include users' browser preferences and browser stripping during HTTPS to HTTP transitions.

What should a web server do when a state-changing request comes in without a `Referer` header? A strict policy would deny the request, potentially damaging usability in case the request is actually legitimate. A lenient policy, on the other hand, would allow such a request, while possibly enabling a CSRF attack. This usability-security tradeoff is a decision that has to be made by the website.

Custom HTTP headers. Issue all state-changing requests using `XMLHttpRequest`, attaching a custom HTTP header. The server can then check for this header and reject any state-changing requests without the header. A commonly-used header for this purpose is `X-Requested-By: XMLHttpRequest`.

This defense is enabled by the same-origin policy: browsers forbid sending requests with custom HTTP headers to URLs of a different origin. This restriction also implies that the site can no longer support state-changing requests across domains by default, although XHR2 allows whitelisting cross-site requests on the server side.

3 Robust Defenses against CSRF

This section is based on Brandon Lin's presentation on the paper "Robust Defenses for Cross-Site Request Forgery" by Barth, et al. The presentation covers session initialization vulnerabilities, in particular the login CSRF attack, and discusses general CSRF defenses and pitfalls.

3.1 Login CSRF

Special care needs to be taken to prevent CSRF attacks that target the login flow, i.e., *before* a user is logged in—this kind of attack is termed *login CSRF*. In a login CSRF attack, an attacker sends a forged login request to a legitimate website, causing the user's browser to be logged into the website as the attacker. As an example:

1. User visits a malicious webpage on `evil.com`.
2. The page sends a login request to bank website `bank.com` with *attacker's* credentials.
3. Bank website verifies the credentials and sends back a `Set-Cookie` header, instructing the browser to store attacker's session ID for `bank.com`.
4. User's subsequent requests to `bank.com` are accompanied by attacker's session ID; user is now logged in as attacker.

Login CSRF attacks can have serious consequences. For example, if such an attack is mounted against a search engine like Google, the user's search history can be saved and associated with the attacker's account, leaking sensitive queries. As another example, if a user is unknowingly logged onto PayPal as an attacker, she might link her credit card or bank account to the attacker's account, potentially causing monetary loss.

3.2 Review of CSRF defenses

How effective are the defenses from §2.2 in general, and how effective are they against login CSRF attacks in particular?

Secret token validation. The effectiveness of secret token validation depends on how the token is chosen. This defense is complicated by login CSRF attacks: in order to prevent login CSRF, a user must be assigned a "pre-session" token before login; this mechanism must be implemented carefully as there are multiple pitfalls as explained below.

One way is to use a *session-independent nonce* as the CSRF token: generate a random token and send it in a cookie when a user first visits the site; on a subsequent request, the server can validate the token. This method doesn't protect against an active network attacker, who can overwrite the session-independent nonce with the CSRF token of her own and proceed with the login request (see §3.3).

Another way is to use a *session-dependent nonce*, where the CSRF token is derived from or associated with the session identifier:

- The simplest way is to use the session identifier directly as the token, embedded in every form. One downside is that, if the page content is leaked (e.g., uploaded to the browser's bug tracker), the session ID is also leaked.

- A more refined version, as described in §2.2, is to bind a randomly-generated CSRF token to a session ID on the server side. This requires maintaining a large table for this mapping.

- To avoid storing the mapping, we can use the HMAC of the session ID as the CSRF token:

$$\texttt{csrf\_token} = \text{HMAC}(\texttt{secret\_key}, \texttt{session\_id}).$$

  Given that all servers of a website share the HMAC secret key, each server can independently validate a CSRF token with no need for a mapping table. Furthermore, the cryptographic properties of HMAC ensure that the CSRF token is not guessible by an adversary.

Referer validation. From discussion in §2.2, `Referer` header validation is an effective defense against CSRF, including login CSRF, *given that* the header is present. To measure how often the `Referer` header is suppressed, Barth, et al. bought banner ads and logged whether the header is present in each request for the banner. They found out that:

- Over HTTP, 3–11% of requests have a missing/invalid `Referer` header;

- Over HTTPS, 0.05–0.22% of requests have a missing/invalid `Referer` header.

This finding supports the hypothesis that the `Referer` header is mostly suppressed by the network, not by the browser, because the network is only able to tamper with HTTP traffic. In particular, this means that referer validation effectively defends against login CSRF because most login pages use HTTPS.

Proposal: origin header. Barth, et al. propose a new header, `Origin`, which accompanies every HTTP POST request indicating the *origin* that has made the request. Unlike the `Referer` header, the `Origin` header doesn't leak the entire URI of the previous page or get sent on all HTTP requests. As it provides better privacy, the `Origin` header will hopefully not be suppressed. A web server can use the `Origin` header to determine if a POST request just as it does with the `Referer` header.

3.3 Cookie overwriting

The ability for an attacker to overwrite cookies opens doors to more session initialization attacks. One such attack is session fixation: if a site assigns a "pre-session" to each user but doesn't change the session ID after login, the attacker can overwrite the user's session ID with her own, so that once the user logs in, the attacker's session ID will be associated with the user.

An active network attacker can overwrite cookies by mounting a man-in-the-middle (MITM) attack. The attack workflow looks like this:

1. User visits a page on attack site `evil.com`, which includes an image tag with URL `http://bank.com`.

2. The browser loads the image by sending a GET request to `bank.com` over *HTTP*.

3. The active network attacker poses as `bank.com`, sending back a response with a `Set-Cookie` header, setting the session ID in the browser.

4. When user then logs onto the actual `bank.com`, the attacker-supplied session ID is sent to the server and gets associated with the user's session.

Note that this attack works even if the site is served entirely over HTTPS: as long as the browser is made to send an HTTP request to `bank.com`, the connection can be hijacked by the attacker, who can inject a forged cookie. The attacker is, furthermore, free to install `Secure` cookies, which will subsequently be sent to the actual `bank.com` over HTTPS.

This particular session fixation attack can be fixed by changing the session ID after login. A more general solution, however, is needed to address the overarching cookie overwriting vulnerability. Here we describe two different angles from which to defend against cookie overwriting.

Cookie integrity. Barth, et al. propose a `Cookie-Integrity` header; it indicates which cookies in the request's `Cookie` header were set over HTTPS. This way, when an attacker overwrites the session ID cookie over HTTP, the browser will make it clear to the website's actual server that the session ID wasn't set over HTTPS; the server can then reject the cookie. Unfortunately, no browser currently implements the `Cookie-Integrity` header.

HSTS header. A website can prevent browsers from connecting to it over HTTP by sending an *HTTP Strict Transport Security* (HSTS) header. Upon seeing this header, the browser prevents any future HTTP connections to the specific domain, instead communicating exclusively over HTTPS; this behavior cannot be overridden by the user. An HSTS header looks like this:

```
Strict-Transport-Security: max-age=86400.
```

HSTS enforcement expires after the max-age specified in the header.

In order for this defense to work, the browser has to have received the HSTS header from the legitimate server *before* the attack happens, i.e., the user needs to connect to the legitimate server *first*. One way to work around this restriction is to use the *HSTS preload list*[4], which is a list of sites hardcoded into most major browsers as being HTTPS only. This way, the browser knows a priori that a site is HTTPS only, without having to receive an HSTS header form it first.

Despite the enhanced security provided by HTTPS and the HSTS header, transitioning a website to using HTTPS exclusively is hard. One barrier to adopting HTTPS is that many sites contain contents that must be served over HTTP (e.g., CDN and ads); even if a page is served over HTTPS, its security can be compromised if parts of its contents are not tamper-proof. Furthermore, sites may be hesitant to use HSTS because HSTS enforcement is not easily reversible: if a site wishes to switch back to using HTTP (e.g., due to certificate issues), it would have to wait until HSTS expires.

[4] `https://hstspreload.org/`.