Welcome to

8. Web Application Security: Defensive: XSS, CSRF, tokens etc.

Security in Web Development Elective, KEA

Henrik Kramselund he/him han/ham hkj@zencurity.com @kramse

Slides are available as PDF, kramse@Github

9-defensive-XXS-CSRF-security-in-web.tex in the repo security-courses

Goals for today

Todays goals:

- Doing defensive for protecting the session
- Defending Against XSS, CSRF and XXE Attacks

Photo by Thomas Galler on Unsplash

Plan for today

Subjects

- Authentication
- anti-XSS
- Content Security Policies
- Security headers
- XML External Entity Prevention

Exercises

- Security headers for protecting against security issues

Time schedule

- 1) Defending Against XSS Attacks + exercise 60-70min
- 2) Defending Against CSRF Attacks 45min
- 3) XXE and deserialization 30min
- 4) Researching protection 30-45min

We will not follow this to the minute, but be flexible

Reading Summary

Web Application Security, Andrew Hoffman, 2020, ISBN: 9781492053118

Part III. Defense, chapters 19-21

22. Defending Against XSS Attacks

23. Defending Against CSRF Attacks

24. Defending Against XXE

22. Defending Against XSS Attacks

Cross-Site Scripting (XSS) vulnerabilities are some of the most common vulnerabilities throughout the internet, and have appeared as a direct response to the increasing amount of user interaction in today's web applications.

At its core, an XSS attack functions by taking advantage of the fact that web applications execute scripts on users' browsers. Any type of dynamically created script that is executed puts a web application at risk if the script being executed can be contaminated or modified in any way—in particular by an end user.

XXS attacks are categorized a number of ways, with the big three being:

- Stored (the code is stored on a database prior to execution)
- Reflected (the code is not stored in a database, but reflected by a server)
- DOM-based (code is both stored and executed in the browser)

Source: *Web Application Security*, Andrew Hoffman, 2020, ISBN: 9781492053118

Authentication and Authorization Systems

In a world where most applications consist of both clients (browsers/phones) and servers, and servers persist data originally sent from a client, systems must be in place to ensure that future access of persisted data comes from the correct user.

We use the term **authentication** to describe a flow that allows a system to **identify a user**. In other words, authentication systems tell us that “joe123” is actually “joe123” and not “susan1988.”

The term **authorization** is used to describe a flow inside a system for **determining what resources “joe123” has access to, as opposed to “susan1988.”** For example, “joe123” should be able to access his own uploaded private photos, and “susan1988” should be able to access hers, but they should not be able to access each other’s photos.

Source: *Web Application Security*, Andrew Hoffman, 2020, ISBN: 9781492053118

Authentication Systems

- HTTP Basic

  The header consists of a string containing `Basic: <base64-encoded user name:password>`
- HTTP Digest Access Authentication

  cryptographic hashes instead of base64 encoding

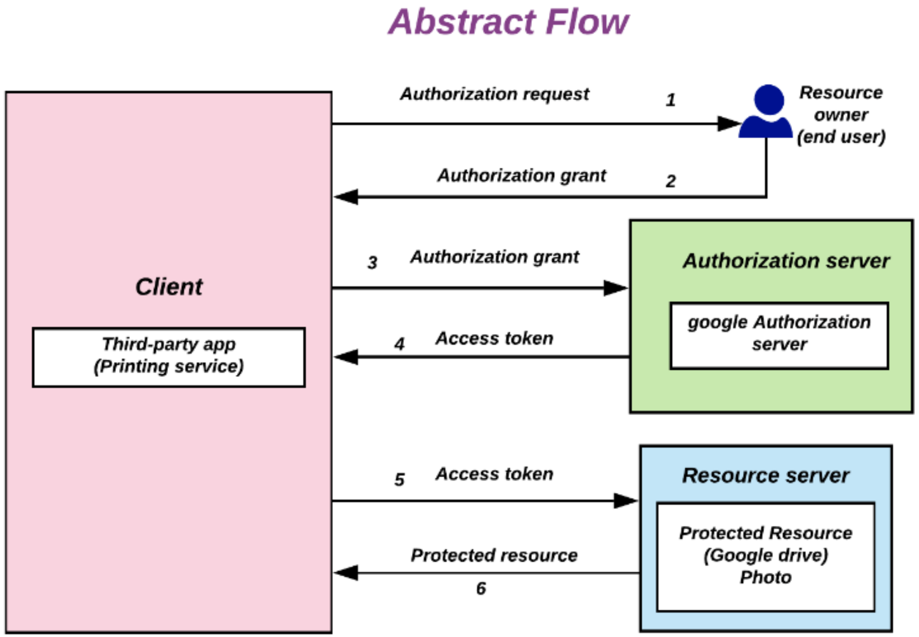
  RFC 2069 *An Extension to HTTP: Digest Access Authentication*

```
WWW-Authenticate: Digest realm="testrealm@host.com",
                        qop="auth,auth-int",
                        nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
                        opaque="5ccc069c403ebaf9f0171e9517f40e41"
```

- Today we often see a mix of local schemes, independent or centralized/federated like OAUTH

  https://en.wikipedia.org/wiki/OAuth
- Many web sites also use multi-factor (MFA) / two-factor (2FA) authentication to prevent brute force.

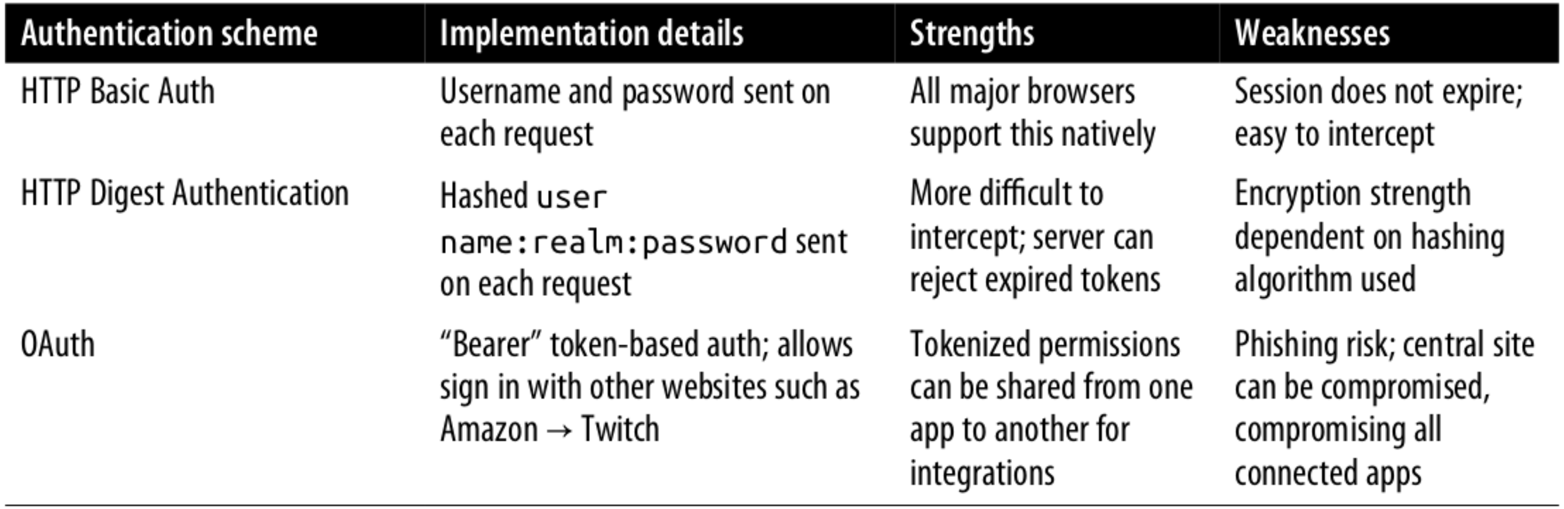
A high-level overview of Oauth 2.0 flow



Source: https://en.wikipedia.org/wiki/OAuth#/media/File:Abstract-flow.png

Major Authentication Schemes

Table 5-2. Major authentication schemes



Authentication scheme	Implementation details	Strengths	Weaknesses
HTTP Basic Auth	Username and password sent on each request	All major browsers support this natively	Session does not expire; easy to intercept
HTTP Digest Authentication	Hashed `username:realm:password` sent on each request	More difficult to intercept; server can reject expired tokens	Encryption strength dependent on hashing algorithm used
OAuth	"Bearer" token-based auth; allows sign in with other websites such as Amazon → Twitch	Tokenized permissions can be shared from one app to another for integrations	Phishing risk; central site can be compromised, compromising all connected apps

Source: *Web Application Security*, Andrew Hoffman, 2020, ISBN: 9781492053118

Anti-XSS Coding Best Practices

> There is one major rule you can implement in your development team in order to dramatically mitigate the odds of running into XSS vulnerabilities: “don’t allow any user-supplied data to be passed into the DOM—except as strings.”
>
> Such a rule is not applicable to all applications, as many applications have features that incorporate users to DOM data transfer. In this case, we can make this rule more specific: “never allow any unsanitized user-supplied data to be passed into the DOM.”
>
> Allowing user-supplied data to populate the DOM should be a fallback, last-case option rather than a first option. Such functionality will accidentally lead to XSS vulnerabilities, so when other options are available, they should be chosen first.

Source: *Web Application Security*, Andrew Hoffman, 2020, ISBN: 9781492053118

- User input should not be scripts, escape input, validate input

Avoid DOM APIs that convert text to DOM/text to script

A good rule of thumb for DOM APIs to be aware of in your sanitization is anything that converts text to DOM or text to script is a potential XSS attack vector. Stay away from the following APIs when possible:

- element.innerHTM‘L / ‘element.outerHTML
- Blob
- SVG
- document.write / document.writeln
- DOMParser.parseFromString
- document.implementation

- I recommend using proper input sanitizing, as described – not the tricks in the early part of the chapter.

HTML Entity Encoding – escaping characters

Another preventative measure that can be applied is to perform HTML entity escap- ing on all HTML tags present in user-supplied data. Entity encoding allows you to specify characters to be displayed in the browser, but in a way that they cannot be interpreted as JavaScript. The “big five” for entity encoding are shown in Table 22-1.

- `& &amp;` ampersand
- `< &lt;` – less than
- `> &gt;` – greater than
- `" &#034;`
- `' &#039;`

Burp has encoding/decoding built-in for checking, and abusing this.

Recommend using facilities in your chosen language, like https://www.php.net/manual/en/function.htmlspecialchars.php and https://www.php.net/manual/en/function.htmlentities.php

Content Security Policy (CSP) for XSS Prevention

The **CSP is a security configuration tool that is supported by all major browsers.** It provides settings that a developer can take advantage of to either **relax or harden security rules regarding what type of code can run inside your application.**

CSP protections come in several forms, including **what external scripts can be loaded, where they can be loaded, and what DOM APIs are allowed to execute the script**.

Let's evaluate some CSP configurations that aid in mitigating XSS risk.

- Very highly recommended, easy to check, quite easy to implement
- Tools available for you to generate even complex CSPs
- and lots of references:
  https://developers.google.com/web/fundamentals/security/csp
  https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

Implementing CSP

> To enable CSP, you need to configure your web server to return the Content-Security-Policy HTTP header. (Sometimes you may see mentions of the X-Content-Security-Policy header, but that's an older version and you don't need to specify it anymore.)
>
> Alternatively, the <meta> element can be used to configure a policy, for example:

```
<meta http-equiv="Content-Security-Policy"
      content="default-src 'self'; img-src https://*; child-src 'none';">
```

Source: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

- I usually see this as part of the headers
- Other headers can also be checked using Mozilla Observatory

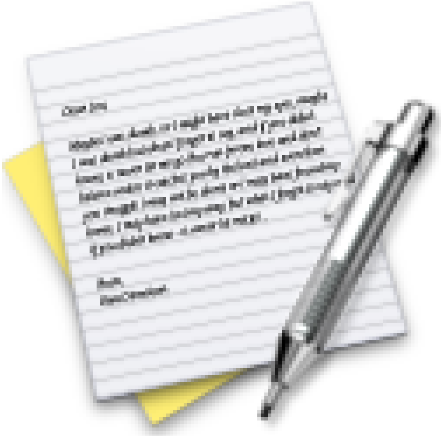
Scanning for HTTP settings

Nmap can also report these settings:

```
|     X-XSS-Protection: 1;mode=block
|     Content-Security-Policy: script-src 'self' 'unsafe-inline' 'unsafe-eval'
|     X-Content-Type-Options: nosniff
```

- If you have many sites, Nmap can also report this
- Use the documentation from Mozilla Observatory for full explainations about these settings

Exercise



Now lets do the exercise

Identify Session Tokens 20 min

which is number **42** in the exercise PDF.

23. Defending Against CSRF Attacks

> In Part II we built Cross-Site Request Forgery (CSRF) attacks that took advantage of a user's authenticated session in order to make requests on their behalf. We built CSRF attacks with <a></a> links, via <img></img> tags, and even via HTTP POST using web forms. We saw how effective and dangerous CSRF-style attacks are against an application, because they function at both an elevated privilege level and often are undetectable by the authenticated user.
>
> In this chapter, we will learn how to defend our codebase against such attacks, and mitigate the probability that our users will be put at risk for any type of attack that targets their authenticated session.

Source: *Web Application Security*, Andrew Hoffman, 2020, ISBN: 9781492053118

- CSRF are often related to phishing https://en.wikipedia.org/wiki/Phishing

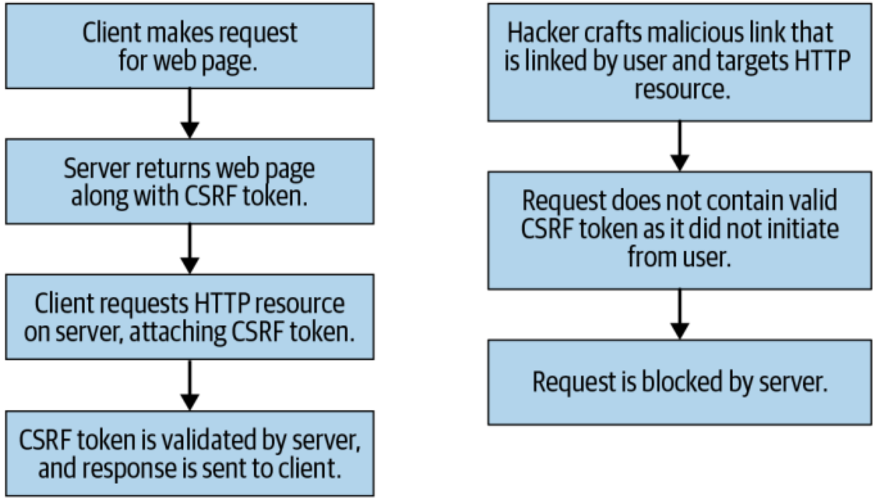
Header Verification – first check

> Because the **origin of many CSRF requests is separate from your web application**, we can mitigate the risk of CSRF attacks by checking the origin of the request. In the world of HTTP, there are two headers we are interested in when checking the origin of a request: **referer and origin . These headers are important because they cannot be modified programmatically with JavaScript in all major browsers**. As such, a properly implemented browser's referer or origin header has a low chance of being spoofed.

- Origin header The origin header is only sent on HTTP POST requests. It is a simple header that indicates where a request originated from. Unlike referer , this header is 247also present on HTTPS requests, in addition to HTTP requests.
- Referer header The referer header is set on all requests, and also indicates where a request ori- ginated from. The only time this header is not present is when the referring link has the attribute rel=noreferer set.

These headers are a first line of defense, but there is a case where they will fail.

Better: CSRF Tokens



Client makes request for web page.

Server returns web page along with CSRF token.

Client requests HTTP resource on server, attaching CSRF token.

CSRF token is validated by server, and response is sent to client.

Hacker crafts malicious link that is linked by user and targets HTTP resource.

Request does not contain valid CSRF token as it did not initiate from user.

Request is blocked by server.

Figure 23-1. CSRF tokens, the most effective and reliable method of eliminating cross-site request forgery attacks

Source: *Web Application Security*, Andrew Hoffman, 2020, ISBN: 9781492053118

- Where do you come from, where do your browser go

Anti-CRSF Coding Best Practices

There are many methods of eliminating or mitigating CRSF risk in your web applica- tion that start at the code or design phase. Several of the most effective methods are:

- Refactoring to stateless GET requests
- Implementation of application-wide CSRF defenses
- Introduction of request-checking middleware

Implementing these simple defenses in your web application will dramatically reduce the risk of falling prey to CSRF-targeting hackers.

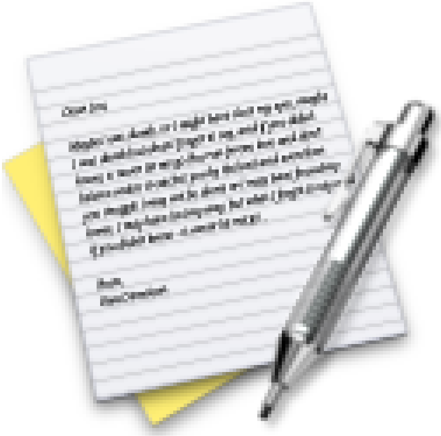
Example CSRF Protection Django

> **Cross Site Request Forgery protection**
> The CSRF middleware and template tag provides easy-to-use protection against Cross Site Request Forgeries. This type of attack occurs when a malicious website contains a link, a form button or some JavaScript that is intended to perform some action on your website, using the credentials of a logged-in user who visits the malicious site in their browser. A related type of attack, 'login CSRF', where an attacking site tricks a user's browser into logging into a site with someone else's credentials, is also covered.
>
> The **first defense against CSRF attacks is to ensure that GET requests (and other 'safe' methods, as defined by RFC 7231#section-4.2.1) are side effect free**. Requests via 'unsafe' methods, such as POST, PUT, and DELETE, can then be protected by following the steps below.

Source: https://docs.djangoproject.com/en/4.0/ref/csrf/

- Many existing options can help with CSRF protection, so make sure to include this in your architecture and planning

Repeat exercise



Lets redo the exercise, but with focus on CSRF tokens

Identify Session Tokens 20 min

which is number **42** in the exercise PDF.

24. Defending Against XXE

> Generally speaking, XXE is indeed easy to defend against—simply disable external entities in your XML parser (see Figure 24-1). How this is done depends on the XML parser in question, but is typically just a single line of configuration:
>
> ```
> factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
> ```
>
> XXE is noted by OWASP to be particularly dangerous against Java-based XML pars- ers, as many have XXE enabled by default. Depending on the language and parser you are relying on, it is possible that XXE is disabled by default.

- Note: this applies to multiple applications and formats which you wouldn't consider as XML, example CVE-2020-25042 https://www.linkedin.com/pulse/how-exploit-bigbluebutton-file-disclosure-ssrf-pawan-jaiswal
- Parsing and converting formats is dangerous
- https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=xxe
- https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html
- Related https://github.com/QubesOS/qubes-app-linux-pdf-converter

Comparing XML versus JSON

Book describes XML versus JSON,

> JSON, on the other hand, would not be practical if your application is parsing actual XML, SVG, or other XML-derived file types. It would, however, be a practical solu- tion if your application is sending standard hierarchical payloads that just happen to be in XML shape.

and concluding

> The comparison of the two formats could go on for an extensive amount of time, but you should grasp a few things right off the bat with Table 24-1:

- JSON is a much more lightweight format than XML.
- JSON offers less rigidity, but brings with it faster and easier to work with payloads.
- JSON maps to JavaScript objects, while XML more closely maps to DOM trees (as the DOM is an XML-derived format).

Which I think is a fair recommendation. XML is complex!

Exploiting insecure deserialization vulnerabilities

> Exploiting insecure deserialization vulnerabilities In this section, we'll teach you how to exploit some common scenarios using examples from PHP, Ruby, and Java deserialization. We hope to demonstrate how exploiting insecure deserialization is actually much easier than many people believe. This is even the case during blackbox testing if you are able to use pre-built gadget chains.

https://portswigger.net/web-security/deserialization/exploiting

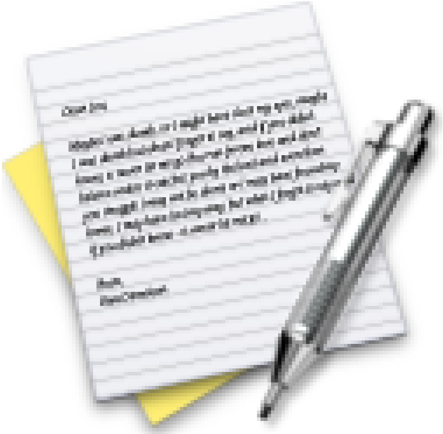
- OWASP has multiple pages about these problems: https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data
- Note:

  Platform

  Languages: C, C++, Java, Python, Ruby (and probably others)

  Operating platforms: Any

Exercise

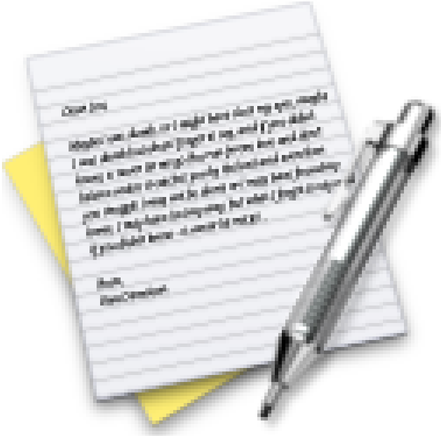


Now lets do the exercise

Research XSS and CSRF protection for your projects 30 min

which is number **43** in the exercise PDF.

Exercise



Now lets do the exercise

Optional: Run parts of a Django tutorial 30min

which is number **44** in the exercise PDF.

For Next Time

Think about the subjects from this time, write down questions

Check the plan for chapters to read in the books

Visit web sites and download papers if needed

Retry the exercises to get more confident using the tools