KonvergeAI

CCT 310 - 3
Application Security

UNIT 2
Web Application Security - I

Web App Reconnaissance

Web application reconnaissance refers to the explorative data-gathering phase that generally occurs prior to hacking a web application. Web application reconnaissance is typically performed by hackers, pen testers, or bug bounty hunters, but can also be an effective way for security engineers to find weakly secured mechanisms in a web application and patch them before a malicious actor find them. Reconnaissance (recon) skills by themselves do not have significant value, but become increasingly valuable when coupled with offensive hacking knowledge and defensive security engineering experience.

Information: *Basically we'll try to gain information about organization's digital footprints, like their IP addresses, DNS records, mail server,sub domains , older snapshots of an web application , backend technologies, server information, publicly disclosed vulnerabilities in the softwares being used etc.*

Target: *Our target is nothing but web application on which we'll perform testing.*

Active Reconnaissance: *It means whenever we engage with target to get information is called active reconnaissance.*

Passive Reconnaissance: *It means when we collect publicly available information about target without engaging with target is known as passive reconnaissance.*

Vulnerability: *Vulnerability is nothing but the weakness or lack of security which we found in the target.*

Web Application Mapping: build up a map that represents the structure, organization, and functionality of a web application. It is important to note that this should generally be the first step you take before attempting to hack into a web application.

Tools :- https://geekflare.com/reconnaissance-exploit-search-tools/

API Analysis

End Point Discovery

Table 5-1. HTTP verbs that REST architecture supports

REST HTTP Verb	Usage
POST	Create
GET	Read
PUT	Update/Replace
PATCH	Update/Modify
DELETE	Delete

Using the knowledge of what HTTP verbs are supported by the architecture spec, we can look at the requests we found in the browser console targeting particular resources. Then we can attempt to make requests to those resources using different HTTP verbs and see if the API returns anything interesting.

The HTTP specification defines a special method that only exists to give information about a particular API's verbs. This method is called `OPTIONS`, and should be our first go-to when performing recon against an API. We can easily make a request in curl from the terminal:

```
curl -i -X OPTIONS https://api.mega-bank.com/users/1234
```

Authentication Mechanisms

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

Third Party Dependencies

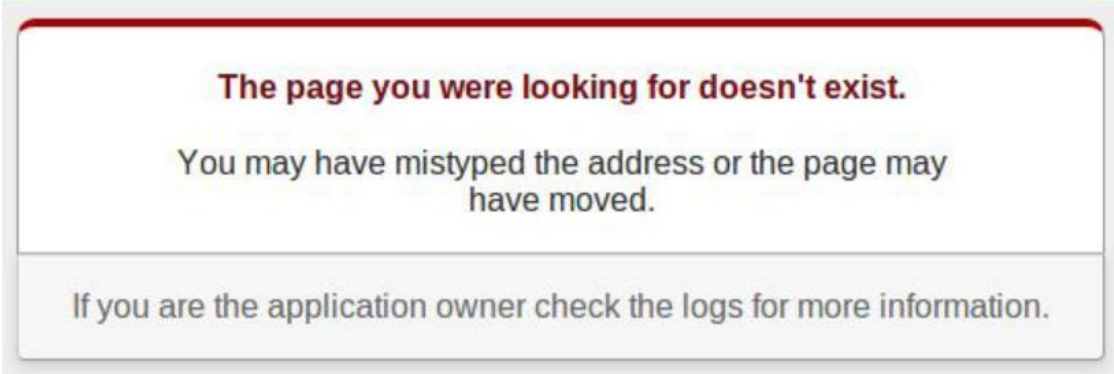
Detecting Client-Side Frameworks

- Detecting Javascript Libraries - EmberJS (LinkedIn, Netflix)
- AngularJS (Google)
- React (Facebook)
- VueJS (Adobe, GitLab)

Detecting CSS Libraries

Detecting Server-Side Frameworks - Database Detection

```
X-Powered-By: ASP.NET
```



`GET users/:id`
Where `:id` is a primary key

`PUT users, body = { id: id }`
Where `id` again is a primary key

`GET users?id=id`
Where the `id` is a primary key but in the query params

Sometimes the `ids` will appear in places you least expect them, such as in meta-data or in a response regarding a user object:

```
{
  _id: '507f1f77bcf86cd799439011',
  username: 'joe123',
  email: 'joe123@my-email.com',
  role: 'moderator',
  biography: '...'
}
```

Architecture Weak Points

- Technology used in the web application
- List of API endpoints by HTTP verb
- List of API endpoint shapes (where available)
- Functionality included in the web application (e.g., comments, auth, notifications, etc.)
- Domains used by the web application
- Configurations found (e.g., Content Security Policy or CSP)
- Authentication/session management systems

Secure Versus Insecure Architecture Signals

Multiple Layers of Security

- API POST
- Database Write
- Database Read
- API GET
- Client Read

OWASP Secure Coding Checklist :- https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/

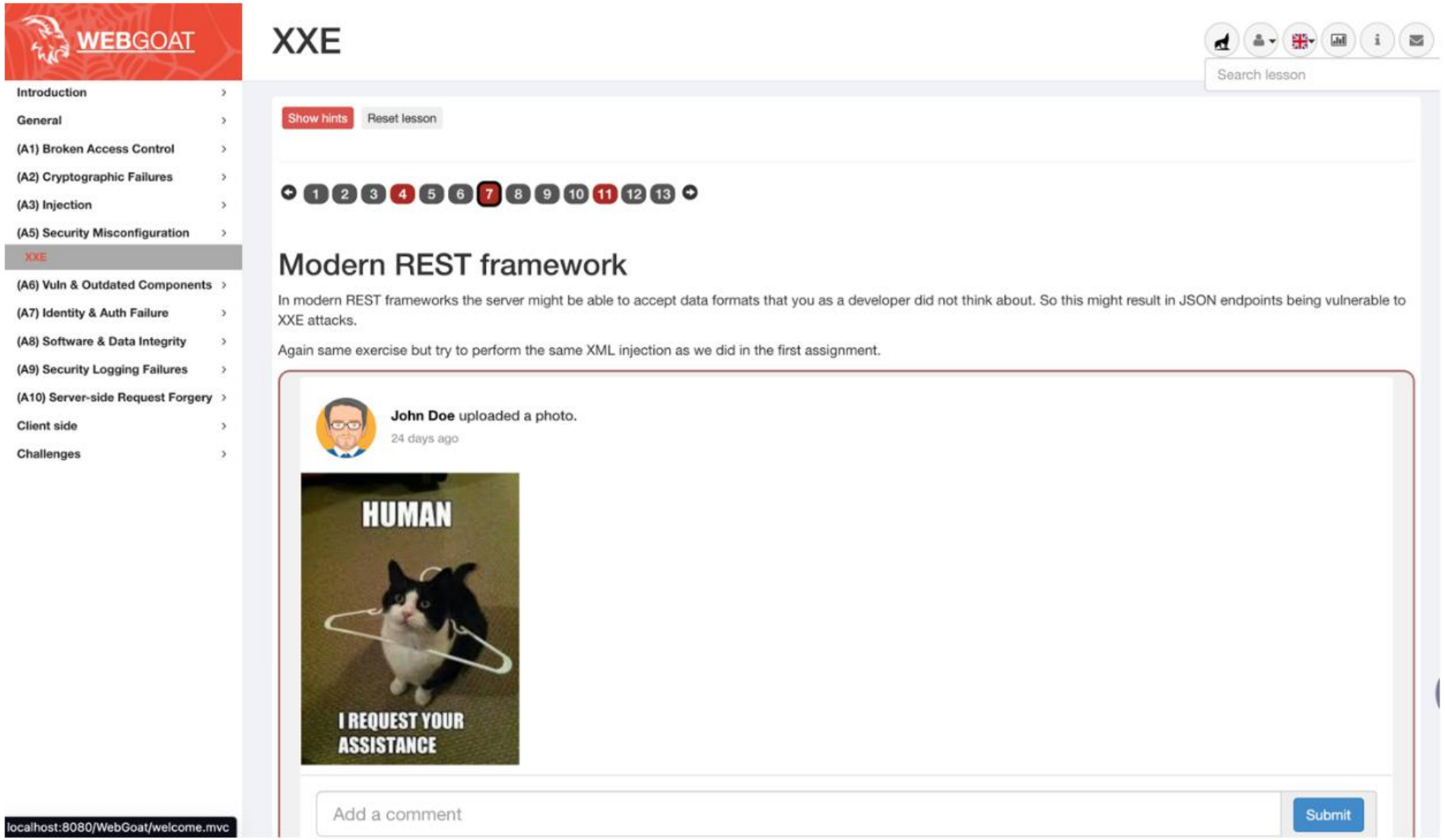
Secure Coding Best Practices

Input Validation
Output Encoding
Authentication and Password Management
Session Management
Access Control
Cryptographic Practices
Error Handling and Logging
Data Protection
Communication Security
System Configuration
Database Security
File Management
Memory Management
General Coding Practices/Code Push

OWASP Top 10/Sans25

https://owasp.org/www-project-top-ten/

https://www.sans.org/top25-software-errors/


WEBGOAT
XXE
Search lesson
Introduction
General
(A1) Broken Access Control
(A2) Cryptographic Failures
(A3) Injection
(A5) Security Misconfiguration
XXE
(A6) Vuln & Outdated Components
(A7) Identity & Auth Failure
(A8) Software & Data Integrity
(A9) Security Logging Failures
(A10) Server-side Request Forgery
Client side
Challenges
Show hints
Reset lesson
1 2 3 4 5 6 7 8 9 10 11 12 13
Modern REST framework
In modern REST frameworks the server might be able to accept data formats that you as a developer did not think about. So this might result in JSON endpoints being vulnerable to XXE attacks.
Again same exercise but try to perform the same XML injection as we did in the first assignment.
John Doe uploaded a photo.
24 days ago
HUMAN
I REQUEST YOUR ASSISTANCE
Add a comment
Submit
localhost:8080/WebGoat/welcome.mvc