**API Testing:**

**Step 1: Content discovery**

Read API docs if there is any. Search github as well with relevant keywords. There can also be machine-readable documentation in JSON or XML that could be useful.

We can also discover content by simply using the application and seeing what API requests are made while doing so. We will also likely want to fuzz the api maybe even try to spider it. (seclists/webcontent/api) Spidering the application itself could also uncover api endpoints. Fuzzign should be done after initial app exploration once we have manually discovered some endpoints. The fuzzing should really be done only on the most interesting endpoints like ones that reference the user.

\*\* if we find a valid endpoint, we will want to look at all directories in the valid path as well because this could lead to new endpoints. \*\*

\*\* using xnlinkfinder on JS will likely also uncover api endpoints or at least paths that could be used in an API call. **NEED to start generating website specific wordlists for fuzzing using linkfinder and GAP extension in burp** \*\*

**Step 2: Interacting With the API**

This first step in this process is to test all http methods to potentially identify new functionality and vulnerabilities.

We can fuzz the request using a list of all HTTP verbs (likely best to do this in ZAP) or manually change them ourselves. We should try GET, POST, OPTIONS, DELETE.

We should also try PUT which replaces an existing resource with the new data in that request and **PATCH** which replaces part of an existing resource with the new data in that request.

**\*\*\* PATCH is very useful in API testing as this is often used to change various fields always check to see if this method is allowed. \*\*\***

Other methods: HEAD, TRACE, CONNECT

\*\* try to target low-priority objects so as not to alter critical items or create excessive records. \*\*

For example: if we see a price being fetched by an API and see the price appear in a JSON object within the response, we can try to change the HTTP method to PATCH and add content type application/json along with our own JSON object to try to change the price. This is extrenly easy to do with **the change content type burp extension** literally just right click, extensions, change content on any request and it will automatically add everything needed for a valid JSON/XML request.

**Identifying supported content types**

API endpoints often expect data in a specific format. They may therefore behave differently depending on the content type of the data provided in a request. Changing the content type may enable you to:

* Trigger errors that disclose useful information.
* Bypass flawed defenses.
* Take advantage of differences in processing logic. For example, an API may be secure when handling JSON data but susceptible to injection attacks when dealing with XML.

To change the content type, modify the Content-Type header, then reformat the request body accordingly. You can use the [Content type converter](https://portswigger.net/bappstore/db57ecbe2cb7446292a94aa6181c9278) BApp to automatically convert data submitted within requests between XML and JSON.

\*\* **So essentially if we see JSON data being passed we can use this extension to quickly convert it to XML and begin to test for XXE vulnerabilities. (of course we will need to refer back to the XXE labs and notes to refamiliarize with it**\*\*

**Finding Hidden Parameters:**

Use param miner on API endpoints to guess query params. Param miner will automatically guess names relevant to the application based on information from apps in the scope.

Even in JSON data within the request body (or any POST request for that matter) param miner as an option “guess body params” to test for this. Param Miner has very specialized options for many of the vulns detailed in the cache-implementation-flaws document. Some of these can be useful in API testing likely the guess body params option will be most beneficial.

-----------------------------------------

**Mass assignment vulnerabilities**

Mass assignment (also known as auto-binding) can inadvertently create hidden parameters. It occurs when software frameworks automatically bind request parameters to fields on an internal object. Mass assignment may therefore result in the application supporting parameters that were never intended to be processed by the developer.

**Identifying hidden parameters For Mass assignment:**

Since mass assignment creates parameters from object fields, you can often identify these hidden parameters by manually examining objects returned by the API.

For example, consider a PATCH /api/users/ request, which enables users to update their username and email, and includes the following JSON:

{

"username": "wiener",

"email": "wiener@example.com",

}

A concurrent GET /api/users/123 request returns the following JSON:

{

"id": 123,

"name": "John Doe",

"email": "john@example.com",

"isAdmin": "false"

}

This may indicate that the hidden id and isAdmin parameters are bound to the internal user object, alongside the updated username and email parameters.

\*\*\* Essentially if we see certain fields in the response when trying to access a specific object \*\*\*

**Testing mass assignment vulnerabilities**

To test whether you can modify the enumerated isAdmin parameter value, add it to the PATCH request:

{

"username": "wiener",

"email": "wiener@example.com",

"isAdmin": false,

}

In addition, send a PATCH request with an invalid isAdmin parameter value:

{

"username": "wiener",

"email": "wiener@example.com",

"isAdmin": "foo",

}

If the application behaves differently, this may suggest that the invalid value impacts the query logic, but the valid value doesn't. This may indicate that the parameter can be successfully updated by the user.

You can then send a PATCH request with the isAdmin parameter value set to true, to try and exploit the vulnerability:

{

"username": "wiener",

"email": "wiener@example.com",

"isAdmin": true,

}

If the isAdmin value in the request is bound to the user object without adequate validation and sanitization, the user wiener may be incorrectly granted admin privileges. To determine whether this is the case, browse the application as wiener to see whether you can access admin functionality.

**\*\*\* Again this is a very simple vulnerability that leverages the PATCH method to update fields that could result in potential priv escalation or potentially stored/blind xss \*\*\***

**Essentially we need to observe API responses particularly ones with JSON objects in the response. We then try to change fields either by using PATCH, PUT or POST. Often the original request will not contain JSON just the response. Use content type coverter to add JSON and even try to copy the entire JSON object in the response to the request body. Change fields around and look for unexpecvted behavior. This is very good for interfering with business logic.**