API recon

To start API testing, you first need to find out as much information about the API as possible, to discover its attack surface.

To begin, you should identify API endpoints. These are locations where an API receives requests about a specific resource on its server

```
GET /api/books HTTP/1.1
Host: example.com
```

Once you have identified the endpoints, you need to determine how to interact with them. This enables you to construct valid HTTP requests to test the API.

- The input data the API processes, including both compulsory and optional parameters.
- The types of requests the API accepts, including supported HTTP methods and media formats.
- Rate limits and authentication mechanisms.

API documentation

Documentation can be in both human-readable and machine-readable forms.

Human-readable documentation is designed for developers to understand how to use the API

Machine-readable documentation is designed to be processed by software for automating tasks like API integration and validation. It's written in structured formats like JSON or XML

Discovering API documentation

Even if API documentation isn't openly available, you may still be able to access it by browsing applications that use the API.(use Burp Scanner to crawl the API)

endpoints that may refer to API documentation

- `/api`
- `/swagger/index.html`
- `/openapi.json`

You could also use a list of common paths to directly fuzz for documentation.

Using machine-readable documentation

You can use Burp Scanner to crawl and audit OpenAPI documentation, or any other documentation in JSON or YAML format. You can also parse OpenAPI documentation using the OpenAPI Parser BApp.

You may also be able to use a specialized tool to test the documented endpoints, such as Postman or SoapUI

Identifying API endpoints

While browsing the application, look for patterns that suggest API endpoints in the URL structure, such as `/api/`

Interacting with API endpoints

For example, you could investigate how the API responds to changing the HTTP method and media type.

As you interact with the API endpoints, review error messages and other responses closely. Sometimes these include information that you can use to construct a valid HTTP request

Identifying supported HTTP methods

The HTTP method specifies the action to be performed on a resource.

- `GET` - Retrieves data from a resource.
- `PATCH` - Applies partial changes to a resource.
- `OPTIONS` - Retrieves information on the types of request methods that can be used on a resource.

An API endpoint may support different HTTP methods. It's therefore important to test all potential methods when you're investigating API endpoints.

- `GET /api/tasks` - Retrieves a list of tasks.
- `POST /api/tasks` - Creates a new task.
- `DELETE /api/tasks/1` - Deletes a task.

Identifying supported content types

API endpoints often expect data in a specific format

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.

You can use the Content type converter BApp to automatically convert data submitted within requests between XML and JSON.

LAB : Finding and exploiting an unused API endpoint

hidden API endpoint to buy a **Lightweight l33t Leather Jacket**

Use the HTTP History in Burp to find "api" Redirects.

```
/api/products/3/price
```

Send this Resquest to Repeater , SInce this is a GET Request which Means it is taking out some specified value from the server. So we need to check for the HTTP methods will the api allows so we can try to modified some data.

We send OPTIONS HTTP Method request to analyis the Errors.

Change the method for the API request from `GET` to `PATCH`, then send the request. Notice that you receive an `Unauthorized` message.

log in to the application using the credentials `wiener:peter`.Notice that this causes an error due to an incorrect `Content-Type`.

1. Add a `Content-Type` header and set the value to `application/json`.
2. Add an empty JSON object `{}`

Notice that this causes an error due to the request body missing a `price` parameter.

Add a `price` parameter with a value of `0` to the JSON object `{"price":0}`.

Fuzzing to find hidden endpoints

Once you have identified some initial API endpoints, you can fuzz to uncover hidden endpoints.

you could fuzz the `/update` position of the path with a list of other common functions, such as `delete` and `add`.

Finding hidden parameters

When you're doing API recon, you may find undocumented parameters that the API supports.

- Burp Intruder enables you to fuzz for hidden parameters, using a wordlist of common parameter names to replace existing parameters or add new parameters
- Param miner BApp enables you to automatically guess up to 65,536 param names per request.
- The Content discovery tool enables you to discover content that isn't linked from visible content that you can browse to, including parameters

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

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

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.

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.

Preventing vulnerabilities in APIs

- Secure your documentation if you don't intend your API to be publicly accessible.
- Ensure your documentation is kept up to date so that legitimate testers have full visibility of the API's attack surface.
- Apply an allowlist of permitted HTTP methods.
- Validate that the content type is expected for each request or response.
- Use generic error messages to avoid giving away information that may be useful for an attacker.
- Use protective measures on all versions of your API, not just the current production version.

Server-side parameter pollution

Some systems contain internal APIs that aren't directly accessible from the internet. Server-side parameter pollution occurs when a website embeds user input in a server-side request to an internal API without adequate encoding.

which may enable them to, for example:

- Override existing parameters.
- Modify the application behavior.
- Access unauthorized data.

Truncating query strings

You can use a URL-encoded `#` character to attempt to truncate the server-side request.

For example, you could modify the query string to the following:

`GET /userSearch?name=peter%23foo&back=/home`

```
GET /users/search?name=peter#foo&publicProfile=true
```

if the response returns the user `peter`, the server-side query may have been truncated. If an `Invalid name` error message is returned, the application may have treated `foo` as part of the username. This suggests that the server-side request may not have been truncated.

Injecting invalid parameters

You can use an URL-encoded `&` character to attempt to add a second parameter to the server-side request.

For example, you could modify the query string to the following:

```
GET /userSearch?name=peter%26foo=xyz&back=/home
```

This results in the following server-side request to the internal API:

```
GET /users/search?
name=peter&foo=xyz&publicProfile=true
```

Review the response for clues about how the additional parameter is parsed. For example, if the response is unchanged this may indicate that the parameter was successfully injected but ignored by the application.

For example, if you've identified the `email` parameter, you could add it to the query string as follows:

```
GET /userSearch?name=peter%26email=foo&back=/home
```

This results in the following server-side request to the internal API:

```
GET /users/search?name=peter&email=foo&publicProfile=true
```

Overriding existing parameters

To confirm whether the application is vulnerable to server-side parameter pollution, you could try to override the original parameter.

```
GET /userSearch?name=peter%26name=carlos&back=/home
GET /users/search?name=peter&name=carlos&publicProfile=true
```

The internal API interprets two `name` parameters. The impact of this depends on how the application processes the second parameter.

- PHP parses the last parameter only.
- ASP.NET combines both parameters.
- Node.js / express parses the first parameter only.

Lab: Exploiting server-side parameter pollution in a query string

To solve the lab, log in as the `administrator` and delete `carlos`.

```
POST /forgot-password WHERE username=administrator
```

Lets Add a '#' at the end of administrator to check if it ask for another parameter.

It Returns Field Parameter Not Specified , Which told us that 'field' parameter exist.

```
username=administrator%26field=reset-token
```

We Came to know about reset-token from the /forgot-password/js file .

After POST this username request it Generates a token

We Searched for :

```
POST /forgot-password?reset_token=vklnwkqpou78htik8q513xfrx9yfh8r7
```

Testing for server-side parameter pollution in REST paths

A RESTful API may place parameter names and values in the URL path, rather than the query string.

```
/api/users/123
```

- `/api` is the root API endpoint.
- `/users` represents a resource, in this case `users`.
- `/123`represents a parameter, here an identifier for the specific user.

Requests are sent to the following endpoint:

```
GET /edit_profile.php?name=peter
```

This results in the following server-side request:

```
GET /api/private/users/peter
```

An attacker may be able to manipulate server-side URL path parameters to exploit the API. To test for this vulnerability, add path traversal sequences to modify parameters

submit URL-encoded `peter/../admin` as the value of the `name` parameter:

```
GET /edit_profile.php?name=peter%2f..%2fadmin
```

This may result in the following server-side request:

```http
GET /api/private/users/peter/../admin
```

If the server-side client or back-end API normalize this path, it may be resolved to `/api/private/users/admin`.

Testing for server-side parameter pollution in structured data formats

An attacker may be able to manipulate parameters to exploit vulnerabilities in the server's processing of other structured data formats, such as a JSON or XML

When you edit your name, your browser makes the following request

```
POST /myaccount?name=peter
```

This results in the following server-side request:

```
PATCH /users/7312/update
{"name":"peter"}
```

You can attempt to add the `access_level` parameter to the request as follows:

```
POST /myaccount?name=peter","access_level":"administrator
```

```http
PATCH /users/7312/update
{name="peter","access_level":"administrator"}
```

In JSON, the backslash (`\`) is used as an escape character.

`\"`: Represents a double quote character (`"`).

`\\`: Represents a single backslash character.

`\/`: Represents a solidus (forward slash) character (`/`)

```
POST /myaccount
{"name": "peter\",\"access_level\":\"administrator"}
```

This example below is in JSON, but server-side parameter pollution can occur in any structured data format. For an example in XML, see the XInclude attacks section in the XML external entity (XXE) injection topic.