

htb academy stuff

- [Sensitive Data Exposure](#)

Sensitive Data Exposure

All of the `front end` components we covered are interacted with on the client-side. Therefore, if they are attacked, they do not pose a direct threat to the core `back end` of the web application and usually will not lead to permanent damage. However, as these components are executed on the `client-side`, they put the end-user in danger of being attacked and exploited if they do have any vulnerabilities. If a front end vulnerability is leveraged to attack admin users, it could result in unauthorized access, access to sensitive data, service disruption, and more.

Although the majority of web application penetration testing is focused on back end components and their functionality, it is important also to test front end components for potential vulnerabilities, as these types of vulnerabilities can sometimes be utilized to gain access to sensitive functionality (i.e., an admin panel), which may lead to compromising the entire server.

[Sensitive Data Exposure](#) refers to the availability of sensitive data in clear-text to the end-user. This is usually found in the `source code` of the web page or page source on the front end of web applications. This is the HTML source code of the application, not to be confused with the back end code that is typically only accessible on the server itself. We can view any website's page source in our browser by right-clicking anywhere on the page and selecting `View Page Source` from the pop-up menu. Sometimes a developer may disable right-clicking on a web application, but this does not prevent us from viewing the page source as we can merely type `ctrl + u` or view the page source through a web proxy such as `Burp Suite`. Let's take a look at the google.com page source. Right-click and choose `View Page Source`, and a new tab will open in our browser with the URL `view-source:https://www.google.com/`. Here we can see the `HTML`, `JavaScript`, and external links. Take a moment to browse the page source a bit.

[illegible]

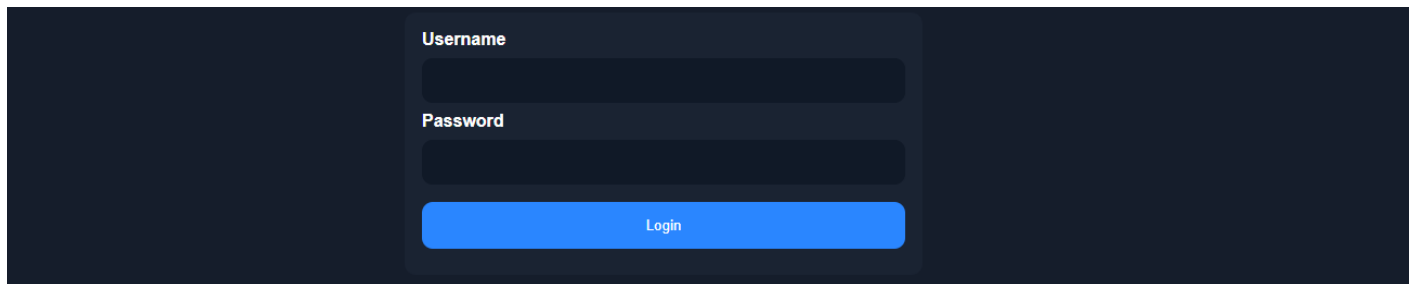
metimes we may find login `credentials`, `hashes`, or other sensitive data hidden in the comments of a web page's source code or within external `JavaScript` code being imported. Other sensitive information may include exposed links or directories or even exposed user information, all of which can potentially be leveraged to further our access within the web application or even the web

application's supporting infrastructure (webserver, database server, etc.).

For this reason, one of the first things we should do when assessing a web application is to review its page source code to see if we can identify any 'low-hanging fruit', such as exposed credentials or hidden links.

Example

At first glance, this login form does not look like anything out of the ordinary:



```
<form action="action_page.php" method="post">

  <div class="container">
    <label for="uname"><b>Username</b></label>
    <input type="text" required>

    <label for="psw"><b>Password</b></label>
    <input type="password" required>

    <!-- TODO: remove test credentials test:test -->

    <button type="submit">Login</button>
  </div>
</form>

</html>
```

We see that the developers added some comments that they forgot to remove, which contain test credentials:

```
<!-- TODO: remove test credentials test:test -->
```

The comment seems to be a reminder for the developers to remove the test credentials. Given that the comment has not been removed yet, these credentials may still be valid.

Although it is not very common to find login credentials in developer comments, we can still find various bits of sensitive and valuable information when looking at the source code, such as test pages or directories, debugging parameters, or hidden functionality. There are various automated tools that we can use to scan and analyze available page source code to identify potential paths or directories and other sensitive information.

Leveraging these types of information can give us further access to the web application, which may help us attack the back end components to gain control over the server.

Prevention

Ideally, the front end source code should only contain the code necessary to run all of the web applications functions, without any extra code or comments that are not necessary for the web application to function properly. It is always important to review the code that will be visible to end-users through the page source or run it through tools to check for exposed information.

It is also important to classify data types within the source code and apply controls on what can or cannot be exposed on the client-side. Developers should also review client-side code to ensure that no unnecessary comments or hidden links are left behind. Furthermore, front end developers may want to use `JavaScript` code packing or obfuscation to reduce the chances of exposing sensitive data through `JavaScript code`. These techniques may prevent automated tools from locating these types of data.

Another major aspect of front end security is validating and sanitizing accepted user input. In many cases, user input validation and sanitization is carried out on the back end. However, some user input would never make it to the back end in some cases and is completely processed and rendered on the front end. Therefore, it is critical to validate and sanitize user input on both the front end and the back end.

[HTML injection](#) occurs when unfiltered user input is displayed on the page. This can either be through retrieving previously submitted code, like retrieving a user comment from the back end database, or by directly displaying unfiltered user input through `JavaScript` on the front end.

When a user has complete control of how their input will be displayed, they can submit `HTML` code, and the browser may display it as part of the page. This may include a malicious `HTML` code, like an external login form, which can be used to trick users into logging in while actually sending their login credentials to a malicious server to be collected for other attacks.

Another example of `HTML Injection` is web page defacing. This consists of injecting new `HTML` code to change the web page's appearance, inserting malicious ads, or even completely changing the page. This type of attack can result in severe reputational damage to the company hosting the web application.

Example

The following example is a very basic web page with a single button "`Click to enter your name`." When we click on the button, it prompts us to input our name and then displays our name as "`Your name is ...`":

`Click to enter your name`

`Your name is user`

If no input sanitization is in place, this is potentially an easy target for `HTML Injection` and `Cross-Site Scripting (XSS)` attacks. We take a look at the page source code and see no input sanitization in place whatsoever, as the page takes user input and directly displays it:

```
<!DOCTYPE html>
<html>

<body>
  <button onclick="inputFunction()">Click to enter your name</button>
  <p id="output"></p>

  <script>
    function inputFunction() {
      var input = prompt("Please enter your name", "");

      if (input != null) {
        document.getElementById("output").innerHTML = "Your name is " + input;
      }
    }
  </script>
</body>
</html>
```

```
</script>
</body>

</html>
```

To test for `HTML Injection`, we can simply input a small snippet of `HTML` code as our name, and see if it is displayed as part of the page. We will test the following code, which changes the background image of the web page:

```
<style> body { background-image: url('https://academy.hackthebox.com/images/logo.svg'); }
</style>
```

Once we input it, we see that the web page's background image changes instantly:



In this example, as everything is being carried out on the front end, refreshing the web page would reset everything back to normal.