Intro to File Inclusions

Many modern back-end languages, such as PHP, Javascript, or Java, use HTTP parameters to specify what is shown on the web page, which allows for building dynamic web pages, reduces the script's overall size, and simplifies the code. In such cases, parameters are used to specify which resource is shown on the page. If such functionalities are not securely coded, an attacker may manipulate these parameters to display the content of any local file on the hosting server, leading to a Local File Inclusion (LFI) vulnerability.

Local File Inclusion (LFI)

The most common place we usually find LFI within is templating engines. In order to have most of the web application looking the same when navigating between pages, a templating engine displays a page that shows the common static parts, such as the `header`, `navigation bar`, and `footer`, and then dynamically loads other content that changes between pages. Otherwise, every page on the server would need to be modified when changes are made to any of the static parts. This is why we often see a parameter like `/index.php?page=about`, where `index.php` sets static content (e.g. header/footer), and then only pulls the dynamic content specified in the parameter, which in this case may be read from a file called `about.php`. As we have control over the `about` portion of the request, it may be possible to have the web application grab other files and display them on the page.

LFI vulnerabilities can lead to source code disclosure, sensitive data exposure, and even remote code execution under certain conditions. Leaking source code may allow attackers to test the code for other vulnerabilities, which may reveal previously unknown vulnerabilities. Furthermore, leaking sensitive data may enable attackers to enumerate the remote server for other weaknesses or even leak credentials and keys that may allow them to access the remote server directly. Under specific conditions, LFI may also allow attackers to execute code on the remote server, which may compromise the entire back-end server and any other servers connected to it.

Examples of Vulnerable Code

Let's look at some examples of code vulnerable to File Inclusion to understand how such vulnerabilities occur. As mentioned earlier, file Inclusion vulnerabilities can occur in many of the most popular web servers and development frameworks, like `PHP`, `NodeJS`, `Java`, `.Net`, and many others. Each of them has a slightly different approach to including local files, but they all share one common thing: loading a file from a specified path.

Such a file could be a dynamic header or different content based on the user-specified language. For example, the page may have a `?language` GET parameter, and if a user changes the language from a drop-down menu, then the same page would be returned but with a different `language` parameter (e.g. `?language=es`). In such cases, changing the language may change the directory the web application is loading the pages from (e.g. `/en/` or `/es/`). If we have control over the path being loaded, then we may be able to exploit this vulnerability to read other files and potentially reach remote code execution.

PHP

In `PHP`, we may use the `include()` function to load a local or a remote file as we load a page. If the `path` passed to the `include()` is taken from a user-controlled parameter, like a `GET` parameter, and `the code does not explicitly filter and sanitize the user input`, then the code becomes vulnerable to File Inclusion. The following code snippet shows an example of that:

Code: php

```php
if (isset($_GET['language'])) {
    include($_GET['language']);
}
```

We see that the `language` parameter is directly passed to the `include()` function. So, any path we pass in the `language` parameter will be loaded on the page, including any local files on the back-end server. This is not exclusive to the `include()` function, as there are many other PHP functions that would lead to the same vulnerability if we had control over the path passed into them. Such functions include `include_once()`, `require()`, `require_once()`, `file_get_contents()`, and several others as well.

Note: In this module, we will mostly focus on PHP web applications running on a Linux back-end server. However, most techniques and attacks would work on the majority of other frameworks, so our examples would be the same with a web application written in any other language.

NodeJS

Just as the case with PHP, NodeJS web servers may also load content based on an HTTP parameters. The following is a basic example of how a GET parameter `language` is used to control what data is written to a page:

Code: javascript

```javascript
if(req.query.language) {
    fs.readFile(path.join(__dirname, req.query.language), function (err, data) {
        res.write(data);
    });
}
```

As we can see, whatever parameter passed from the URL gets used by the `readfile` function, which then writes the file content in the HTTP response. Another example is the `render()` function in the `Express.js` framework. The following example shows uses the `language` parameter to determine which directory it should pull the `about.html` page from:

Code: js

```js
app.get("/about/:language", function(req, res) {
    res.render(`/${req.params.language}/about.html`);
});
```

Unlike our earlier examples where GET parameters were specified after a (`?`) character in the URL, the above example takes the parameter from the URL path (e.g. `/about/en` or `/about/es`). As the parameter is directly used within the `render()` function to specify the rendered file, we can change the URL to show a different file instead.

Java

The same concept applies to many other web servers. The following examples show how web applications for a Java web server may include local files based on the specified parameter, using the `include` function:

Code: jsp

```jsp
<c:if test="${not empty param.language}">
    <jsp:include file="<%= request.getParameter('language') %>" />
</c:if>
```

The `include` function may take a file or a page URL as its argument and then renders the object into the front-end template, similar to the ones we saw earlier with NodeJS. The `import` function may also be used to render a local file or a URL, such as the following example:

Code: jsp

```jsp
<c:import url= "<%= request.getParameter('language') %>"/>
```

.NET

Finally, let's take an example of how File Inclusion vulnerabilities may occur in .NET web applications. The `Response.WriteFile` function works very similarly to all of our earlier examples, as it takes a file path for its input and writes its content to the response. The path may be retrieved from a GET parameter for dynamic content loading, as follows:

Code: cs

```cs
@if (!string.IsNullOrEmpty(HttpContext.Request.Query['language'])) {
    <% Response.WriteFile("<% HttpContext.Request.Query['language'] %>"); %>
}
```

Furthermore, the `@Html.Partial()` function may also be used to render the specified file as part of the front-end template, similarly to what we saw earlier:

Code: cs

```cs
@Html.Partial(HttpContext.Request.Query['language'])
```

Finally, the `include` function may be used to render local files or remote URLs, and may also execute the specified files as well:

Code: cs

```cs
<!--#include file="<% HttpContext.Request.Query['language'] %>"-->
```

Read vs Execute

From all of the above examples, we can see that File Inclusion vulnerabilities may occur in any web server and any development frameworks, as all of them provide functionalities for loading dynamic content and handling front-end templates.

The most important thing to keep in mind is that `some of the above functions only read the content of the specified files, while others also execute the specified files`. Furthermore, some of them allow specifying remote URLs, while others only work with files local to the back-end server.

The following table shows which functions may execute files and which only read file content:

Function	**Read Content**	**Execute**	**Remote URL**
PHP			
`include()`/`include_once()`	✅	✅	✅
`require()`/`require_once()`	✅	✅	❌
`file_get_contents()`	✅	❌	✅
`fopen()`/`file()`	✅	❌	❌
NodeJS			
`fs.readFile()`	✅	❌	❌
`fs.sendFile()`	✅	❌	❌
`res.render()`	✅	✅	❌

Function	Read Content	Execute	Remote URL
Java			
`include`	✓	✗	✗
`import`	✓	✓	✓
.NET			
`@Html.Partial()`	✓	✗	✗
`@Html.RemotePartial()`	✓	✗	✓
`Response.WriteFile()`	✓	✗	✗
`include`	✓	✓	✓

This is a significant difference to note, as executing files may allow us to execute functions and eventually lead to code execution, while only reading the file's content would only let us to read the source code without code execution. Furthermore, if we had access to the source code in a whitebox exercise or in a code audit, knowing these actions helps us in identifying potential File Inclusion vulnerabilities, especially if they had user-controlled input going into them.

In all cases, File Inclusion vulnerabilities are critical and may eventually lead to compromising the entire back-end server. Even if we were only able to read the web application source code, it may still allow us to compromise the web application, as it may reveal other vulnerabilities as mentioned earlier, and the source code may also contain database keys, admin credentials, or other sensitive information.

Next ➜ ✓ Mark Complete & Next

Cheat Sheet

Table of Contents

Skills Assessment - File Inclusion

My Workstation

O F F L I N E

Start Instance

1 / 1 spawns left